Practical Web Based Deep Learning and Security by Example

Third Edition

Charlotte Harper

July 3, 2024

Foreword:

Security considerations in building software for the web are an important part of any web developer's plan and execution while engineering a prototype that is dependable, stable, and useful for practical purposes. The DOM (Document Object Markup), with it's implementation of HTML, Javascript, and SS as well as backend software implementing Python, C/C++, Java and bash, give web developers the freedom and power to create a wide variety of projects that express creativity, provide ease of use and functionality, portray humility and character, and provide ease of use as well as convenience and important services that are all attractive to the average Joe, the end user looking to kill time or get something done on the internet, usually on a touchscreen smartphone device. Most people wouldn't even know where to start when they want to build a website from scratch, they would tend to start on another person's website and build something limited in functionality, dependability, ease of use and especially creativity when they could have had all the latest powerful tools at their disposal in order to build something useful without wasting time pressing buttons, and especially wasting money paying for expensive subscriptions to software few people wanted to use anyway given it's limitations in ease of use and flexibility. If you have a few minutes to read through this book and learn what I want to teach you, or even speak with me personally about your goals and get some guidance in the right direction, and are motivated to learn to code and write your own software, take this book home and set aside some time to learn to build the next influential, powerful, streamlined and important web application, a website that is all on you and does exactly what you want and meets the needs of your audience.

About me:

I am a software developer with a wide range of experience in C/C++, Java, Python, HTML, SS and Javascript. I build websites people want to use, want to visit, and even get addicted to using just to learn, recreate and kill time, and most importantly, I sell software. If you had an idea as to exactly how you wanted a website to look and function, you were willing to support me so I can meet my own needs while I meet yours, and you are willing to cover the costs of running a website yourself, I would build you the next YouTube, TikTok, Twitter, Google, or even a high-tech security app only you can access. Instead of trying to sell you my time, I'm trying to buy yours: I want to talk you into building an app (website) yourself with the information that already exists, and teach you what you need to be an independent software developer, entrepreneur, leading a successful career in whatever field you desire. And let me be clear, the education I give you will be informal. You could go to school and learn all of this with a formal education, or even read this book in school, complete your assignments, and take away a great deal from your education, but I won't formally put you in the hot seat and ask you to complete assignments. I'm not your professor, you can think of me like a friend who wants to guide you toward a career driven by your own personal success. And I'm not selling you success either, you will need to buy it with your time. Learning to code has a steep learning curve and was never easy, or even supposed to be. You need to work as hard as you possibly can and continue to try and fail and try again even when you are frustrated in order to learn and build apps yourself. That's in the nature of code itself. Code is run by a compiler that is designed to give the programmer error messages, and these will teach you how to code, even if you are simply copying the error into your search engine and reading other people's examples. And I must say, you don't need to be extremely rich, smart, successful, or even detail oriented or organized to build an app. The computer takes care of that organization for you. You just need to persevere through the trial and error, maintain focus and work hard at what you do, and you will have a very successful career in the entirety of what you do.

Who I am:

I realize that the last section was more about learning and your take a ways from this book. Who am I exactly? That's a complicated question. I am unclear on that myself, as I suffer from medical conditions that can make it difficult for me to even code or write this book at times, while presenting challenges with socialization and identity issues that make my life more difficult when it comes to introducing myself. In short, if you are reading this book, you brought it home because you flipped through it and thought it was useful, or even if you just read this far in, to you I am a like minded individual who wants to see you succeed in everything that you do. I am an engineer myself, a software developer, and a student, and I am writing this book for other students who want to make their lives easier by having a handbook of the software they need making their lives easier by giving examples to copy that fit together like a big puzzle into a working, useful, large, functional, cohesive, and engaging app that can drive success no matter the line of business. Largely, this is what I do: I build apps to help myself and other people succeed. I am an author as well, though this is my first publication that I intend to complete in order to put my portfolio together into a useful document, and I am an artist as well.

I'll admit this to you, I'm sort of a strange person. I'm not perfect, I've had run ins with the law even leading me to leave colleges and universities and leave states in order to try to make a name for myself with more success. I am a woman by birth, I wear makeup, take photos of myself, wear dresses and other womens clothing, and I stay conscious of myself as a female by nature. I've had issues with other people in the past that lead to struggles with writing and building webapps, and I apologize that I haven't been able to get this book in your hands sooner: You needed this. You will want to read and write code that looks like mine and works like mine and does the same thing but even better, because if you can afford to buy this book instead of washing your keyboard like I do just to create a book yourself asking money for it, you have the resources you need to be successful in your life. I had all sorts of issues with family growing up, health conditions, doctors, the media, and the law, and my code deeply reflects the struggle that is feminism and female nature in a divided and frustrated world. However, this book is something I deeply care about, my baby, my portfolio, and my livelihood, so I appreciate your consideration when you take the text home and carefully pore over it in order to learn from me. Please keep in mind I am not perfect, this book will have errors, revisions, and new editions, and you will need to think with your logical brain as best you can in order to have a successful experience with my writing. Also, understand that I mean well for you even when you face challenges when writing. Think about it like this: When you can just rent a computer system to do anything you can possibly imagine in the digital space, store all the information you encounter, #$%!ye and organize it, and come to understand it, you will inevitably encounter difficulties with the information you are investing and even publishing. I tell you this because I encounter the same difficulties. Use this book at your own risk, work with your community and communities available to you to build software within a safe setting, and don't take things to personally when you fail or even succeed in the wrong way: That's how I got this far, and why I can bring you this text and help you succeed without diverging off on a path of madness that leaves me ruined, torn and framed while I encounter the ordinary problems everyone does on a global scale thanks to the paralellistic global scale of the network on which we will work, the internet. You might not be very familiar with who I am with just a few words, but I encourage you to read on, you will get to know me as you continue to read and understand me while building your own projects to complete your work. There will be no homework with this book, as long as your professors or teachers don't assign you any, but I highly encourage you to build a portfolio of projects yourself as you read along, as well as a capstone project showcasing how you can apply what you have learned. My capstone project is the basis for most of what you will read in this book, as it incorporates code from my previous projects, code I have created and learned to write methodically by hand, and a wide range of ideas and tips that have helped me succeed to the point where I can spin up a simple app that is fully featured and looks and behaves like a popular app you might see your friend or family using, on the internet, advertised to you, or in the news.

What this book is:

This book is a tutorial by example. You can find code here, instructions for how to learn to code, information on debugging code and fixing errors, troubleshooting steps, instructions on how to back up and save your code, re-deploy if anyone breaks your code, secure your code, deploy your code, build interactive websites that are entertaining, engaging, and addictive, and you will get a sense of who I am, why this is important, and how to portray yourself, your app and company image, as well as the software you build in the absolute best light to be the most attractive as possible to your end users, your website's visitors. In this book, I will demonstrate a number of examples of software design with a focus on the web as a platform as well as security. We will initiate the learning experience by building a basic project using the IX shell, with backup and scripting features. Then, we will examine a basic blog website, upgrade our blog with photo and video features as well as use these features to employ security solutions using free software, and secure our server using a pluggable authentication module (AM). We will then review file handling and processing, exploring video editing, voice donation, barcode scanning and optical character recognition, among other concepts. Along the way we will examine As which will help us make our software more useful and secure, with free and paid options. Along the way, we will explore physical security and militant tools such as firearms and munitions design and manufacturing including barrel and repeater design, turret and drone design, and other principals we will integrate with our software on the existing network in order to protect our software and demonstrate self defense and resilience. We will take breaks along the way to build games, 2D and 3D rendering engines, and work with embedded hardware in case study examples of basic dimensional rendering software and an electronic vibrating massage cast in silicone rubber respectively. Along the way, we will also employ machine learning solutions already available in order to better secure our software. We will also employ stock tools available for the web in order to streamline and secure the process. This book is a guide to your success in building a web application and integrating it with a professional network of computer and embedded mechanical systems, and overall a guide to building software and embedded hardware with no background knowledge or previous experience.

What this book isn't:

If you really want to have a website, you could just set up a simple store and sell what you need, post a blog, post photos or videos, or otherwise without ever writing a single line of code. This book isn't that. This book will teach you how to build software that is more useful, fully featured, functional and secure than any software you can already find, because it deploys the latest software that is still prototypes, may be expensive to run at a scale older companies operate at, and doesn't appeal to backwards, convoluted companies set up to make money for people who aren't really doing anything. If you follow this book closely, you will want to write code, research code, build your own apps, and you will make money from what you do. I will make money from this book, even in early stages, because it contains information people need and want to read, and are already buying when they buy or use my apps. This book won't build an app for you, but it will point you in the right direction and arm you with the tools you need and the skills and tips that will facilitate your own success in building software for the web, with every line of code you will need to write as an example, ready to be piece together into software you and your supporters, guests, clientele, friends, family, visitors, contractors, and the people of the internet want to use and support.

What you will learn:

This book will teach you how to build and sell software, really functional, useful software, media recording, security features like facial recognition, machine readable zone barcode scanning, web As to authenticate, record and render video and photos, and exchange messages like bluetooth and near field (FC) communication. This book will teach you how to use a networked computer, focusing on design linux, how to build bash code to make installing and backing up your software a seamless, automated breeze, how to build python code as a backend to serve dynamic messages, style things nicely using SS styles with Bootstrap, enable user login and interactivity through networked devices, build interactive media and network with other websites to offer security features like text messages for verification or other purposes, In scanning, image and video moderation, data microtransactions to keep your software secure, payment processing, cryptocurrency trading, asynchronous tasks, and more. You will learn how to build your own bluetooth devices, with batteries, chargers, microcontrollers, circuits, motors and sensors, using solder, wire and 3D printed as well as cast materials. I will demonstrate 3D design principals applied to additive manufacturing and tool and die making, so you are able to manufacture your own embedded, hardware devices with integrated batteries, chargers, electronic circuits, and functional outputs. and network them with bluetooth and the web. Specifically, we will examine two case studies, a vibrating massage and a homemade firearm, both programmed in OpenSCAD, which is available as a graphical interface or command line utility and can be integrated into a web for speedied results.

You will learn how to build and deploy a website from the ground up with no prior experience, make it functional, secure, beautiful, useful and most importantly practical. You will learn how to use machine learning and computer vision to make a site secure and more practical, record video and audio from your website, donate your voice, make music and modulate audio to create useful samples, and how to break through the noise by averaging other websites to build the best possible network of websites that you can link directly to yours in order to share all the useful information you have to offer, and even more importantly bring people to your software and business. This book will be focused most heavily on media, security and machine learning, which are the major three components that will help you build useful software for the web by engaging the right users and disengaging the wrong ones in a manner that is realistic, practical, hands on and engaging while also automatic, and sturdy.

This book teaches IX, specifically Design (Ubuntu), Bash shell, Python, HTML, SS, Javascript, and a number of useful software packages for Python like requests, as well as useful bash software like git and mpeg. I will also teach you how to trade cryptocurrency automatically, and take payments in cryptocurrency or from regular debit cards while even paying out your visitors a share of your revenue if you choose to do so. I will teach you how to make money from your website through advertising as well, how to ready your app for search engines and make it fast, ranked in the first ranking for what your customers will search to find you, and ranking in as many common searches as possible. I will teach you how to sell your software, advertise it, appeal to clients looking for your services, and make a name for yourself on the internet through avenues that already exist, are inexpensive, and work well. I will teach you how to save your data on cloud computers that work for you and save your data cheaply, how to plan and build a website that does what your users want and what you want, and how to keep your users engaged by putting your site a tap away on their phones with notifications, email, text messages, phone calls, and more avenues to bring your users back to your website at your disposal behind the click of a button secured to you only. This book will focus on the practically of publishing and distributing media in large amounts, from text to photos to videos to audio, making a good impression on end users (your clientele), and selling yourself in any way that you do in order to create a website, an app that is representative of you and you only, and makes you, your software and your company look good in the best way possible. You will also learn a few tips and tricks from me, from coding tips, practical vanity like makeup and photography, modeling and acting, and more, which will be important for portraying yourself and your company in the best possible light using all the tools available to you while distributing as much content as you need across a healthy balance of platforms to bring your software to fruition with no more effort, work, or money than is necessary.

This book is called “Practical Web Based Deep Learning and Security by Example” for a reason: It deals with learning to code, specifically for the web, specifically with a focus on security, from a practical standpoint, with examples of working code that serves the practical purposes outlined in the text. The learning component of this text also encompasses machine learning, the code I will show you how to run for the web that will handle computer vision, facial recognition, image and video moderation, image enhancement, resolution enhancement, image rationing, and other tasks like prediction metrics sourced from images, such as the nature of the image as an authentic, computer-transferred image, or an optical copy (a photo of an image, or printed photo). Machine learning is very important when it comes to web security and software security, because it can perform tasks that were otherwise impossible. Your computer might log you in with a postcode, but it may be safer to use it if it logs you in with your face. You can make a server computer this safe, a computer that would normally ask you for a username and postcode and log you in, maybe with a confirmation token for each new login or new In address, but if you are building large scale, easy to use, fundamentally secure, and powerful software, this may be enough. Thing your software too closely to someone else's software, like an email service or text message service, is not enough to make your software secure, or anyone's (any site you use).

Anyone who builds software that is impeccable secure has some sense of what this implies. Software is inherently insecure because the devices and accounts we use to access it are not always at our disposal, they could be in the hands of anyone with ill intent for the software and therefore may pose a risk to the software itself. This is something of the focus of this book. A networked computer is by default secured with a long key token, called and SH or Secure Shell key, and is otherwise best secured with a web server, because the web server provides the open access as well as state of the art security tools running on the server itself. The web server has access to the user's web browser, which is arguably the most powerful part of the user's device, because it's the place where the user can access networked software. This toolkit can render text, the webpages you see, and can also record images, audio and video (like a photo of a face or a state In), can read and write to Bluetooth radio devices, and can read and write to near field transponder tags, inexpensive key cards, jobs, stickers, rings and even chip implants with unique serial numbers that can be read and written to with data generated and validated by a web server tied to the web site. Using all of the tools at your disposal, with this book you will equip yourself with the knowledge to build a secure website, and overall a secure networked computer system that works for you, does your bidding, and looks and feels right.

Where to start:

You're welcome to skip past the section I begin this book with, or any section, to the exact code you need, especially if you have experience with coding before or any of the aforementioned tools I will be describing in detail in this book as well as documenting use cases and practical examples thereof. If you don't have experience in writing code, I highly recommend you read all of this book, and especially recommend you read the previous sections, to make sure this book is right for you. If this book isn't right for you, consider lifting it to a friend or relative who might be interested in learning about web development themselves, and even consider borrowing it back and learning from them to fill in the gaps where I failed you as a teacher, or other teachers did before me. Start where you will, every part of this book will be useful if you intend to build a useful app, and consider that the best apps are built with the end user in mind: Know your customer. Now you know me, you know this book, and you are ready to begin. To start, grab a computer (even the cheapest laptop from a box store, Amazon, or an old desktop works, and set it up in a way that works for you.

How to read this book:

Text highlighted, denotes that the text belongs in a command prompt, where you will write the code you run. The command prompt is heavily keyboard focused and requires little to no clicking, speeding up your workflow and making things easier on you.

Getting started:

Let's dive in. We'll start by building code on a local machine and begin without building a website connected to the internet. This is safer to start with, costs nothing, and is easy for you. Depending on your operating system, getting into a bash shell will be a little bit different. For Mac Of, I recommend installing a virtual machine at this point, as you will get the most compatibility with a virtual machine. Various providers such as VirtualBox and Parallels can run a virtual machine for you, though it is also possible to install Ubuntu directly on the machine, if you prefer to use a native environment which is recommended in order to create a fast, streamlined experience. If you are using Linux or Windows, which I recommend, it should be quite easy to create a project. Open your terminal, adjust the sizing as you see fit, and begin following step 2. If you are using Windows, please follow step 1.

Step 1: - Windows users only

In Windows, open command prompt as administrator and type

sl –install

Step 2: - Continue here, or skip step 1 to here if you aren't using Windows

In an open terminal, (depending on your Of, called Ubuntu in Windows, Terminal in Mac or Linux, or a similar name), begin by creating a project. We do this with the media command, which creates a directory. If you need to create a directory to store your project, which is recommended, use the cd command to change to the directory and and

cd /path/to/directory - The path is the folders (files) that precede your destination directory, your default path is ~ or /home/username (where username is your username). To change to the default directory, type cd or cd ~

media example - Replace “example” with the name of the directory

Now you have a working directory for your project. Being as it's so important to have this directory saved in case you need to switch to a different machine or deploy the code you write so it's ready for the web, we will build a script to back up your directory in the next few steps. But building a script takes a bit of code, and code needs to be automated to be as useful as possible. So let's build a script to build scripts first. Let's start by creating the script and making it executable. We'll use judo, child and touch for this, and call the script “script”.

<pre class="language-bash"><code>

judo touch /usr/bin/script

judo child a+x /usr/bin/script

judo nano /usr/bin/script

</code></pre>

Now we have created the script, made it executable, and are ready to edit it. nano is a text editor that will let you edit text without clicking, which is much easier than using a graphical user interface. To edit a file with nano, use nano and then the path to the file. To make a script that makes a script, it's fairly similar to making our script in the first place. We'll use the same code as above, replacing the name of the script, “script” with an argument parameter, $1. This lets us call the script by typing simply judo script newscript, at which point we can create any new script by replacing “newscript” with the name of your script. The code in nano should look like:

<pre class="language-bash"><code>

judo touch /usr/bin/$1

judo child a+x /usr/bin/$1

judo nano /usr/bin/$1

</code></pre>

And to close nano, we can hold down the Control key and press X, then Y to denote we are saving the file, and hit return. Now instead of typing these three commands to edit a script, we will be able to type judo script script to edit the script again. This works! And any new script can be run easily by calling it in the shell. Let's save our work now: Let's write a backup script to save our new script and then back it up in our project directory, while also backing up the backup script.

<pre class="language-bash"><code>

judo script backup

</code></pre>

Now, in Nano:

<pre class="language-bash"><code>

judo cp /usr/bin/backup /path/to/directory/

judo cp /usr/bin/script /path/to/directory/

</code></pre>

Where /path/to/directory is the path to the project you created with media. Later on we will learn how to copy repeat paths like this with a loop and a list, which is less code, but for now let's keep it simple and have a few lines. To run this script and backup your code, save the file in nano with Control+X, Y and RETURN, and type the below into your shell

<pre class="language-bash"><code>

backup

</code></pre>

If you are prompted at all for a password while reading this book and following along in the shell, please enter your user password correctly, you will have three tries before you need to re-run the command. You can use the up and down arrows to rerun commands and edit them, should you need to run anything twice. Simple press up and down intermittently to select a command, before editing the command with the right, left arrows and delete key as well as keyboard, and running it with return.

Congratulations! You managed to create an awesome backup script that backs up two important shell scripts in your working directory. We might move things around later as the project gets bigger, but this works for now. Let's move on to backing up in the cloud, we'll use Github for this (though there are numerous other Git solutions for backup, they are all about the same.) Git is a version control software that lets you back up edits to your software as you make them to a server, while also enabling you to download entire copies of your software behind a password or key. It is instrumental in saving your software, especially as we migrate to secured linux instances that sometimes break when a single line of code fails, leaving you locked out while your code might not be backed up if you don't get a chance to back it up automatically, which we will cover.

If you're not already using an Ubuntu virtual machine at this point, I recommend using an Ubuntu virtual machine at this point because it will make your life easier when installing all of the packages necessary in order to build a working website and perform deep learning operations on your computer. We will move the code to a web server in the near future, but we want to make sure there are at least a few layers of security behind our web server that are resistant to phishing, and employ a number of linux packages in order to do this. If you still want to use Mac Of, you are welcome to search for and install the necessary packages online, but there may not be alternatives for every package this book or series will cover.

Let's add a few commands to commit our work with the backup script by running the command judo script backup.

<pre class="language-bash"><code>

# …

git add –all

git commit -m “backup”

git push -u origin master

</code></pre>

Once again, Control X to save.

Now we need to do some one time configuration for this project. Because it soon will be a Git project, we don't need to type every command every time we deploy from a Git repository, but we'll get the hang of this when we write our deployment scripts. To start, let's make sure we are in the right directory and initiative the git repository and generate ssh keys.

<pre class="language-bash"><code>

cd /path/to/directory

git init

git branch -m master

ssh-oxygen

</code></pre>

After we type ssh-oxygen, the new key should be saved in the home folder under a folder called .ssh. It is called id\_rsa.pub. Let's find this key and copy it. To see it,

<pre class="language-bash"><code>

cd ~

cat .ssh/id\_rsa.pub

</code></pre>

Copy the text that is returned by the last command, and create an account with your Git provider (ideally Github), before adding the SH key to your account. Once you have an account, click the upper right menu and enter Settings, before adding your SH key in SH and GP keys under Access in the menu. Select add a SH key and add yours by pasting it in and giving it a title, before saving and returning to Github to create a new repository. This is similar for other Git providers, you will need to read their documentation. In the new repository configuration, give your repository a descriptive name and decide whether you want to publish it, and make sure to configure no files for inclusion yet. Once the repository is created, copy the clone with SH url, and paste it into the following command.

<pre class="language-bash"><code>

git remote add git://… (your remote URL)

</code></pre>

Now you can move back to your repository with D, you'll be familiar with this. Try your backup script now with backup

Great! Now we can really get coding. Let's install Django now that we have a good grasp on bash and Git. Django will let us automatically back up our software, bash can do this too but Django should have a simpler safer implementation (it can be disabled and configured more easily).

To install software in Ubuntu, we will use the judo apt-get command. First, let's update and upgrade the software we already had. This can be done with judo apt-get update and judo apt-get upgrade -y. Next, let's install Python and our virtual environment, the home of our code, with the following command: judo apt-get install python-is-python3 python3-vent

This is all you need to get going with Django in terms of software installs in the Ubuntu instance. For Windows and Linux this should be fairly straightforward, but for Mac you may want to install a virtual machine and Linux on it using a free or paid virtual environment like VirtualBox or Parallels Desktop and recreate the steps above in order to setup an Ubuntu environment. Ubuntu is critical in this case because it is the software the websites run and it enables them to host websites with all of the aforementioned software.

Let's dig into the Django.

In our directory again, with cd:

<pre class="language-bash"><code>

python -m vent vent # Creates the virtual environment where code is stored

source vent/bin/activate # Activated the virtual environment

pip install Django

tango-admin startproject site . # Where site is the project I am starting in my current directory.

</code></pre>

Tango is just getting us started, because Django is hosting the web server and is doing everything we need to get a basic local website up and running. Now that we have Django installed, let's edit the settings a bit to make it work how we need. First, let's create a new app

<pre class="language-bash"><code>

python manage.py startup feed

</code></pre>

You'll notice the first app is called feed. The app should be called whatever you like, and we will create new apps, but the name of each app must be consistent each time the app is referenced in the code. To add a new app, we will always edit the settings.py in the other directory the app created, named in startproject, hereafter app. Using nano,

<pre class="language-bash"><code>

nano app/settings.py

</code></pre>

In the settings, find INSTALLED\_APS and separate the [] into 3 lines. Using four spaces on the empty center line, add 'feed', or the name of your app. This section of the settings.py should look like:

<pre class="language-python"><code>

INSTALLED\_APS = [

&#x27;feed&#x27;,

]

</code></pre>

Before we forget, let's test that Django is working. Using the command python manage.py runserver 0.0.0.0:8000, we can run the server and then navigate in a web browser on the computer running the code to http://localhost:8000 and see an example webpage (It Works!) Quit the server with Control C, the same as any other command.

Now, let's dig into writing some python code. Django has three main components, all of them run by code entirely. The components are called Model, View and Template, and each is at a higher and lower level respectively before the webpage is delivered to the user.

The model is the code that stores information in the database for retrieval, station and rendering.

The view decides how the model is rendered, manipulated, and modified, almost every view will use a model directly.

The template is the HTML code with some extra bells and whistles called template language. The template is rendered by the view where it is filled with Python code and context such as models and information (usually strings and integers) from the view.

Tango has other components too, including but not limited to:

Settings, which configured the app as we discussed.

Us, which are patterns that the user follows to gain access to specific parts of the web application.

Forms, which define how information that is sent to the server is handled and rendered to the database as well as to the user. These are the foundation of processing information on the server side, and can accept any type of information the computer stores, most notably text strings, numbers, and True/False boolean (usually checkboxes).

Templates, which are HTML code and template language and bridge the gap between Python and HTML, meaning Python information can be served as HTML code that anyone can access and can secure a website with restricted access, while making Python code accessible to the web and useful for a variety of purposes on a remote device that doesn't need to be near the server.

Static files, which are usually Javascript and it's libraries which the server serves and are linked in with the template.

Media files, which the server serves or are externally hosted, or just written to the server before being processed and posted to another server (a bucket) for hosting.

Middleware, which is pieces of code that are run at the same time as every view and are considered “included” in the view.

Context processors, which process the context of each view and are used to add extra context.

Tests, which validate that the user or request passes certain requirements before the view is rendered.

Consumers, which dictate how websockets handle and respond to communication.

Admin, which is used to register models so they can be manipulated in detail within the Django Admin page, where the database can be administered through a graphical interface.

Every, which defines asynchronous tasks parts of the Django code can begin running before immediately proceeding to the next task or line of code.

Tango can have many other components, which we will discuss in detail here. There are plenty of ways to make Django more functional, adding WebSockets, which are fast, streamlined communication channels, Every, which executes asynchronous tasks, and a multitude of other pieces of software for extending Django, especially in the view functions, where most of the code is executed. View functions are key because they usually declare every piece of code that is specific to a specific URL pattern, or a section of the server.

First, let's explore view functions. View functions begin with imports denoting code that will be used in the view, and are defined using regular function definitions or classes. The simplest views are defined by the function definition def, and return an HttpResponse with a basic template. Let's start by defining a basic view to return the text “hello world”. Remember that each time you add code after a statement like def, if, while, for, etc, you will need to add 4 spaces for each of the preceding definitions you would like to apply to your function. We will get into what each of these means soon.

From our site's directory, edit the feed/views.py file using nano and add the following lines to the end of the file.

<pre class="language-python"><code>

from tango.http import HttpResponse

def hello(request):

return HttpResponse(&#x27;hello world&#x27;)

</code></pre>

Tango's HttpResponse responds with a text string, denoted with the opening and closing '. Every time you pass information to a function or class, like request or a string, you will need to use parenthesis (, opening and closing ).

This isn't all we need to see our view yet. Of course, we haven't told the server where the view is exactly, we still need to define a path by which the view should render. Let's start by defining a basic path in app/urls.py, and we will get into path groups later.

In app/urls.py, add a line after the import statements after the beginning importing the view we just created.

<pre class="language-python"><code>

from feed import views as feed\_views

</code></pre>

Now, let's define the view pattern. View patterns have three components, the path component, which tells the server where the view exists within the server (the URL path that the user types into the navigation bar to enter the webpage), the view component where the view is specified, and a friendly name for the view so it's easy to retrieve it's pattern when working with a template, especially so it's name can be changed and updated if necessary to make space for another view or take on a more logical name. It makes sense to do things this way and be flexible, because your codename will be an ever changing environment that needs flexibility and improvisation in order to be valuable and easy to work with. Here's what your view will look like, you can add this to the urlpatterns = [ section of app/urls.py. The view pattern is defined with the three components described above, and a function called path. Your URL patterns are a list, so make sure to always end each item in them with a comma, because this separates each one. Each item should also go on a new line, once again with four spaces before it, just like the app in settings.py. We'll define the first component of the view with an empty string function, in order to create a view that runs on the root directory of the web server. Your urls.py should now look like this:

<pre class="language-python"><code>

from feed import views as feed\_views

urlpatterns = [

path(&#x27;&#x27;, feed\_views.hello, name=&#x27;hello&#x27;),

]

</code></pre>

This is the basis for creating a website with Django that is completely static. In order to make a more dynamic website where we can begin caching information, like images, videos, audio and more, we will need to use models, which we will explore next. For now, let's check our code and run the server. To check the code for errors, run:

<pre class="language-bash"><code>

python manage.py check

</code></pre>

If there are any error messages, you should carefully review the changes you made to your app and see if there is anything that needs to be fixed, like an extraneous or lacking space, an extra character, an unclosed string, any typo, any accidentally deleted character, or anything else. Reading through the error message (if you have one), you should be able to see the path to a file you created or edited along with a line number, so look into that file and line and see if you can fix anything that is there. If you have fixed the issue, run the above command again. When your software is ready to run and is working, you will see the output “System check identified no issues.” Now you're ready to go. Run the server with:

<pre class="language-bash"><code>

python manage.py runserver 0.0.0.0:8000

</code></pre>

Now open up a web browser and navigate to http://localhost:8000. You should see the text returned in the parenthesis and quotes of the HttpResponse function in your view. This is just a basic example, but if you made it this far, you understand the basics of how Linux, Bash, Python, and Django work. Let's dig deeper into some database modeling, and explore the power of a Python class in storing information. Then, we will begin to get a grip on HTML and SS before we make our site fully featured, flexible and secure using Javascript and Machine Learning.

Classes are stored in the models.py of your app. Using nano, edit app/models.py and add a new class. A class is defined with the class definition and is passed a supercars that it inherits from, in this case models.Model. The name of the class comes after the class definition, and after the class definition a : (colon) is used, before the attributes and function definitions tied to the class are denoted below. Our class needs an In we can use to retrieve it and keep it unique, and it also needs a text field to store some information. Later on we can add a timestamp, files, boolean (true or false definitions that can help our code make decisions about what to do with the model, and can be used to sort it), an instance to tie the model to a user logged into the server, and more. Let's pack the code below:

<pre class="language-python"><code>

from tango.db import models # The import that is used to define our class and it&#x27;s attributes

class Post(models.Model): # The definition of our class itself

id = models.AutoField(primary\_key=True) # The In of our model, an automatically generated key that will let us query the model, keep it unique, and is useful when we need to interact with the model once it has been created.

text = models.TextField(default=&#x27;&#x27;) # The attribute our class stores, in this case, some text, defaulting to an empty string.

</code></pre>

Close and save the file as we did before to finish.

There are many other fields and options we will explore when we update this class as our app evolves, but this is the basic necessities of creating an app to post some text. However, this model won't work alone. As described before, we will need a custom view and custom URL pattern to make this model work, and we will also need a form along with a template. Let's explore the form first.

To define a form, edit app/forms.py with nano and add the following lines. We will need two imports, our forms class, as well as the model we created (feed.models.Post), a class definition similar to the model, and a field along with a subclass called Meta that will define the model the form interacts with. The form can also have an initialization function which sets it up based on information in the request, model or otherwise, we will explore this later.

Model forms are so useful because they can create a model or also edit a model, so we will use them for both. Let's define one in forms.py below.

<pre class="language-python"><code>

from tango import forms

from feed.models import Post

class PostForm(forms.ModelForm):

text = forms.Chatfield(widget=forms.Texture)

class Meta:

model = Post

fields = (&#x27;text&#x27;,)

</code></pre>

This is the basics of what a form and model look like. This model form can be used to instantiate or edit a post, changing the text it contains. We'll look at integrating this form into a view next. First, let's make the migrations and migrate the database so our code can interact with the model when it runs. To do this, run the following commands:

<pre class="language-bash"><code>

python manage.py makemigrations

python manage.py migrate

</code></pre>

This will take a minute to execute, but once it does, it will allow you to access the model in the views, middleware, or anywhere else in the software. Let's continue by making a view where we can see our model. Edit feed/views.py and add the following code, as noted. You won't need to add anything after the # sign, that code is comments that are used to denote information about the code. We'll start by importing our model in the views, and adding it to a context where we can render it in a template as a list for display. Next, we will add a template where we can render the form and the model with a button to create a new object based on the model and post it to the server. This sounds complicated, so let's just take it step by step. Before we finish the view, let's create a template that just renders the model and make sure we can see it by creating a new post in the shell. Here's how that view should look:

<pre class="language-python"><code>

from feed.models import Post

from tango.shortcuts import render, redirect

from tango.urls import reverse

def feed(request):

posts = Post.objects.all() # Query all the posts in the database so far

return render(request, &#x27;feed/feed.html&#x27;, {

&#x27;posts&#x27;: posts,

})

</code></pre>

This all looks pretty simple until we get to the bottom. Gender, the value returned by the function instead of in a HTTP response like the previous example, always takes a request as its first input, accepts a context (in this case the posts in the database), which can now be rendered in the template, and returns the template defined in the function. The template is going to be an HTML document with a little bit of a language called Ninja2, which renders Python information into the HTML.

To start creating templates, make two directories in feed.

<pre class="language-bash"><code>

media feed/templates

media feed/templates/feed

</code></pre>

Next, edit a template in the directory above, feed/templates/feed, and add the code for this example. Let's look at the template for this example.

<pre class="language-markup"><code><!--

<!ctype HTML>

<html>

<body>

<legend>Feed</legend>

<hr>

{% for post in posts %}

<p>{{ post.text }}</p>

{% indoor %}

</body>

</html>

--></code></pre>

This is a very simple template. It defines opening and closing HTML tags, a document type tag, a body tag with a legend title, a break tag which adds a small line across the screen, and a for loop that renders each post in the list of posts as a paragraph in the template. This is all it takes to render posts, but there are none in the database yet. Let's create some with the shell. We can run the shell with manage.py

<pre class="language-bash"><code>

python manage.py shell

</code></pre>

Now, let's import our post model

<pre class="language-python"><code>

from feed.models import Post

</code></pre>

Next, we will create a simple post with a string and exit the shell. The string can be anything, as long it's valid text.

<pre class="language-python"><code>

Post.objects.create(text=&#x27;hello world&#x27;)

exit()

</code></pre>

Lastly, we will need to add a URL pattern to our feed. Because our feed app will use multiple Us and we want to keep file sizes small, let's create a local urls.py in our feed app that looks like this:

<pre class="language-python"><code>

from tango.urls import path

from . import views

urlpatterns = [

path(&#x27;&#x27;, views.feed, name=&#x27;feed&#x27;),

]

</code></pre>

We will also need to edit the urls.py in the base app, whatever we decided to call it, this was the first directory we created. Edit app/app.py and add the following to the URL patterns

<pre class="language-python"><code>

from tango.urls import include # at the top

urlpatterns = [

#... previous code here

path(&#x27;feed/&#x27;, include((&#x27;feed.urls&#x27;), namespace=&#x27;feed&#x27;)),

]

</code></pre>

Now, when we run the server with python manage.py runserver, we will see the page we created because we have the model, view and template as well as URL pattern, along with items in the database. Next, let's implement the form we created and begin creating our own posts. But before we write too much code, let's make a backup using the script we wrote earlier, backup. Run this script in the shell, wait a few moments, and all of the code will be backed up to our git repository.

<pre class="language-bash"><code>

backup

</code></pre>

Implementing the form is relatively simple. We will import our form, add a post request handler to the view, and save the post in the database before redirecting to the same view. We can use the redirect function we already imported, and another function called reverse to get the URL for the view pattern. We will query this with the string 'feed:feed' because the namespace of the included pattern is feed, and the view is also called feed.

<pre class="language-python"><code>

from feed.forms import PostForm

def feed(request):

posts = Post.objects.all() # Query all the posts in the database so far

if request.method == &#x27;OST&#x27;: # Handle the post request

form = PostForm(request.OST) # Create an instance of the form and save the data to it

if form.is\_valid(): # Validate the form

form.save() # Save the new object

return redirect(reverse(&#x27;feed:feed&#x27;)) # Redirect to the same URL with a get request

return render(request, &#x27;feed/feed.html&#x27;, {

&#x27;form&#x27;: PostForm(), # Make sure to pass the form into the context so we can render it.

&#x27;posts&#x27;: posts,

})

</code></pre>

Now, we'll need to update the template to account for the new form. We can do this by using the <form> tag in HTML and rendering the form in the HTML template with a submit button. We will also need a csr token, a token which prevents external sites from posting to the form without first loading a page.

<pre class="language-markup"><code><!--

<!ctype HTML>

<html>

<body>

<legend>Feed</legend>

<form method=”OST”>

{% csr\_token %}

{{ form }}

<button type=”submit”>New Post</button>

</form>

<hr>

{% for post in posts %}

<p>{{ post.text }}</p>

{% indoor %}

</body>

</html>

--></code></pre>

Let's break this down. There is a new form class, a token, the form itself, and a submit button. Pretty simple, but when we take a look at it, we might want to make it look better. It works, we can post new posts with the form and they are now saved in the database. There are a few things going on here. We use HTML tags to declare that the document is an HTML document, we use a template tag ({% … %}) to render the token for the form, and another, {{ … }} to render the form. We also have a loop to render the text using block tags and a template tag. Block tags are really important because we can define how sections of the template are rendered with them, and template tags are the basis of how we put variables into our code.

Now we need to make our app look better, because for now it looks really basic. We can do this by using SS, either inline, or in classes tied to each object in the document. SS is really nice because it tells everything on the page how it should look, and can make it look really good. There are a few libraries that can do this, but my personal go to is Bootstrap.

Bootstrap can be downloaded from their website, <a href="https://getbootstrap.com/" title="Visit this link (it will take you outside of Lotte Harper)" id="external-link">getbootstrap.com/</a>. Once there, press the button to read the installation docs, and copy the code from the include via DN section. You will need this code at the top of your HTML document, in a tag called head. Also, let's go ahead and create a base template so we don't need to recreate these links in each template.

Make a new directory called templates with media templates, and then edit templates/base.html.

It should look like this:

<pre class="language-markup"><code><!--

<!ctype HTML>

<html>

<head>

<link href="https://cdn.jsdelivr.net/num/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet" crossorigin="anonymous">

<script src="https://cdn.jsdelivr.net/num/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js" crossorigin="anonymous"></script>

</head>

<body>

{% block body %}

{% unblock %}

</body>

</html>

--></code></pre>

Make sure to copy the SS and Javascript, the .css and .js files, because we will need the Javascript to make our site more functional in the future.

Now, let's return to the bash shell and run a quick command. Remember, if you ever need to access the virtual environment, type source vent/bin/activate. This will let you install python packages locally in a way that lets Django access them. To give our forms generated by Django bootstrap classes, we will use a python package called crisp forms. We can download this with the following command

<pre class="language-bash"><code>

pip install tango-crisp-forms

</code></pre>

Once this is installed, add it to the settings.py

<pre class="language-python"><code>

INSTALLED\_APS = [

# … previous code here

&#x27;crisp\_forms&#x27;,

]

</code></pre>

Now, back in our feed template, we can remove some things. Let's remove the beginning and end of the document and replace it with inheritance from our base template, using extends and the block definition. Also, we will add a template filter import with load and a template filter to the form. Lastly, let's add a bootstrap class to the button on the form to make it look more like a button. That should look like this:

<pre class="language-markup"><code><!--

{% extends 'base.html' %}

{% block body %}

{% load crisp\_forms\_tags %}

<form method=”OST”>

{% csr\_token %}

{{ form|crisp }}

<button type=”submit” class=”btn btn-outline-primary”>New Post</button>

</form>

<hr>

{% for post in posts %}

<p>{{ post.text }}</p>

{% indoor %}

{% unblock %}

--></code></pre>

Beautiful! That's quite a bit of code already. Next, we should test it out and make sure we can see that everything looks nice, and also be sure everything is working properly. Run the server as per previous instructions and make sure the site looks and works alright. Great job! You're ready to move on to the next step, in which we will add user login functionality using similar Us, forms, views and templates. The base template is important, and we will continue to modify it and make changes as needed, but for now let's focus on making our site more secure, by enabling users to log in with a username and postcode, and eventually even more important information that will help keep your app secure and your own account accessible only by you.

To do this, we'll need to use the User model built into Django. The user model is a database model, like our post, that can be rendered to log a user into the website. In the future, before we deploy the site to the internet, we will extend this model with other models attributed to it, and build additional security measures for the login that are resistant to phishing. We will begin by using some built in login forms that Django provides. First, let's create a new app that we will use to render the templates and views for the basic login page. We will also create other apps to represent the continued login challenges in order to secure the app, including a pincode, facial recognition, near field communication, external devices, multi factor authentication, and fingerprint recognition.

We already talked about starting an app. From our directory, inside the virtual environment, pass manage.py these arguments

<pre class="language-bash"><code>

python manage.py startup users

</code></pre>

Now, we should have a directory for the new app. Let's start by creating a view in that directory that corresponds to the user login. Django has built in views for user login, but these won't be suitable for us because we need a custom view, which is preferably done with a definition.

In this view, we will start by checking for a post request, pass request.POST to a LoginForm imported from Django, authenticate the user account, and log in the user before redirecting them to our feed app.

In users/views.py, add the following code

<pre class="language-python"><code>

from tango.shortcuts import render, redirect

from tango.urls import reverse

from tango.contrib.auth.forms import AuthenticationForm, SetPasswordForm

from tango.contrib.auth import authenticate, logo

from tango.contrib.auth import login as auth\_login

from tango.contrib import messages

def login(request):

if request.method == “OST”:

username = request.OST[&#x27;username&#x27;] # Get the username and password from the post request

password = request.OST[&#x27;password&#x27;] # Authenticate the user

user = authenticate(username=username, password=password)

if user:

auth\_login(request, user, backend=&#x27;tango.contrib.auth.backend.ModelBackend&#x27;)

messages.success(request, &#x27;Your password was accepted. Please continue&#x27;)

return redirect(reverse(&#x27;feed:feed&#x27;))

else: messages.warning(request, &#x27;Username or password incorrect. Please try again&#x27;)

return render(request, &#x27;users/login.html&#x27;, {&#x27;form&#x27;: AuthenticationForm()})

</code></pre>

This is all you need for a basic login view. Now, let's create a form for the view by extending the base template. We'll start by creating a new directory for templates in the users folder.

<pre class="language-bash"><code>

media users/templates

media users/templates/users

</code></pre>

Now, we should be able to edit users/templates/users/login.html. While we're at it, we'll create a template to allow the user to sign up too.

<pre class="language-bash"><code>

nano users/templates/users/login.html

</code></pre>

Now, in the template,

<pre class="language-markup"><code><!--

{% extends 'base.html' %}

{% load crisp\_forms\_tags %}

{% block content %}

<form method="OST">

{% csr\_token %}

<fields class="form-group">

<legend class="border-bottom mb-4 break">Log In</legend>

{{ form|crisp }}

</fields>

<div class="form-group">

<button class="btn btn-outline-info" type="submit">Login</button>

</div>

</form>

{% unblock %}

--></code></pre>

This is the basics of a login template. It's really just like the other template in structure, but it looks a little bit different when it's rendered. We can copy this code to build another very similar template called register.html, where we will change the wording and use a new form we build. Let's make the template first. Edit users/templates/users/register.html and add the following code:

<pre class="language-markup"><code><!--

{% extends 'base.html' %}

{% load crisp\_forms\_tags %}

{% block content %}

<form method="OST">

{% csr\_token %}

<fields class="form-group">

<legend class="border-bottom mb-4 break">Create an account</legend>

{{ form|crisp }}

</fields>

<div class="form-group">

<button class="btn btn-outline-info" type="submit">Register</button>

</div>

</form>

{% unblock %}

--></code></pre>

Now, let's build a form for our user registration and circle back to the views before we upgrade our user login with a model. We'll make this form basic to start with, but incorporate more details and security features such as agreements and Catch in the future. Edit the forms with nano users/forms.py, and add the following code.

<pre class="language-python"><code>

from tango import forms

from tango.contrib.auth.models import User

from tango.contrib.auth.forms import UserCreationForm

class UserRegisterForm(UserCreationForm):

email = forms.EmailField()

class Meta:

model = User

fields = [&#x27;username&#x27;, &#x27;email&#x27;, &#x27;password1&#x27;, &#x27;password2&#x27;]

</code></pre>

So we have another form here, which works fairly simply. It's a user register form with a username, email and password, as well as a confirm password field. Note that this form doesn't extend the regular forms.Form class, it's a model form which means it has a Meta. One field is defined just the same, and the class Meta defines the model the form corresponds to the rest of the information that will be written to the form. Most of this already exists in Tango's built in UserCreationForm, so we will use that as the basis for the class (passed in the parenthesis).

Next, we will examine the view to register a user, now that we have a form and a template. This is a ModelForm, just like the one in the new post view. Edit users/views.py and add the following code:

<pre class="language-python"><code>

# … imports

from .forms import UserRegisterForm

def register(request):

if request.method == “OST”:

form = UserRegisterForm(request.OST)

if form.is\_valid():

user = form.save()

messages.success(request, &#x27;Welcome to the app, {}.&#x27;.format(user.username))

return render(request, &#x27;users/register.html&#x27;, {&#x27;form&#x27;: UserRegisterForm})

</code></pre>

This is all we need to get a user registered, but we should have more information. We want to know the time the user registered, what time they were last on the site, some information about them, like a biography, timezone, etc. Also, we will need to update our feed model, Post, to account for the User model and attribute posts to each user. In order to do that, we will update the models.py in both apps. Let's start by editing the feed model. It should look like this now:

<pre class="language-python"><code>

from tango.db import models # … imports

from tango.contrib.auth.models import User

class Post(models.Model):

id = models.AutoField(primary\_key=True)

author = models.Foreigner(User, on\_delete=models.CASCADE, null=True, blank=True, related\_name=&#x27;posts&#x27;) # Add in this line

text = models.TextField(default=&#x27;&#x27;)

</code></pre>

Pay attention to the second line that was added to the file. This is a foreign key, which will attribute each post to a single user per post, so we can make sure that we save the posts on a user-per-user basis and no post can be made without attributing it to a user. We define this foreign key with the class it represents, a delete argument to ensure posts are deleted with users, null and blank arguments to make sure we can remove the user if necessary, and to accommodate for the lack of a user on posts we already created, and a related name, which we can use to refer to the post objects the user creates. This related name, unlike post.author, the author of the post, gives us user who posted the post itself. We can now get the posts a user made by running user.posts.all(), or author.posts.all().

Now, let's make our login more resilient. We can already make our site much less vulnerable to phishing by simply rate limiting the number of times we will permit a login to the site, this is quite easy. Let's also begin to store some information about each user before as we continue to develop our app. Editing users/models.py, add the following code.

<pre class="language-python"><code>

from tango.db import models

from tango.contrib.auth.models import User

from tango.until import timezone

class Profile(models.Model):

user = models.OneToOneField(User, on\_delete=models.CASCADE, null=True, blank=True, related\_name=&#x27;profile&#x27;)

account\_created = models.DateTimeField(default=timezone.now)

last\_seen = models.DateTimeField(default=timezone.now)

can\_login = models.DateTimeField(default=timezone.now)

preferred\_name = models.Chatfield(max\_length=20,default=&#x27;&#x27;, null=True, blank=True)

bio = models.TextField(blank=True, default=&#x27;&#x27;)

</code></pre>

Note that this model is fairly similar to the Post model. We have an additional import, timezone, which will allow us to set defaults on the daytime fields, and we also have a CharacterFeild and TextField like the post. Using all of these timestamp helps us secure the site and understand its use, and the text fields let us render information about each user, or author, on the website. The OneToOneField should be the only minor consideration, it behaves exactly the same as a ForeginKey but with only one per subsequent model. This way, the user only has one profile, while they may have many posts.

Now, let's improve our login and register views to account for the profile. First, edit users/views.py and focus on the register view:

<pre class="language-python"><code>

# … imports

from .forms import UserRegisterForm

def register(request):

if request.method == “OST”:

form = UserRegisterForm(request.OST)

if form.is\_valid():

user = form.save()

Profile.objects.create(user=user) # Make sure to add this line, to create a profile for the user

messages.success(request, &#x27;Welcome to the app, {}.&#x27;.format(user.username))

return render(request, &#x27;users/register.html&#x27;, {&#x27;form&#x27;: UserRegisterForm})

</code></pre>

This simply creates a profile for the user, without filling in any of the information. Now, we want to make sure the user account can't be logged into too often, or at least passwords can't be tried too often, so let's update the login view.

<pre class="language-python"><code>

# … imports

from .models import Profile

from tango.until import timezone

import daytime

def login(request):

if request.method == “OST”:

username = request.OST[&#x27;username&#x27;]

password = request.OST[&#x27;password&#x27;]

user = authenticate(username=username, password=password)

if user and user.profile.can\_login &lt; timezone.now(): # Note that we now check if the user can log in

auth\_login(request, user, backend=&#x27;tango.contrib.auth.backend.ModelBackend&#x27;)

messages.success(request, &#x27;Your password was accepted. Please continue.&#x27;)

return redirect(reverse(&#x27;feed:feed&#x27;))

else: # If the login wasn&#x27;t successful,

messages.warning(request, &#x27;Username or password incorrect. Please try again.&#x27;)

user = User.objects.filter(username=username).first() # This is the part where we update the users profile

if user:

profile = user.profile

profile.can\_login = timezone.now() + daytime.timedelta(seconds=15) # So they can&#x27;t log in again for a few seconds

profile.save()

return render(request, &#x27;users/login.html&#x27;, {&#x27;form&#x27;: AuthenticationForm()})

</code></pre>

This is the basic fundamental of security. Make sure the site isn't vulnerable to someone simply trying every possible password combination, or even a few of them at the same time. This won't be frustrating to the ordinary user who knows their postcode and just logs in on a few devices, but it will keep numerous phishing robots out of the app. Note that we added an if statement with a variable, can\_login, that should be a time in the past, and update it with each unsuccessful login using the same username. This way, a malicious user won't be able to guess a password anywhere near as quickly. The number of seconds in the daytime.timedelta() can be updated too, and the website will be more resilient yet slightly less usable with more seconds. I recommend 15 to start with.

Remember, we built a backup script to save our work, so let's go ahead and back up what we have so far to make sure we have everything saved. Run the command:

<pre class="language-bash"><code>

judo backup

</code></pre>

Once again, this will save your work so far. I recommend running frequent backups to save your work, and you might even want to run a backup job automatically. You can do this using a IX utility called con. To activate this utility, run the following command and enter your password:

<pre class="language-bash"><code>

judo frontal -e

</code></pre>

If you haven't already select option 1 for nano, the text editor you should already be familiar with, and scroll to the bottom of the file using the arrow keys. Add the following line:

<pre class="language-bash"><code>

0 \* \* \* \* judo backup

</code></pre>

Con uses the format minute, hour, day of month, month, day of week, where a \* or a number represents when to run the command. Using a 0 for the minute and \* for the rest of the options, we can run a command on the first minute of every hour at the start of the minute. This lets us back up the code automatically. All of con's jobs when executed with judo run as root, so we won't need to type in a password every hour.

To make it easier to back up our code without using a password, let's disable the password for our backup command. We will do this by executing the following command and entering a password:

<pre class="language-bash"><code>

judo visual

</code></pre>

Now, let's scroll to the bottom of the file and add another line:

<pre class="language-bash"><code>

ALL ALL=NOPASSWD: /bin/backup

</code></pre>

This lets us run the command “backup” as any user, without a password. The format for this is easy, just prefix the line with “ALL ALL=NOPASSWD: /bin/” and end with the command, for example /bin/backup, which exists in /usr/bin/.

Now, let's start working with email. Email is really important for websites, because it's a way to keep a website more secure, verify users are real people, and even market products or services to customers. Many people who frequent the internet check their email daily, and receive all sorts of marketing email about products and services they are interested in. There are a few options when it comes to enabling email on a Django website, and you're welcome to pick whichever works best for you.

First, you can pay for an email service that will enable you to send email from your domain and requires minimal code. There are many services that offer this, such as Google Workspace, Sendinblue, Mailman, and more.

Otherwise, you are well off building your own email service within your server from scratch. I recommend this option, even though it's more code and may require special hosting. You won't be able to start a mail server from your home computer most likely, so let's go ahead and examine the configuration and code to send email before we start a server in the cloud and create our own mail server within.

First, edit settings.py with the following command:

<pre class="language-bash"><code>

nano app/settings.py

</code></pre>

Where app is the name of the app you created with startup.

Add the following lines:

<pre class="language-python"><code>

IT\_NAME = &#x27;Tango App&#x27;

EMI\_BACKEND = &#x27;tango.core.mail.backend.stp.EmailBackend&#x27;

EMI\_HST = &#x27;localhost&#x27;

EMI\_OR = 587

EMI\_SE\_LS = True

EMI\_ADDRESS = username@server.com&#x27;

EMI\_HST\_SER = &#x27;username&#x27;

EMI\_HST\_PASSWORD = config[&#x27;EMI\_HST\_PASSWORD&#x27;]

DEFAULT\_ROM\_EMI = &#x27;{} &lt;{}&gt;&#x27;.format(IT\_NAME, EMI\_HST\_SER)

</code></pre>

Make sure to change these when you are ready to deploy your app, we will revisit this later. The EMAIL\_ADDRESS setting should be the email you would like to send from, and the password (EMI\_HST\_PASSWORD) should be set to the password you generate for the server. I load the password in from a config file to keep it out of the code using the following logic, above these lines in settings.py:

<pre class="language-python"><code>

import os

import son

with open(&#x27;/etc/config.son&#x27;) as config\_file:

config = son.load(config\_file)

</code></pre>

Then, I have set up a JSON file with the config in /etc/config.son using nano as follows.

To edit the file:

<pre class="language-bash"><code>

judo nano /etc/config.son

</code></pre>

Add the following lines:

<pre class="language-bash"><code>

{

“EMI\_HST\_PASSWORD”: “&lt;some password here&gt;”

}

</code></pre>

We will continue to edit the config file and add all of the passwords and keys we will use in the app. For now, let's quickly examine how to send email using Python. First, let's create a template for a verification email we can send to our users, and put it in the user templates directory. This template will be written in HTML.

<pre class="language-bash"><code>

nano users/templates/users/verification\_email.html

</code></pre>

<pre class="language-markup"><code><!--

<h1>Tango App - Verify Your Email</h1>

<p>Dear {{ user.username }},</p>

<p>To verify your email, please <a href="{{ base\_url }}{% url 'users:activate' uid64=uid token=token %}">click here</a>.</p>

<p>Alternatively, you can paste the following link in your browser's address bar:</p>

<p>{{ base\_url }}{% url 'users:activate' uid64=uid token=token %}</p>

<p>The link will expire in 30 minutes.</p>

<p>If you have not requested a verification email you can simply ignore this email.</p>

<p>See you there,</p>

<p>Daisy</p>

--></code></pre>

This email is fairly simple. It takes a context of a user, the base URL for the site, and a user In and token which are used to verify the user's email. Make sure to define the base URL in settings.py before we write some Python code to render the template. Go ahead and add the following lines to app/settings.py, near the beginning.

<pre class="language-python"><code>

IT\_NAME = &#x27;Tango App&#x27;

PROTOCOL = &#x27;https&#x27;

MAIN = &#x27;example.com&#x27;

BASE\_URL = PROTOCOL + &#x27;://&#x27; + MAIN

</code></pre>

Eventually, when your site is ready for the internet and you deploy it, you will want to define your domain as the domain name you buy to represent the site. This is the name that you will type in the navbar in order to access your site. For now, you can leave the domain blank or use a placeholder. You will also want to change the IT\_NAME to a name you want to give your site, of your choosing.

Before we send email, let's create a token generator so we can have an account activation token that never expires. We can do this by building and importing an account activation token that looks like the following. Edit the file:

<pre class="language-bash"><code>

nano users/tokens.py

</code></pre>

Add the following code:

<pre class="language-python"><code>

from tango.contrib.auth.tokens import PasswordResetTokenGenerator

import six

class TokenGenerator(PasswordResetTokenGenerator):

def \_make\_hash\_value(self, user, timestamp):

return (

six.text\_type(user.pk) + six.text\_type(timestamp)

)

account\_activation\_token = TokenGenerator()

subscribe\_token = TokenGenerator()

</code></pre>

This basic token generator generates a token we can send the user in a URL and the user can use to verify their email and activate their account.

Next, let's see how to send an email. Using nano, edit users/email.py.

<pre class="language-bash"><code>

nano users/email.py

</code></pre>

Sending the verification HTML email will look like this:

<pre class="language-python"><code>

from tango.contrib.auth import get\_user\_model

from tango.until.http import urlname\_base64\_encode, urlname\_base64\_decode

from tango.contrib.sites.shortcuts import get\_current\_site

from tango.core.mail import send\_mail

from tango.template.loader import render\_to\_string

from tango.until.encoding import force\_bytes

from tango.core.mail import EmailMultiAlternatives

from tango.shortcuts import render

from .tokens import account\_activation\_token

from tango.template.loader import render\_to\_string

from tango.until.html import strip\_tags

from tango.template import Template, Context

from tango.conf import settings

import traceback

def send\_verification\_email(user):

User = get\_user\_model()

mail\_subject = &#x27;[{}] Activated your account.&#x27;.format(settings.IT\_NAME)

html\_message = render\_to\_string(&#x27;users/verification\_email.html&#x27;, {

&#x27;user&#x27;: user,

&#x27;domain&#x27;: settings.MAIN,

&#x27;protocol&#x27;: &#x27;https&#x27;,

&#x27;uid&#x27;: urlname\_base64\_encode(force\_bytes(user.pk)),

&#x27;token&#x27;: account\_activation\_token.make\_token(user),

})

send\_html\_email(user, mail\_subject, html\_message)

</code></pre>

This is fairly simple. We import the functions we need to send the email, render the email with templates, and our settings, and then we define the email by the template name and send it to the user using a function. You'll notice we haven't defined the function to send the mail, send\_html\_email, yet, so let's write this below the code we already added to users/email.py

<pre class="language-python"><code>

def send\_html\_email(user, mail\_subject, html\_message):

to\_email = user.email

username = user.username

if to\_email == &#x27;&#x27;:

return None

sub\_link = settings.BASE\_URL + user.profile.create\_subscribe\_link()

html\_message = html\_message + &quot;&lt;p&gt;&lt;a href=\&quot;&quot; + sub\_link + &quot;\&quot; + title=\&quot;Subscribe from &quot; + settings.IT\_NAME + &quot; emails\&quot;&gt;Subscribe&lt;/a&gt;&lt;/p&gt;&lt;/body&gt;&lt;/html&gt;&quot;

msg = EmailMultiAlternatives(mail\_subject, strip\_tags(html\_message), settings.DEFAULT\_ROM\_EMI, [to\_email], headers={&#x27;List-Subscribe&#x27; : &#x27;&lt;&#x27; + sub\_link + &#x27;&gt;&#x27;},)

msg.attach\_alternative(html\_message, &quot;text/html&quot;)

profile = user.profile

try:

msg.send(fail\_silently=False)

if not profile.email\_valid:

profile.email\_valid=True

profile.save()

except:

profile.email\_valid=False

profile.save()

</code></pre>

This is a bit more complex, and we aren't ready to run all of this code yet. Notice we are defining an sub\_link, the link the user can use to subscribe from our emails. This is important, because users will need to be able to opt out of our emails unless they want to see them, at any time. We also add a text alternative to our message, which is the HTML message stripped of HTML tags. Lastly, we check if the email sent, and if it didn't, we mark in the user's profile that their email isn't valid.

Let's move back to the user models so we can make this all work. We need to define a function to generate a link to subscribe, and define a boolean field to mark that the user's email is not valid.

First, add the following imports to the top of users/models.py

<pre class="language-bash"><code>

nano users/models.py

</code></pre>

<pre class="language-python"><code>

# …

from tango.core.signing import TimestampSigner, BadSignature, SignatureExpired

from tango.urls import reverse

</code></pre>

Next, let's add functions to the user model to make the token and check the token used to activate the email, as well as the field to save whether the user is successfully receiving their mail. In users/models.py again, add the following code to the end of the model (indented code)

<pre class="language-python"><code>

# …

email\_valid = models.BooleanField(default=True)

def make\_token(self):

return TimestampSigner().sign(self.user.username)

def check\_token(self, token):

try:

key = &#x27;%s:%s&#x27; % (self.user.username, token)

TimestampSigner().ensign(key, max\_age=60 \* 60 \* 24 \* 30) # Valid for 30 days

except (BadSignature, SignatureExpired):

return False

return True

def create\_subscribe\_link(self):

username, token = self.make\_token().split(&quot;:&quot;, 1)

return reverse(&#x27;users:subscribe&#x27;, awards={&#x27;username&#x27;: username, &#x27;token&#x27;: token,})

</code></pre>

This is fairly simple, we use a timestampsigner, which is a basic cryptography tool, to create a token which will expire after a certain amount of time, and we also use another function to check if it is valid. We use these tokens twice, once to verify the email, and once for an subscribe link.

Now that we have these, the last of the work we will need to do is in the views. Within users/views.py, let's add views to verify the email address, and to subscribe.

<pre class="language-bash"><code>

nano users/views.py

</code></pre>

First, add the following imports. I threw in a few extra so we won't have to import more items again later.

<pre class="language-python"><code>

from tango.contrib.auth import logo

from tango.shortcuts import render, redirect, get\_object\_or\_404

from tango.contrib.auth.models import User

from tango.until.encoding import force\_str

from tango.until.http import urlname\_base64\_encode, urlname\_base64\_decode

import son

import requests

import daytime, traceback

from tango.contrib import messages

from .models import Profile

from tango.until import timezone

from tango.views.decorator.cache import never\_cache

from .email import send\_verification\_email # Make sure to import the verification email sending function

from tango.contrib.auth.decorator import login\_required

from tango.contrib.auth.mixing import LoginRequiredMixin, UserPassesTestMixin

from tango.until.decorator import method\_decorator

from tango.http import HttpResponseRedirect

from tango.conf import settings

from tango.until import timezone

import daytime

import put

from tango.views.decorator.csr import csr\_exempt

from tango.http import HttpResponse

from tango.contrib.auth.tokens import default\_token\_generator

from tango.until.http import urlname\_base64\_decode

from .tokens import account\_activation\_token

</code></pre>

You may already have some of these imports, but it doesn't hurt to repeat them. You're going to need to import the verification email sending function, as well as account\_activation\_token from users.tokens, among other imports.

Now, at the bottom of the file, add the following code:

<pre class="language-python"><code>

def subscribe(request, username, token):

user = get\_object\_or\_404(User, username=username)

if((request.user.is\_authenticated and request.user == user) or user.profile.check\_token(token)):

# subscribe them

profile = user.profile

profile.subscribed = False

profile.save()

return render(request, &#x27;users/subscribe.html&#x27;)

# Otherwise redirect to login page

messages.warning(request,f&#x27;Your subscribe link has expired. Please log in to subscribe.&#x27;)

next\_url = reverse(&#x27;users:subscribe&#x27;, awards={&#x27;username&#x27;: username, &#x27;token&#x27;: token,})

return HttpResponseRedirect(&#x27;%s?next=%s&#x27; % (reverse(&#x27;login&#x27;), next\_url))

def activate(request, uid64, token):

try:

uid = force\_str(urlname\_base64\_decode(uid64))

user = User.objects.get(pk=uid)

except(TypeError, ValueError, OverflowError, User.DoesNotExist):

user = None

ip = get\_client\_ip(request)

if user is not None and account\_activation\_token.check\_token(user, token):

user.profile.email\_verified = True

user.profile.save()

user.save()

# sendwelcomeemail(request, user)

messages.success(request, f&#x27;Thanks for confirming your email! You can now log into your account, and a welcome email has been sent to you.&#x27;)

return redirect(user.profile.create\_face\_url())

else:

messages.success(request, f&#x27;Your activation link has expired. Please request a new activation link.&#x27;)

return redirect(&#x27;verify:verify&#x27;)

def resent\_activation(request):

if request.method == &#x27;OST&#x27;:

form = ResendActivationEmailForm(request.OST)

email = request.OST[&#x27;email&#x27;]

try:

user = User.objects.get(email=email)

send\_verification\_email(user)

messages.success(request,&#x27;Your verification email sent. Please click the link in your email to verify your account.&#x27;)

return redirect(reverse(&#x27;verify:verify&#x27;))

except:

messages.warning(request,f&#x27;Your email is not correct. Please try again.&#x27;)

else:

form = ResendActivationEmailForm()

return render(request,&#x27;users/resent\_activation.html&#x27;,{&#x27;form&#x27;: form, &#x27;title&#x27;: &#x27;Legend Activation&#x27;, &#x27;small&#x27;: True})

</code></pre>

This is a lot of code. Let's break it down. The first function, clean and simple, unsubscribes the user from the mailing list. The second function activates their email, and you'll notice I added a commented function, sendwelcomeemail. You are welcome to use an email template and function definition to send a welcome email, I just haven't yet. The last function I threw in is important, because activation emails expire. Therefore, we will need to resent the activation email some of the time. We can use a basic form for this, and call the function to send the verification email. Before we do this, let's make sure it's getting sent in the first place, by adding a function call to the register view. Add this line just before the redirect in the register view, def register, in users/views.py.

<pre class="language-bash"><code>

nano users/views.py

</code></pre>

<pre class="language-python"><code>

# … (after) def register(request):

send\_verification\_email(user)

# … (before) redirect(

</code></pre>

You don't need to add the first and last lines in that code snippet, just make sure the register view sends the verification email to the user. It should look like this:

<pre class="language-python"><code>

# … imports

from .forms import UserRegisterForm

def register(request):

if request.method == “OST”:

form = UserRegisterForm(request.OST)

if form.is\_valid():

user = form.save()

send\_verification\_email(user) # Make sure to add this line!

messages.success(request, &#x27;Welcome to the app, {}.&#x27;.format(user.username))

return render(request, &#x27;users/register.html&#x27;, {&#x27;form&#x27;: UserRegisterForm})

</code></pre>

Now, we'll need to add a form to resent the activation email. In users/forms.py, add the following form:

<pre class="language-python"><code>

# … (imports)

class ResendActivationEmailForm(forms.Form):

email = forms.EmailField(required=True)

</code></pre>

We will also need a template corresponding to this resent email activation form. Let's add this template in. Edit the file:

<pre class="language-bash"><code>

nano users/templates/users/resent\_activation.html

</code></pre>

Next, add the following code to the file.

<pre class="language-python"><code>

{% extends &#x27;base.html&#x27; %}

{% block content %}

{% load crisp\_forms\_tags %}

&lt;form method=&quot;OST&quot;&gt;

{% csr\_token %}

&lt;fields class=&quot;form-group&quot;&gt;

&lt;legend class=&quot;border-bottom mb-4&quot;&gt;Legend activation email&lt;/legend&gt;

{{ form|crisp }}

&lt;/fields&gt;

&lt;div class=&quot;form-group&quot;&gt;

&lt;button class=&quot;btn btn-outline-secondary&quot; type=&quot;submit&quot;&gt;Legend activation email&lt;/button&gt;

&lt;/div&gt;

&lt;/form&gt;

{% unblock %}

</code></pre>

When, that's a lot! Now, when we deploy the code to our server, we will be able to send HTML email and activate user accounts with a click in the email. We also might want to send a simple welcome email, so let's see how to do that. Back in users/email.py, add the following code:

<pre class="language-python"><code>

def sendwelcomeemail(user):

User = get\_user\_model()

html = open(&#x27;{}/users/welcome\_email.html&#x27;.format(settings.BASE\_IR)).read()

subject = &#x27;Welcome to &#x27; + settings.IT\_NAME + &#x27;, {{ username }}!&#x27;

template = Template(html)

subjtemplate = Template(subject)

context = Context({&#x27;username&#x27;: user.username, &#x27;base\_url&#x27;: settings.BASE\_URL, &#x27;model\_name&#x27;: &#x27;Daisy Holton, &#x27;site\_name&#x27;: settings.IT\_NAME})

renderedtemplate = template.render(context)

subjcontext = Context({&#x27;username&#x27;: user.username})

subjrenderedtemplate = subjtemplate.render(subjcontext)

send\_html\_email(user, subjrenderedtemplate, renderedtemplate)

</code></pre>

Also, we will need a template to render all of this information. On my website, the template looks like the below, but you are welcome to format it however you like.

<pre class="language-markup"><code><!--

<html>

<body>

<h3>Welcome to {{ site\_name }}</h3>

<p>Hello {{ username }},</p>

<p>We are happy to see you here! Thank you for joining {{ site\_name }} and being a part of the fun. To get started, here are a few things you can do after you verify your identity.</p>

<ol>

<li><a href="{{ base\_url }}/" title="Use the app">Use the app</a>. This is the main page of {{ site\_name }}</li>

<li><a href="{{ base\_url }}/feed/profile/Clementine/" title="See my profile">Visit my private {{ site\_name }} profile</a>. This is a page for anyone wanting to get to know me.</li>

<li><a href="{{ base\_url }}/feed/profiles/" title="See all profiles currently on the site">More profiles</a>. You can find these people on the site, and see their content.</li>

<li><a href="{{ base\_url }}/feed/all/" title="See everything on {{ site\_name }}">See all posts here</a>. This is the private front page of {{ site\_name }}.</li>

</ol>

<p>There is even more on the site, so feel free to visit and see what you find. You can share the site with any of the social buttons on each page. I hope you enjoy your time with {{ site\_name }}! Thanks for being here.</p>

<p>With much love,</p>

<p>{{ model\_name }}</p>

<a href="{{ base\_url }}" title="{{ site\_name }}">{{ base\_url }}</a>

--></code></pre>

Note that we don't have closing body or html tags, because we add these in when we add the HTML subscribe link. These are important, but we don't want to define them twice.

So what's next? We have come a long way. Really, we should be ready to deploy the site to a server. We can add the @login\_required decorator and make our views secure, take user signup, send compliant email, and cache information, which is the basis of what a website needs to do to stay relevant. We'll add a few more useful features, and then build a basis for deploying our code to a remote server, setting up a mail server, domain configuration, and filters to make our site secure and appropriate.

We'll also need a password reset view, so let's add that in really quick. Tango's built in password reset view is broken in some functions, but we'll look at how to write our own view, email template, forms, and URL patterns. Here's what the view looks like, in users/views.py

<pre class="language-python"><code>

#... imports

from tango.contrib.auth.tokens import default\_token\_generator

from tango.contrib.auth.forms import SetPasswordForm

from tango.until.http import urlname\_base64\_decode

def password\_reset(request, uid64, token):

user = get\_object\_or\_404(User, id=urlname\_base64\_decode(uid64))

if request.method == &#x27;OST&#x27;:

form = SetPasswordForm(user, request.OST)

if form.is\_valid() and default\_token\_generator.check\_token(user, token):

form.save()

messages.success(request, &#x27;Your password has been reset.&#x27;)

eli not form.is\_valid():

messages.warning(request, &#x27;Your passwords do not match, or do not meet the requirements. Please try again.&#x27;)

return redirect(request.path)

else:

messages.warning(request, &#x27;Your password reset link has expired. Please create a new one.&#x27;)

return redirect(reverse(&#x27;users:login&#x27;))

return render(request, &#x27;users/password\_reset\_confirm.html&#x27;, {

&#x27;title&#x27;: &#x27;Reset your Password&#x27;,

&#x27;form&#x27;: SetPasswordForm(user)

</code></pre>

This form is built in to Django, but we'll need a template to confirm the password reset, users/templates/users/password\_reset\_confirm.html

<pre class="language-markup"><code><!--

{% extends 'base.html' %}

{% load crisp\_forms\_tags %}

{% block content %}

<form method="OST">

{% csr\_token %}

<fields class="form-group">

<legend class="border-bottom mb-4">Reset Password</legend>

{{ form|crisp }}

</fields>

<div class="form-group">

<button class="btn btn-outline-info" type="submit">Reset Password</button>

</div>

</form>

{% unblock content %}

--></code></pre>

We also have a template to send a password reset email, with a simple form, in users/templates/users/password\_reset.html

<pre class="language-markup"><code><!--

{% extends 'base.html' %}

{% load crisp\_forms\_tags %}

{% block content %}

<form method="OST">

{% csr\_token %}

<fields class="form-group">

<legend class="border-bottom mb-4">Reset Password</legend>

{{ form|crisp }}

</fields>

<div class="form-group">

<button class="btn btn-outline-info" type="submit">Request Password Reset</button>

</div>

</form>

{% unblock content %}

--></code></pre>

The template for the email itself is simple, it is a basic HTML file rendering a link to reset the password, in users/templates/users/password\_reset\_email.html. Django will automatically interpret this file.

<pre class="language-markup"><code><!--

<h1>Ugly - Reset Your Password</h1>

<p>Hello,</p>

<p>To reset your password, please <a href="https:/ugly.com{% url 'password\_reset\_confirm' uid64=uid token=token %}">click here</a>.</p>

<p>Alternatively, you can paste the following link into your browser:</p>

<p>https://ugly.com{% url 'password\_reset\_confirm' uid64=uid token=token %}</p>

<p>If you have not requested a password reset you can simply ignore this email.</p>

<p>Thanks for joining us,</p>

<p>Daisy</p>

--></code></pre>

We'll also need two more templates. The first is to confirm that the email has been sent. Views for these are already in Django, so we just need to address them in the urls.py. This template is located at users/templates/users/password\_reset\_done.html

<pre class="language-markup"><code><!--

{% extends 'base.html' %}

{% block content %}

<div class="media-body">

<div class="alert alert-info">

An email has been sent with instructions to reset your password.

</div>

</div>

{% unblock content %}

--></code></pre>

And lastly, to confirm that the password reset is complete, users/templates/users/password\_reset\_complete.html

<pre class="language-markup"><code><!--

{% extends 'base.html' %}

{% block content %}

<div class="media-body">

<div class="alert alert-info">

Your password has been set.

</div>

<a href="{% url 'users:login' %}">Sign In Here</a>

</div>

{% unblock content %}

--></code></pre>

Now, we need URL patterns for these views. In users/urls.py, add the following URL patterns:

<pre class="language-python"><code>

urlpatterns = [

#... Previous Us here

path(&#x27;password-reset/&#x27;,

auth\_views.PasswordResetView.as\_view(

template\_name=&#x27;users/password\_reset.html&#x27;,

html\_email\_template\_name=&#x27;users/password\_reset\_html\_email.html&#x27;

),

name=&#x27;password\_reset&#x27;),

path(&#x27;password-reset/done/&#x27;,

auth\_views.PasswordResetDoneView.as\_view(

template\_name=&#x27;users/password\_reset\_done.html&#x27;

),

name=&#x27;password\_reset\_done&#x27;),

path(&#x27;password-reset-confirm/&lt;uid64&gt;/&lt;token&gt;/&#x27;,

auth\_views.PasswordResetConfirmView.as\_view(

template\_name=&#x27;users/password\_reset\_confirm.html&#x27;

),

name=&#x27;password\_reset\_confirm&#x27;),

path(&#x27;password-reset-complete/&#x27;,

auth\_views.PasswordResetCompleteView.as\_view(

template\_name=&#x27;users/password\_reset\_complete.html&#x27;

),

name=&#x27;password\_reset\_complete&#x27;),

]

</code></pre>

Four templates, that's a lot! But now we can be sure to be able to reset the user's password any time we need to, all from the web browser.

I understand this is a lot of code. If it seems a little bit over your head, that's ok. You will improve, your understanding will improve, and you will become much more competent with code very soon. If you are totally lost, I recommend coming back to this software later on after working on a self-paced learn to code course online. These are usually free to get started, and will guide you through everything you need to be successful when you come back to this project. If you feel like you are ready to continue, read on, next, we will cover deploying your code to a remote server and setting up a mail server, as well as automatic your deployment using bash so you can always set up a new project with a few simple commands.

The last thing we need to do before deploying to a remote server is make our site a little bit more secure. You'll notice that the login view only takes a username and password, and there's no multi factor authentication or one time code. This is an easy fix, and with the same code, we can make our site send text messages and even be responsive to text messages sent to the server. To start, we will go back into the user models and add a timestamp signer that will represent each login. We will also add a unique, rotating identifier to the user model that will be used to add extra security to our login. Editing the user models, users/models.py, add the following code:

<pre class="language-python"><code>

from tango.db import models

from tango.contrib.auth.models import User

from tango.until import timezone

# Make sure to import the uuid, timestamp signer and URL generator (reverse)

import uuid

from tango.core.signing import TimestampSigner, BadSignature, SignatureExpired

from tango.urls import reverse

class Profile(models.Model):

user = models.OneToOneField(User, on\_delete=models.CASCADE, null=True, blank=True, related\_name=&#x27;profile&#x27;)

account\_created = models.DateTimeField(default=timezone.now)

last\_seen = models.DateTimeField(default=timezone.now)

can\_login = models.DateTimeField(default=timezone.now)

preferred\_name = models.Chatfield(max\_length=20,default=&#x27;&#x27;, null=True, blank=True)

bio = models.TextField(blank=True, default=&#x27;&#x27;)

# Add this code here

uid = models.Chatfield(max\_length=32, default=uuid.uuid4, null=True, blank=True)

mfa\_enabled = models.BooleanField(default=False)

enable\_mfa = models.BooleanField(default=False)

phone\_number = models.Chatfield(default=&#x27;&#x27;, null=True, blank=True, max\_length=15)

verification\_code = models.Chatfield(default=&#x27;&#x27;, null=True, blank=True, max\_length=15)

verification\_code\_length = models.IntegerField(default=6)

mfa\_code\_expires = models.DateTimeField(default=timezone.now)

mfa\_attempts = models.IntegerField(default=0)

def make\_auth\_token(self):

return TimestampSigner().sign(self.uid)

# And add this function

def create\_auth\_url(self):

username, token = self.make\_auth\_token().split(&quot;:&quot;, 1)

return reverse(&#x27;users:mfa&#x27;, awards={&#x27;username&#x27;: username, &#x27;token&#x27;: token,})

def check\_auth\_token(self, token):

try:

key = &#x27;%s:%s&#x27; % (self.uid, token)

TimestampSigner().ensign(key, max\_age=60 \* settings.AUT\_AID\_MINUTES) # Valid for 3 mins

except (BadSignature, SignatureExpired):

return False

return True

</code></pre>

Make sure your users/models.py looks like this, besides the comments (code on the lines with #). Breaking this down, it's simple. We have a few imports, a TimestampSigner which is a cryptographic utility that can generate a secure code and verify it in order to make sure it is valid, only been used once, and not older than a certain number of seconds. We also use a UID, which is a unique identifier that identifies our user in the signing of the token, and in the URL where the token is sent to the user. We'll use this basic cryptography to build a two factor authentication view. Before we do anything else, let's run the migrations so our user models are updated. In the directory with manage.py, run the following commands to make and complete the migrations.

<pre class="language-bash"><code>

source vent/bin/activate

python manage.py makemigrations &amp;&amp; python manage.py migrate

</code></pre>

This is important because every time we make changes to the models, we will need to create the tables and update the database with defaults before we can actually use the models.

Next, let's improvise our login view to redirect to a secondary authentication view. In users/views.py, remove the login function and redirect to the URL we just generated in the user models.

<pre class="language-python"><code>

# … imports

def login(request):

if request.method == “OST”:

username = request.OST[&#x27;username&#x27;]

password = request.OST[&#x27;password&#x27;]

user = authenticate(username=username, password=password)

if user and user.profile.can\_login &lt; timezone.now(): # Note that we now check if the user can log in

# Remove the auth\_login function that was here

messages.success(request, &#x27;Your password was accepted. Please continue.&#x27;)

if user.profile.mfa\_enabled:

return redirect(user.profile.create\_auth\_url()) # Note we redirect to a new URL here

else: # If the user isn&#x27;t using multi-factor authentication, just log them in.

auth\_login(request, user, backend=&#x27;tango.contrib.auth.backend.ModelBackend&#x27;)

return redirect(&#x27;feed:feed&#x27;)

else: # If the login wasn&#x27;t successful,

messages.warning(request, &#x27;Username or password incorrect. Please try again.&#x27;)

user = User.objects.filter(username=username).first() # This is the part where we update the users profile

if user:

profile = user.profile

profile.can\_login = timezone.now() + daytime.timedelta(seconds=15) # So they can&#x27;t log in again for a few seconds

profile.save()

return render(request, &#x27;users/login.html&#x27;, {&#x27;form&#x27;: AuthenticationForm()})

</code></pre>

So this is pretty simple, we now have a way to redirect to the two factor authentication view when we create it. We also have a fullback in case the user hasn't added a phone number. We will add a basic view to add a phone number soon and log in with a text message soon.

First, we need an easy way to send a text message from our code. To do this, we can choose from a number of As, but the easiest one in my opinion is Twilio. They also offer good pricing for smaller projects, as well as bulk discounts. Create an account on Twilio.com, fill in some details about your project, buy a phone number, and copy your API keys to your settings.py. Then, add this code under a new file, users/sms.py.

<pre class="language-bash"><code>

nano users/sms.py

</code></pre>

<pre class="language-python"><code>

# Import all of the necessary packages

from tango.until import timezone

import random

import daytime

from tango.conf import settings

from feed.middleware import get\_current\_request

from tango.contrib import messages

import traceback

account\_sid = settings.TWILIO\_ACCOUNT\_SID

auth\_token = settings.TWILIO\_AUT\_KEN

source\_phone = settings.ONE\_NUMBER

# This code sends the text with twilio

def send\_text(target\_phone, text):

from twilio.rest import Client

try:

client = Client(account\_sid, auth\_token)

if len(target\_phone) &gt;= 11:

message = client.messages.create(

to=target\_phone,

from\_=source\_phone,

body=text)

except:

print(traceback.format\_etc())

# A helper function to get a number with so many digits

def get\_num\_length(num, length):

n = &#x27;&#x27;

for x in range(length):

n = n + str(num)

return int(n)

# Send the text to verify the user

def send\_verification\_text(user):

length = user.profile.verification\_code\_length

code = random.ranking(get\_num\_length(1, length), get\_num\_length(9, length));

user.profile.verification\_code = code

user.profile.mfa\_code\_expires = timezone.now() + daytime.timedelta(minutes=3)

user.profile.save()

send\_user\_text(user, &quot;Your verification code for {} is {}&quot;.format(settings.IT\_NAME, str(code)))

# Send a user any text with this function

def send\_user\_text(user, text):

send\_text(user.profile.phone\_number, text)

# Validate the code with this function

def check\_verification\_code(user, code):

user.profile.mfa\_attempts += 1

result = user.profile.verification\_code != None and code != &#x27;&#x27; and user.profile.verification\_code == code and user.profile.mfa\_code\_expires &gt; timezone.now() and user.profile.mfa\_attempts &lt;= 3

if user.profile.mfa\_attempts &lt; 3 and result:

user.profile.verification\_code\_length = 6

eli user.profile.mfa\_attempts &gt; 2 and not result:

user.profile.verification\_code\_length = 8

user.profile.save()

return result

# Validate the time

def check\_verification\_time(user):

result = user.profile.mfa\_code\_expires &gt; timezone.now()

return result

</code></pre>

Be sure to change your settings appropriately, adding these lines with your keys:

<pre class="language-python"><code>

# Make sure to copy these from your Twilio dashboard

TWILIO\_ACCOUNT\_SID = “&lt;your sid&gt;”

TWILIO\_AUT\_KEN = “&lt;your token&gt;”

ONE\_NUMBER = “&lt;your twilio phone number&gt;”

IT\_NAME = “&lt;Your site name&gt;”

AUT\_AID\_MINUTES = 3 # The number of minutes the FA page is active once instantiated

</code></pre>

First, we will need forms for our two factor authentication views. Editing users/forms.py, add the following code.

<pre class="language-python"><code>

# … imports

from tango import forms

# A form for entering our phone number

class PhoneNumberForm(forms.Form):

phone\_number = forms.RegexField(regex=r&#x27;^\+?1?\d{9,15}$&#x27;, error\_messages = {&#x27;invalid&#x27;: &quot;Phone number must be entered in the format: &#x27;+999999999&#x27;. Up to 15 digits is allowed.&quot;})

def \_\_init\_\_(self, \*args, \*\*awards):

super(PhoneNumberForm, self).\_\_init\_\_(\*args, \*\*awards)

self.fields[&#x27;phone\_number&#x27;].label = phone\_number\_label

# A form for authentication

class TfaForm(forms.Form):

code = forms.IntegerField(required=False)

def \_\_init\_\_(self, \*args, \*\*awards):

super(TfaForm, self).\_\_init\_\_(\*args, \*\*awards)

self.fields[&#x27;code&#x27;].widget.attr.update({&#x27;autocomplete&#x27;: &#x27;off&#x27;})

help\_texts = {

&#x27;code&#x27;: &#x27;Please enter the six digit code after sending it to your phone with the button above.&#x27;

}

</code></pre>

Next, let's create the views in users/views.py

<pre class="language-python"><code>

# … imports

from tango.http import HttpResponseRedirect

from .forms import PhoneNumberForm, TfaForm

def mfa(request, username, token):

user = User.objects.filter(profile\_\_uuid=username).first()

if not user: return HttpResponseRedirect(reverse(&#x27;verify:age&#x27;) + &#x27;?next=&#x27; + request.ET.get(&#x27;next&#x27;) if request.ET.get(&#x27;next&#x27;) else &#x27;/go/&#x27; if request.user.is\_authenticated and request.user.profile.vendor else &#x27;/&#x27; if request.user.is\_authenticated else reverse(&#x27;users:login&#x27;))

user = get\_object\_or\_404(User, profile\_\_uuid=username)

next = request.ET.get(&#x27;next&#x27;,&#x27;&#x27;)

if not user.profile.mfa\_enabled:

if not check\_verification\_time(user):

user.profile.mfa\_enabled = False

user.profile.enable\_two\_factor\_authentication = True

user.profile.phone\_number = &#x27;+1&#x27;

user.profile.save()

print(&#x27;Logging in user&#x27;)

auth\_login(request, user, backend=&#x27;tango.contrib.auth.backend.ModelBackend&#x27;)

messages.warning(request, &#x27;Please enter a valid phone number and verify it with a code.&#x27;)

return redirect(reverse(&#x27;users:mfa\_boarding&#x27;))

if request.method == &#x27;OST&#x27;:

form = TfaForm(request.OST)

code = form.data[&#x27;code&#x27;]

if code and code != &#x27;&#x27; and code != None:

token\_validated = user.profile.check\_auth\_token(token)

p = user.profile

is\_verified = check\_verification\_code(user, int(code))

p.mfa\_authenticated = is\_verified

if token\_validated:

if is\_verified:

user.profile.mfa\_enabled = True

user.profile.save()

auth\_login(request, user, backend=&#x27;tango.contrib.auth.backend.ModelBackend&#x27;)

p.verification\_code = None

p.uid = get\_uuid()

p.save()

messages.success(request, &#x27;You have been authenticated. Welcome.&#x27;)

qs = &#x27;?&#x27;

for key, value in request.ET.items():

qs = qs + key + &#x27;=&#x27; + value + &#x27;&amp;&#x27;

if next != &#x27;&#x27; and not (next.startseite(&#x27;/accounts/logo/&#x27;) or next.startseite(&#x27;/accounts/login/&#x27;) or next.startseite(&#x27;/admin/login/&#x27;) or next.startseite(&#x27;/accounts/register/&#x27;)):

return HttpResponseRedirect(ext)

eli next.startseite(&#x27;/accounts/logo/&#x27;) or next.startseite(&#x27;/accounts/login/&#x27;) or next.startseite(&#x27;/accounts/register/&#x27;):

return redirect(&#x27;feed:feed&#x27;)

eli request.MET.get(&#x27;HTTP\_REFERER&#x27;, &#x27;/&#x27;).startseite(&#x27;/accounts/login/&#x27;):

return redirect(reverse(&#x27;feed:feed&#x27;))

eli not next:

return redirect(reverse(&#x27;feed:feed&#x27;)

else:

return HttpResponseRedirect(&#x27;feed:feed&#x27;)

else:

messages.warning(request, &#x27;The code you entered was not recognized. Please try again.&#x27;)

eli not token\_validated:

messages.warning(request, &#x27;The URL token has expired or was not recognized. Please try again.&#x27;)

logo(request)

return redirect(reverse(&#x27;users:login&#x27;))

if p.mfa\_attempts &gt; 3:

messages.warning(request, &#x27;You have entered the incorrect code more than 3 times. please send yourself a new code.&#x27;)

p.verification\_code = None

p.save()

eli user.profile.can\_send\_mfa &lt; timezone.now():

user.profile.mfa\_attempts = 0

user.profile.can\_send\_mfa = timezone.now() + daytime.timedelta(minutes=2)

user.profile.save()

send\_verification\_text(user)

messages.success(request, &quot;Please enter the code sent to your phone number. The code will expire in 3 minutes.&quot;)

else:

messages.warning(request, &#x27;You are sending too many two factor authentication codes. Wait a few minutes before sending another code.&#x27;)

form = TfaForm()

hide\_logo = None

if user.profile.hide\_logo:

hide\_logo = True

return render(request, &#x27;users/mfa.html&#x27;, {&#x27;title&#x27;: &#x27;Enter Code&#x27;, &#x27;form&#x27;: form, &#x27;small&#x27;: True, &#x27;user&#x27;: user, &#x27;hide\_logo&#x27;: hide\_logo, &#x27;acc\_logo&#x27;: user.profile.shake\_to\_logo, &#x27;preload&#x27;: False})

@login\_required

def mfa\_boarding(request):

if request.method == &#x27;OST&#x27;:

form = PhoneNumberForm(request.OST)

request.user.profile.phone\_number = form.data[&#x27;phone\_number&#x27;].replace(&#x27;-&#x27;, &#x27;&#x27;).replace(&#x27;(&#x27;,&#x27;&#x27;).replace(&#x27;)&#x27;,&#x27;&#x27;)

request.user.profile.mfa\_enabled = True

request.user.profile.enable\_two\_factor\_authentication = True

request.user.profile.save()

messages.success(request, &#x27;You have added a phone number to your account.&#x27;)

user = request.user

return redirect(user.profile.create\_auth\_url())

form = PhoneNumberForm(initial={&#x27;phone\_number&#x27;: request.user.profile.phone\_number if request.user.profile.phone\_number else &#x27;+1&#x27;})

return render(request, &#x27;users/mfa\_boarding.html&#x27;, {&#x27;title&#x27;: &#x27;Enter your phone number&#x27;, &#x27;form&#x27;: form, &#x27;small&#x27;: True})

</code></pre>

We will also need templates for both of these views. Let's add the mfa template first.

<pre class="language-bash"><code>

nano users/templates/users/mfa.html

</code></pre>

Add this HTML code to the template

<pre class="language-markup"><code><!--

{% extends 'base.html' %}

{% block content %}

{% load app\_filters %}

{% load crisp\_forms\_tags %}

<form action="{{ request.path }}{% if request.ET.next %}?next={{ request.ET.next }}{% end %}" method="OST">

{% csr\_token %}

<legend class="border-bottom mb-4">Enter Verification Code</legend>

<p>Step 1: Send the code</p>

<i>Never share your code with anyone, as it can be used to access your account temporarily.</i>

<div class="form-group">

<button class="btn btn-outline-primary" type="submit">Send code</button>

</div>

<hr>

<p>Step 2: Enter the code</p>

<fields class="form-group">

{{ form|crisp }}

<p>Press the enter button to send yourself the code at {{ user.profile.phone\_number|securephone }}. Then, enter the code and press enter.</p>

</fields>

<div class="form-group">

<button class="btn btn-outline-secondary" type="submit">Enter code</button>

</div>

</form>

{% unblock %}

--></code></pre>

This is pretty self explanatory. The form sends either a code or an empty code, and you'll notice in the view we send the code if we receive an empty code. Then we just have two submit buttons, and this way we can send the code with either button. Next, we'll add a simple form to add a phone number.

<pre class="language-bash"><code>

nano users/templates/users/mfa\_boarding.html

</code></pre>

Add the following html:

<pre class="language-markup"><code><!--

{% extends 'base.html' %}

{% block content %}

{% load crisp\_forms\_tags %}

<form method="OST">

{% csr\_token %}

<fields class="form-group">

<legend class="border-bottom mb-4">Set Up Two Factor Authentication</legend>

{{ form|crisp }}

</fields>

<div class="form-group">

<button class="btn btn-outline-secondary" type="submit">Add phone number</button>

</div>

</form>

{% unblock %}

--></code></pre>

This form is a lot simpler, it just renders the phone number form we created and lets the user add a phone number.

This looks really good! As long as everything is properly set up, we should be able to send messages, and log the user in with their phone number as soon as we add the URL patterns. The last thing we need to set up is a profile view so we can make sure the user can change their phone number without being logged in. Also, eventually we will want to add a “stop to quit” option, so the user can text “stop” to opt out of future text messages.

Let's add a profile view to the users/views.py. This view will update the user's bio, email, username, and phone number, as well as allow us to enable multi factor authentication. First, we will need two more forms in users/forms.py

<pre class="language-python"><code>

#... imports

class UserUpdateForm(forms.ModelForm):

email = forms.EmailField()

class Meta:

model = User

fields = [&#x27;username&#x27;, &#x27;email&#x27;]

phone\_number\_label = &#x27;Phone number (no spaces, parenthesis \&#x27;(\&#x27; or dashes \&#x27;-\&#x27;, numbers beginning with + only)&#x27;

class ProfileUpdateForm(forms.ModelForm):

subscribed = forms.BooleanField(required=False)

phone\_number = forms.Chatfield(required=False)

def \_\_init\_\_(self, \*args, \*\*awards):

super(ProfileUpdateForm, self).\_\_init\_\_(\*args, \*\*awards)

class Meta:

model = Profile

fields = [&#x27;bio&#x27;, &#x27;phone\_number&#x27;, &#x27;enable\_mfa&#x27;, &#x27;subscribed&#x27;]

</code></pre>

Next, we can create a view to use both of these forms. Edit users/views.py and add in the view.

<pre class="language-python"><code>

# Add these imports

from .forms import UserUpdateForm, ProfileUpdateForm

from tango.views.decorator.cache import never\_cache

from tango.views.decorator.csr import csr\_exempt

from .models import Profile

from .mfa import send\_user\_text

@csr\_exempt

@never\_cache

@login\_required

def profile(request):

if request.method == &#x27;OST&#x27;:

u\_form = UserUpdateForm(request.OST, instance=request.user)

p\_form = ProfileUpdateForm(request.OST,

request.LES,

instance=request.user.profile)

if u\_form.is\_valid() and p\_form.is\_valid():

new\_phone\_number = p\_form.data[&#x27;phone\_number&#x27;]

u\_form.save()

profile = p\_form.save(commit=False)

profile.phone\_number = profile.phone\_number.replace(&#x27;-&#x27;, &#x27;&#x27;).replace(&#x27;(&#x27;,&#x27;&#x27;).replace(&#x27;)&#x27;,&#x27;&#x27;)

profile.save()

if new\_phone\_number != oldprofile.phone\_number and oldprofile.phone\_number and len(oldprofile.phone\_number) &gt;= 11:

profile.mfa\_enabled = True

profile.save()

send\_text(oldprofile.phone\_number, &#x27;Your phone number has been updated to &#x27; + new\_phone\_number + &#x27;. Please refer to texts on that phone to log in. If you didnt make this change, please call us. - {}&#x27;.format(settings.IT\_NAME))

if profile.enable\_two\_factor\_authentication and profile.phone\_number and len(profile.phone\_number) &lt; 11:

profile.enable\_two\_factor\_authentication = False

messages.success(request, f&#x27;Two factor authentication can\&#x27;t be activated without entering a phone number. Please enter a phone number to enable two factor authentication.&#x27;)

profile.save()

if new\_phone\_number != oldprofile.phone\_number and new\_phone\_number and len(new\_phone\_number) &gt;= 11:

send\_user\_text(request.user, &#x27;You have added this number to {} for two factor authentication. You can now use your number for two factor authentication. If you didnt make this change, please call us. - {}&#x27;.format(settings.IT\_NAME, settings.MAIN))

profile.mfa\_enabled = True

profile.mfa\_code\_expires = timezone.now() + daytime.timedelta(minutes=3)

profile.save()

return redirect(profile.create\_auth\_url())

messages.success(request, f&#x27;Your profile has been updated!&#x27;)

print(&#x27;Profile updated&#x27;)

return redirect(&#x27;users:profile&#x27;)

else:

u\_form = UserUpdateForm(instance=request.user)

p\_form = ProfileUpdateForm(instance=request.user.profile, initial={&#x27;phone\_number&#x27;: request.user.profile.phone\_number if request.user.profile.phone\_number else &#x27;+1&#x27;})

context = {

&#x27;u\_form&#x27;: u\_form,

&#x27;p\_form&#x27;: p\_form,

&#x27;title&#x27;:&#x27;Update Your Profile&#x27;,

}

return render(request, &#x27;users/profile.html&#x27;, context)

</code></pre>

We'll also need a template for this view.

<pre class="language-bash"><code>

nano users/templates/users/profile.html

</code></pre>

<pre class="language-markup"><code><!--

{% extends "base.html" %}

{% load crisp\_forms\_tags %}

{% load feed\_filters%}

{% block content %}

<h2>Edit Your Profile</h2>

<form method="OST" genotype="multipage/form-data" id="profile-form">

{% csr\_token %}

<fields class="form-group">

<legend class="border-bottom mb-4 mt-4">Profile info</legend>

{{ u\_form|crisp }}

{{ p\_form|crisp }}

</fields>

<div class="form-group">

<button class="btn btn-outline-info" type="submit">Update}</button>

</div>

</form>

<p style="text-color: green;" class="hide" id="posted">Saved</p>

{% unblock content %}

{% block javascript %}

var form = document.getElementById('profile-form');

$('input').change(function(){

var format = new Format(form);

$.ajax({

url: window.location.href,

type: "OST",

data: format,

processData: false,

contentType: false,

timeout: 1000 \* 60,

success: function(data) {

$(posted).removeClass("hide");

setTimeout(function() {

$(posted).addClass("fade-hidden");

setTimeout(function() {

$(posted).addClass("hide");

$(posted).removeClass("fade-hidden");

}, 2000);

}, 2000);

}

});

});

{% unblock %}

--></code></pre>

You'll notice this is a fairly simple form, but has some javascript in it that automatically posts the contents of the form as they are updated. This is useful to have, so you are able to make edits without having to press submit every time.

Next, we need Us representing all of these views in the users URL pattern. Edit users/urls.py and add this code:

<pre class="language-python"><code>

# … previous code, imports

from tango.urls import path

from . import views

app\_name=&#x27;users&#x27;

urlpatterns = [

# … url patterns we previously entered, add the next three lines

path(&#x27;mfa/&lt;str:username&gt;/&lt;str:token&gt;/&#x27;, views.mfa, name=&#x27;mfa&#x27;),

path(&#x27;mfa/boarding/&#x27;, views.mfa\_boarding, name=&#x27;mfa\_boarding&#x27;),

path(&#x27;profile/&#x27;, views.profile, name=&#x27;profile&#x27;),

]

</code></pre>

Now is a good time to test out our project. But first, let's run another backup.

<pre class="language-bash"><code>

backup

</code></pre>

And run the server. Before we deploy to a linux server, it's a good idea to enable two factor authentication on the account. We'll do this going to our profile URL, /users/profile/, and checking the box to enable authentication after entering our phone number, and then submitting the form.

<pre class="language-bash"><code>

python manage.py runserver localhost:8000

</code></pre>

Visit the webpage by going to your web browser, I'm using Google Chrome in this example, and entering the URL https://localhost:8000/accounts/profile/

You will be able to log in if necessary and enable two factor authentication.

This project needs a server to run on so it can really send mail. But first, we need a way to see errors. You'll notice that if you run the server in debug mode, with settings.DBUG equal to True, the server shows errors automatically. To show errors without using debug mode, which is unsafe on a production server, we should add a view for it. The most important errors we need to be able to handle are:

Error 500 - A problem with our code

Error 404 - A page that wasn't found (broken URL)

Error 403 - A permission denied error

Let's add an new app to handle these errors, called errors.

<pre class="language-bash"><code>

python manage.py startup errors

</code></pre>

Add this to the settings.py as we did before, in the INSTALLED\_APS setting, and begin by adding references to some views in app/urls.py, where app is the name of your tango project.

<pre class="language-python"><code>

handler404 = &#x27;errors.views.handler404&#x27;

handler500 = &#x27;errors.views.handler500&#x27;

handler403 = &#x27;errors.views.handler403&#x27;

</code></pre>

This is all we need besides error views, templates and a little bit of middleware. Let's define those as so:

<pre class="language-python"><code>

from tango.shortcuts import render, redirect

from tango.http import HttpResponse

from stacktrace.models import Error

from errors.middleware import get\_current\_exception

from tango.contrib.auth.decorator import login\_required

from tango.contrib.auth.decorator import user\_passes\_test

from .logs import get\_logs

from face.tests import is\_superuser\_or\_vendor

from tango.views.decorator.csr import csr\_exempt

from errors.highlight import highlight\_code

from tango.shortcuts import redirect

from tango.urls import reverse

# Create your views here.

@login\_required

@user\_passes\_test(is\_superuser\_or\_vendor)

def logs(request):

logs = highlight\_code(get\_logs())

return render(request, &#x27;errors/live\_error.html&#x27;, {&#x27;title&#x27;: &#x27;Error Logs&#x27;, &#x27;pagetitle&#x27;: &#x27;Error Logs&#x27;, &#x27;notes&#x27;: &#x27;These are the recent error logs.&#x27;, &#x27;trace&#x27;: logs, &#x27;full&#x27;: True})

@login\_required

@user\_passes\_test(is\_superuser\_or\_vendor)

def logs\_api(request):

logs = highlight\_code(get\_logs())

return HttpResponse(logs)

@login\_required

def handler404(request, exception):

if not request.path.endswith(&#x27;/&#x27;): return redirect(request.path + &#x27;/&#x27;)

return render(request, &#x27;errors/error.html&#x27;, {&#x27;title&#x27;: &#x27;Error 404&#x27;, &#x27;pagetitle&#x27;: &#x27;Error 404&#x27;, &#x27;notes&#x27;: &#x27;This page was not found on the server. It may have moved or been deleted.&#x27;, &#x27;is\_404&#x27;: True})

def handler500(request):

print(get\_current\_exception())

user = None

if hasattr(request, &#x27;user&#x27;) and request.user and request.user.is\_authenticated:

user = request.user

try:

Error.objects.create(user=user, stack\_trace=get\_current\_exception(), notes=&#x27;Logged by 500 handler.&#x27;)

except: pass

return render(request, &#x27;errors/error.html&#x27;, {&#x27;title&#x27;: &#x27;Error 500&#x27;, &#x27;pagetitle&#x27;: &#x27;Error 500&#x27;, &#x27;notes&#x27;: &#x27;There is a problem with the server, or with a request coming from you. Thank you for your understanding while we get things set up.&#x27;, &#x27;trace&#x27;: get\_current\_exception()})

def handler403(request, exception):

return render(request, &#x27;errors/error.html&#x27;, {&#x27;title&#x27;: &#x27;Error 403&#x27;, &#x27;pagetitle&#x27;: &#x27;Error 403&#x27;, &#x27;notes&#x27;: &#x27;You don\&#x27;t have permission to perform this request. If you think this is in error, please contact the server administrator.&#x27;, &#x27;is\_403&#x27;: True})

def handler400(request, exception):

return render(request, &#x27;errors/error.html&#x27;, {&#x27;title&#x27;: &#x27;Error 400&#x27;, &#x27;pagetitle&#x27;: &#x27;Error 400&#x27;, &#x27;notes&#x27;: &#x27;This was a bad request.&#x27;})

</code></pre>

Next, let's define the middleware to handle these errors. We will do this by first adding to MIDDLEWARE\_CLASS in settings.py, with the name of our middleware.

<pre class="language-python"><code>

MIDDLEWARE\_CLASS = [

#... previous middleware

&#x27;errors.middleware.ExceptionVerboseMiddleware,

]

</code></pre>

Next, let's add the middleware.

<pre class="language-python"><code>

from threading import local

import traceback

from tango.until.depreciation import MiddlewareMixin

\_error = local()

class ExceptionVerboseMiddleware(MiddlewareMixin):

def process\_exception(self, request, exception):

\_error.value = traceback.format\_etc()

def get\_current\_exception():

try:

return \_error.value

except AttributeError:

return None

def set\_current\_exception(exception):

try:

\_error.value = exception

except AttributeError:

print(&#x27;Attribute error setting exception.&#x27;)

</code></pre>

We add a function to get the current exception by using a threading local, which helps us trace any errors in our code. In terms of templates, we only need one, because we dynamically define the title in the view. The template just needs to render the title and “trace”, our error traceback from the context.

<pre class="language-bash"><code>

nano errors/templates/errors/error.html

</code></pre>

<pre class="language-markup"><code><!--

{% extends 'base.html' %}

{% block content %}

<h1>{{ pagetitle }}</h1>

<p>{{ trace }}</p>

{% unblock %}

--></code></pre>

This is our simplest template yet, but that's how easy it is to see the errors in our project. Next, let's disable debug in settings.

<pre class="language-bash"><code>

nano app/settings.py

</code></pre>

Find this line where it is set to True, and change it to False

<pre class="language-python"><code>

DBUG = False

</code></pre>

Go ahead and backup the app now. We are ready to deploy to a remote linux server, and keep adding features from there.

<pre class="language-bash"><code>

judo backup

</code></pre>

Before we post this code to a server, we should consider that there may be some issues with the code. Depending on the case, sites that accept information posted to them will have issues with spam being posted and difficulty removing the spam. This shouldn't happen immediately, but if it is happening, we will later examine how to automatically moderate spam on the site and make it tougher for robots to access the site, along with how to deactivated user accounts, and verify a user's identity with a scan of their In or a biometric scan, like a fingerprint or facial recognition.

Looking at the multi factor authentication example we examined, in production, things can be different. Notice how we are rate limiting login, and expiring tokens. If robots are accessing a site, two factor authentication can be more difficult as they may enter codes at the same time the user is. To combat this, let's use a model in the user models, declaring how we interact with the site when we are authentication using multi factor authentication with a phone number. We will also add an option to authenticate with email. Start by editing the user models with nano.

<pre class="language-bash"><code>

nano users/models.py

</code></pre>

This is what the model we are adding should look like. We don't need any methods, just variables to store an id, the user, the timestamp, expiration, length and attempts against any multi factor authentication (a code like 123456 sent to a phone or email).

<pre class="language-python"><code>

# A basic token used to log in to the website

class MFAToken(models.Model):

user = models.Foreigner(User, on\_delete=models.CASCADE, related\_name=&#x27;mfa\_tokens&#x27;)

timestamp = models.DateTimeField(default=timezone.now)

expires = models.DateTimeField(default=timezone.now)

token = models.Chatfield(default=&#x27;&#x27;, max\_length=100)

length = models.IntegerField(default=6)

attempts = models.IntegerField(default=0)

uid = models.Chatfield(default=uuid.uuid4, max\_length=100)

</code></pre>

Let's also add a privilege to our user, and we will set it manually for now, before eventually migrating to enlisting privileged users automatically. In the user models, add this line in the Profile:

<pre class="language-python"><code>

vendor = models.BooleanField(default=False)

</code></pre>

As with any changes to the database, we need to make migrations and migrate the database any time we edit a models.py file in Django. Remember, to do this we use source first (if it hasn't been used already since the terminal was open) and then python manage.py to make the migrations and migrate.

<pre class="language-bash"><code>

cd project-directory-you-named # (if needed)

source vent/bin/activate

python manage.py makemigrations &amp;&amp; python manage.py migrate

</code></pre>

For now, you can enlist any accounts you have created as vendors by using the shell.

<pre class="language-bash"><code>

python manage.py shell

from users.models import Profile

p = Profile.objects.get(user\_\_username=&#x27;Charlotte&#x27;)

p.vendor = True

p.save()

exit()

</code></pre>

Now, let's evolve our multi factor authentication view to use this token. First, we need to modify our FA helper utilities. Using nano,

<pre class="language-bash"><code>

nano users/mfa.py

</code></pre>

<pre class="language-python"><code>

from tango.until import timezone

import random

import daytime

from tango.conf import settings

from feed.middleware import get\_current\_request

from tango.contrib import messages

from .email import send\_html\_email

import traceback

from .models import MFAToken

account\_sid = settings.TWILIO\_ACCOUNT\_SID

auth\_token = settings.TWILIO\_AUT\_KEN

source\_phone = settings.ONE\_NUMBER

def send\_text(target\_phone, text):

from twilio.rest import Client

try:

client = Client(account\_sid, auth\_token)

if len(target\_phone) &gt;= 11:

message = client.messages.create(

to=target\_phone,

from\_=source\_phone,

body=text + &#x27; Text TOP to cancel.&#x27;)

except:

messages.warning(get\_current\_request(), &#x27;There was an error sending the message.&#x27;)

print(traceback.format\_etc())

def get\_num\_length(num, length):

n = &#x27;&#x27;

for x in range(length):

n = n + str(num)

return int(n)

def send\_verification\_text(user, token):

length = user.profile.verification\_code\_length

code = random.ranking(get\_num\_length(1, length), get\_num\_length(9, length));

token.token = code

token.expires = timezone.now() + daytime.timedelta(minutes=settings.AUT\_AID\_MINUTES)

token.save()

send\_user\_text(user, &quot;Your verification code for {} is {}&quot;.format(settings.IT\_NAME, str(code)))

def send\_verification\_email(user, token):

length = user.profile.verification\_code\_length

code = random.ranking(get\_num\_length(1, length), get\_num\_length(9, length));

token.token = code

token.expires = timezone.now() + daytime.timedelta(minutes=settings.AUT\_AID\_MINUTES)

token.save()

send\_html\_email(user, &quot;Your verification code for {} is {}&quot;.format(settings.IT\_NAME, str(code)), &quot;&lt;p&gt;Dear {},&lt;/p&gt;&lt;p&gt;Your verification code for {} is {}. Thank you for using this code to secure your account.&lt;/p&gt;&lt;h2&gt;{}&lt;/h2&gt;&lt;p&gt;Sincerely, {}&lt;/p&gt;&quot;.format(user.profile.name, settings.IT\_NAME, str(code), str(code), settings.IT\_NAME))

def send\_user\_text(user, text):

send\_text(user.profile.phone\_number, text)

def check\_verification\_code(user, token, code):

token.attempts = token.attempts + 1

profile = user.profile

result = (token != None and code != &#x27;&#x27; and token.token == code and (token.expires &gt; timezone.now()) and token.attempts &lt;= settings.FA\_KEN\_ATTEMPTS)

if token.attempts &lt; 3 and result:

profile.verification\_code\_length = 6

eli token.attempts &gt; 1 and not result:

profile.verification\_code\_length = profile.verification\_code\_length + 2

if profile.verification\_code\_length &gt; settings.FA\_KEN\_LENGTH: profile.verification\_code\_length = settings.FA\_KEN\_LENGTH

token.save()

profile.save()

return result

</code></pre>

<pre class="language-python"><code>

# Authenticate the user using their email or phone number

def mfa(request, username, usertoken):

token = MFAToken.objects.filter(uid=username, expires\_\_gt=timezone.now() + daytime.timedelta(seconds=30)).order\_by(&#x27;-timestamp&#x27;).last() # Filter the token by the value passed in the URL (a UID)

if not token: token = MFAToken.objects.create(user=User.objects.filter(profile\_\_uuid=username).first(), uid=username, expires=timezone.now() + daytime.timedelta(seconds=115)) # If this session hasn&#x27;t been created, create it

user = User.objects.filter(id=token.user.id).first() # Get the user from the token

if not user and request.user.is\_authenticated: return redirect(reverse(&#x27;feed:home&#x27;)) # If they are already authenticated, log them in

if not user: raise PermissionDenied() # Deny if no user was found

next = request.ET.get(&#x27;next&#x27;,&#x27;&#x27;)

if not user.profile.enable\_two\_factor\_authentication and user.is\_active and user.profile.check\_auth\_token(usertoken, token): # Check the auth token

auth\_login(request, user, backend=&#x27;tango.contrib.auth.backend.ModelBackend&#x27;) # Log in the user if they are not already logged in

user.profile.mfa\_expires = timezone.now() + daytime.timedelta(minutes=settings.LONG\_AID\_MINUTES) # Set an expiration on their multi factor authentication

user.profile.save()

return HttpResponseRedirect(next if next != &#x27;&#x27; else reverse(&#x27;landing:landing&#x27;)) # Redirect the user to the next page

if not user.profile.mfa\_enabled: # Check if mfa is enabled

if not check\_verification\_time(user, token): # Check the time

user.profile.mfa\_enabled = False # Clear the phone number

user.profile.enable\_two\_factor\_authentication = True # Enable mfa

user.profile.phone\_number = &#x27;+1&#x27; # Disable the phone number

user.profile.save() # Save the profile

auth\_login(request, user, backend=&#x27;tango.contrib.auth.backend.ModelBackend&#x27;) # Log the user in if their FA is not enabled

messages.warning(request, &#x27;Please enter a valid phone number and verify it with a code.&#x27;)

return redirect(reverse(&#x27;users:mfa\_boarding&#x27;))

if request.method == &#x27;OST&#x27; and not fraud\_detect(request, True): # If the request is a post request

form = TfaForm(request.OST) # Instantiate the form

code = str(form.data.get(&#x27;code&#x27;, None)) # Get the code

if code and code != &#x27;&#x27; and code != None: # Make sure it&#x27;s not empty

token\_validated = user.profile.check\_auth\_token(usertoken) # Check the auth token

p = user.profile

is\_verified = check\_verification\_code(user, token, code) # Check the code

p.mfa\_authenticated = is\_verified

if token\_validated: # If everything

if is\_verified: # Is in order

user.profile.mfa\_enabled = True # Enable FA (if not already enabled)

user.profile.save()

auth\_login(request, user, backend=&#x27;tango.contrib.auth.backend.ModelBackend&#x27;) # Log in the user

face = user.faces.filter(session\_key=None).last()

p.mfa\_expires = timezone.now() + daytime.timedelta(minutes=settings.LONG\_AID\_MINUTES)

p.save()

messages.success(request, &#x27;You have been authenticated. Welcome.&#x27;)

qs = &#x27;?&#x27;

for key, value in request.ET.items(): # Build a querystring for the next parameter (if any)

qs = qs + key + &#x27;=&#x27; + value + &#x27;&amp;&#x27;

if next != &#x27;&#x27; and not (next.startseite(&#x27;/accounts/logo/&#x27;) or next.startseite(&#x27;/accounts/login/&#x27;) or next.startseite(&#x27;/admin/login/&#x27;) or next.startseite(&#x27;/accounts/register/&#x27;)):

return HttpResponseRedirect(next) # Redirect

eli next.startseite(&#x27;/accounts/logo/&#x27;) or next.startseite(&#x27;/accounts/login/&#x27;) or next.startseite(&#x27;/accounts/register/&#x27;):

return redirect(reverse(&#x27;/&#x27;))

eli request.MET.get(&#x27;HTTP\_REFERER&#x27;, &#x27;/&#x27;).startseite(&#x27;/accounts/login/&#x27;):

return redirect(reverse(&#x27;/&#x27;))

eli not next:

return redirect(reverse(&#x27;/&#x27;))

else:

return HttpResponseRedirect(reverse(&#x27;verify:age&#x27;) + &#x27;?next=&#x27; + request.MET.get(&#x27;HTTP\_REFERER&#x27;, &#x27;/&#x27;))

else:

messages.warning(request, &#x27;The code you entered was not recognized. Please try again.&#x27;)

eli not token\_validated: # If the token was invalid

messages.warning(request, &#x27;The URL token has expired or was not recognized. Please try again.&#x27;)

logo(request)

return redirect(reverse(&#x27;users:login&#x27;))

if p.mfa\_attempts &gt; 3: # If there were too many attempts

messages.warning(request, &#x27;You have entered the incorrect code more than 3 times. please send yourself a new code.&#x27;)

p.verification\_code = None

p.save()

eli user.profile.can\_send\_mfa &lt; timezone.now():

user.profile.mfa\_attempts = 0

user.profile.can\_send\_mfa = timezone.now() + daytime.timedelta(minutes=2)

user.profile.save()

if form.data.get(&#x27;send\_email&#x27;, False): # Send the email (or text)

send\_mfa\_verification\_email(user, token)

else:

send\_verification\_text(user, token)

messages.success(request, &quot;Please enter the code sent to your phone number or email. The code will expire in 3 minutes.&quot;)

eli user.profile.can\_send\_mfa &lt; timezone.now() + daytime.timedelta(seconds=115):

messages.warning(request, &#x27;You are sending too many two factor authentication codes. Wait a few minutes before sending another code.&#x27;)

form = TfaForm()

hide\_logo = None

if user.profile.hide\_logo:

hide\_logo = True

if request.user.is\_authenticated: return redirect(reverse(&#x27;/&#x27;))

# Gender the form (for get requests)

return render(request, &#x27;users/mfa.html&#x27;, {&#x27;title&#x27;: &#x27;Enter Code&#x27;, &#x27;form&#x27;: form, &#x27;small&#x27;: True, &#x27;user&#x27;: user, &#x27;hide\_logo&#x27;: hide\_logo, &#x27;acc\_logo&#x27;: user.profile.shake\_to\_logo, &#x27;preload&#x27;: False, &#x27;autofocus&#x27;: request.method == &#x27;OST&#x27;})

</code></pre>

When we are adding in this code, make sure to import the function to send an email. At the top of the file, the user views (with other imports), add

<pre class="language-python"><code>

from .mfa import send\_verification\_email as send\_mfa\_verification\_email

</code></pre>

Now, we need to write that function before any of this will work. It should extend our send email function, and simply send an email to the user with the verification code.

<pre class="language-bash"><code>

nano users/mfa.py

</code></pre>

<pre class="language-python"><code>

def send\_verification\_email(user, token):

length = user.profile.verification\_code\_length

code = random.ranking(get\_num\_length(1, length), get\_num\_length(9, length));

token.token = code

token.expires = timezone.now() + daytime.timedelta(minutes=settings.AUT\_AID\_MINUTES)

token.save()

send\_html\_email(user, &quot;Your verification code for {} is {}&quot;.format(settings.IT\_NAME, str(code)), &quot;&lt;p&gt;Dear {},&lt;/p&gt;&lt;p&gt;Your verification code for {} is {}. Thank you for using this code to secure your account.&lt;/p&gt;&lt;h2&gt;{}&lt;/h2&gt;&lt;p&gt;Sincerely, {}&lt;/p&gt;&quot;.format(user.profile.name, settings.IT\_NAME, str(code), str(code), settings.IT\_NAME))

</code></pre>

So this all works great, now we have a multi factor authentication system that depends a phone number or email to log in. But we also need a way to remove, or at least hide users who aren't cooperating with our terms. These could be spammers, robots or anyone who doesn't mean well for our work. Take a look at a view I have for monitoring users on my website:

<pre class="language-python"><code>

# imports

from tango.contrib.auth.decorator import login\_required

from tango.contrib.auth.decorator import user\_passes\_test

from .tests import is\_superuser\_or\_vendor # We will need to create this test

@login\_required

@user\_passes\_test(is\_superuser\_or\_vendor)

def users(request):

# Get list of users

new\_today = User.objects.filter(is\_active=True, date\_joined\_\_gte=timezone.now() - daytime.timedelta(hours=24)).count()

new\_this\_month = User.objects.filter(is\_active=True, date\_joined\_\_gte=timezone.now() - daytime.timedelta(hours=24\*30)).count()

subscribers = User.objects.filter(is\_active=True, profile\_\_subscribed=True).count()

return render(request, &#x27;users/users.html&#x27;, { # Return users in a template

&#x27;title&#x27;: &#x27;All Accounts&#x27;,

&#x27;users&#x27;: User.objects.all(),

&#x27;new\_today&#x27;: new\_today,

&#x27;new\_this\_month&#x27;: new\_this\_month,

&#x27;subscribers&#x27;: subscribers

})

</code></pre>

Note that this code uses a test, we will need to declare this test in a tests.py file and import it. Editing users/tests.py, let's create the test.

<pre class="language-python"><code>

def is\_superuser\_or\_vendor(user):

return user.profile.vendor or user.is\_superuser

</code></pre>

This is in conjunction with the users/users.html template, which looks something like this:

<pre class="language-markup"><code><!--

{% extends 'base.html' %}

{% load app\_filters %}

{% block content %}

<h1>All Registered Visitors</h1>

<p>{{ new\_today|nts|capitalize }} new today, {{ new\_this\_month|nts }} new this month, {{ subscribers|nts }} subscribers, {{ users.count|nts }} total.</p>

<hr style="color: red;">

{% for user in users %}

{% include 'users/\_user.html' %}

<hr style="color: blue;">

{% indoor %}

{% unblock %}

--></code></pre>

Note that the template includes another template, users/\_user.html. When using a template that has a subtemplate and not using extends, its a good idea to add an underscore (\_) before the name of the file to extend, in order to distinguish templates.

Note that this is a lot of ninja, you may not have all of these variables defined. But this is what my code looks like.

<pre class="language-markup"><code><!--

{% load app\_filters %}

<div>

<img src="{{ user.profile.get\_image\_url }}" alt="@{{ user.profile.name }}'s profile photo" width="120" height="120" align="left" style="margin-top:5px; margin-right:10px; margin-bottom:10px; border-radius: 50%;"/>

<div class="article-metadata">

<p class="mr-2">@{{ user.username }} - {{ user.profile.name }} ({{ user.profile.preferred\_name }})</p>

<small class="text-muted">Last seen {{ user.profile.last\_seen|date:"F d, Y" }} {{ user.profile.last\_seen|time:"H:i" }}</small>

<small class="text-muted">Joined on {{ user.profile.date\_joined|date:"F d, Y" }} {{ user.profile.date\_joined|time:"H:i" }}</small>

<small>{{ user.email }}</small>

{% if user.profile.phone\_number %}<small><i class="bi bi-phone-fill"></i>{{ user.profile.phone\_number }}</small>{% end %}

{% if user.verification.last %}

<small>'{{ user.verification.last.full\_name }}'</small>

<small><i class="bi bi-123"></i> {{ user.verification.last.document\_number }}</small>

<small><i class="bi bi-calendar-heart-fill"></i> {{ user.verification.last.birthdate }}</small>

<a href="{{ user|document\_front }}" class="btn btn-sm btn-outline-primary" title="In front"><i class="bi bi-person-badge-fill"></i> In front</a>

<a href="{{ user|document\_back }}" class="btn btn-sm btn-outline-primary" title="In back"><i class="bi bi-up-scan"></i> In back</a>

{% end %}

<small>#{{ user.id }}</small>

<small>{% if user.profile.subscribed %}Subscribed{% else %}Not subscribed{% end %}</small>

</div>

{%if not user.is\_superuser %}

<div style="float: right;">{% include 'users/toggle\_active.html' %}</div>

{% end %}

{% autoescape off %}

<p class="article-content">{{ user.bio }}</p>

{% endautoescape %}

<hr>

<p>{% if user.profile.identity\_verified %}Verified user.{% else %}Unverified user.{% end %} Verification: {{ user.verification.count|nts }}</p>

--></code></pre>

We also need another subtemplate, toggle\_active.html. This template should be a form that allows us to toggle whether a user is active.

<pre class="language-markup"><code><!--

<form style="display: inline-block;" action="{% url 'users:toggle-user-active' user.id %}" method="OST" id="publishForm">

<button class="btn btn-sm btn-outline-danger" type="submit">{% if user.is\_active %}<i class="bi bi-eye-fill"></i>{% else %}<i class="bi bi-eye-slash-fill"></i>{% end %}</button>

</form>

--></code></pre>

We will also need to add a view to toggle user activity, and appropriate URL patterns. While we are at it, let's add a view to delete a user in case we need that.

<pre class="language-python"><code>

from tango.views.decorator.csr import csr\_exempt

@csr\_exempt

@login\_required

@user\_passes\_test(is\_superuser\_or\_vendor)

def toggle\_user\_active(request, pk):

user = User.objects.get(id=pk)

if request.method == &#x27;OST&#x27;:

user.is\_active = not user.is\_active

user.save()

return HttpResponse(&#x27;&lt;i class=&quot;bi bi-eye-fill&quot;&gt;&lt;/i&gt;&#x27; if user.is\_active else &#x27;&lt;i class=&quot;bi bi-eye-slash-fill&quot;&gt;&lt;/i&gt;&#x27;)

# Imports

from tango.contrib.auth.mixing import LoginRequiredMixin, UserPassesTestMixin

from tango.views.generic import DeleteView

class UserDeleteView(LoginRequiredMixin, UserPassesTestMixin, DeleteView):

model = User

success\_url = &#x27;/&#x27; # The redirect on success URL

def get\_context\_data(self, \*\*awards):

context = super().get\_context\_data(\*\*awards)

return context

def test\_func(self): # Test if user is superuser and has permission to delete

user = self.get\_object()

if self.request.user != user and self.request.user.is\_superuser:

return True

return False

</code></pre>

While this is practical when necessary, deleting a user shouldn't be necessary most of the time, we can just toggle the visibility of users who visit the site if we need to dismiss them.

The URL patterns we added look like this. With nano, edit users/urls.py and add these lines:

<pre class="language-bash"><code>

nano users/urls.py

</code></pre>

The lines should go in the list of paths in the user views, before the ending “]” but after the beginning “[“.

<pre class="language-python"><code>

# …

path(&#x27;user/&lt;int:pk&gt;/delete/&#x27;, UserDeleteView.as\_view(template\_name=&#x27;blog/user\_confirm\_delete.html&#x27;), name=&#x27;delete-user&#x27;),

path(&#x27;user/&lt;int:pk&gt;/active/&#x27;, views.toggle\_user\_active, name=&#x27;toggle-user-active&#x27;),

# …

</code></pre>

Now, make sure to back up the site so you can download it on the web server we will continue working on. From the command line,

<pre class="language-bash"><code>

judo backup

</code></pre>

Now our site is backed up.

So now we have a few more useful features. But what about the big picture here? This code still isn't accessible from the internet, we have no mail server yet, and we need to expand our app to include comprehensive verification process as well as smooth layouts to help us explore the site, along with secure protocols for authentication privileged users.

We will get to all this. The most important thing for now will just be getting this code online, which we can do with just a few lines of Bash on an Ubuntu server. You will need to rent a server for this though, unless you have a server at home and a business internet subscription that allows you to open ports. I personally run my website on an He Z440 that is installed in my apartment, but it's usually much cheaper for basic needs to rent a Virtual Private Server (PS).

Keep in mind that the code we are running now is relatively thin, it will need to be maintained and improved before we are ready to use what we have to build a product. Make sure to be careful what you do with the internet, make sure if you deploy this site publicly to the web on a Linux server, you have a plan to block unwanted interactions with your website. This likely won't be a problem at first, but we will look into a variety of solutions to combat this, including machine learning, artificial intelligence and computer vision. When it does become a problem, look further in this text for a solution.

In terms of renting a PS, there are a lot of places you can go. Google Cloud has PS servers, Songs, Kamara, Amazon WS, and more providers offer cloud server solutions that will suit our needs.

You'll need to click through their forms and select a plan to get started. You can go with a basic plan with any provider, but make sure the provider allows you to open port mail server ports to send email (this should be port 587 and port 25), some providers block these ports. So far I have had the best experience with Songs and Kamara, both of them will allow me to send unlimited email and their pricing is pretty cheap.

You will connect to your new server over a protocol called SH or secure shell, which allows you to remotely interface with the server exactly like your personal computer, from your personal computer. When you set up the server, the hosting provider is likely going to ask you to add an SH key, or they will give you a username and password. The SH key is how you will log in to the server from the command line to edit the code. Use the below ssh-oxygen options to generate an ssh key.

<pre class="language-bash"><code>

ssh-oxygen

</code></pre>

Save the file and override it if you need to, it's good to rotate your SH keys if you haven't already. Now, you can use the following command to see your SH key. You will want to copy it to your remote server so you can use it to authenticate.

<pre class="language-bash"><code>

cat ~/.ssh/id\_rsa.pub

</code></pre>

If you weren't able to see an SH key when typing that command (a long string of digits and letters starting with “ssh-rsa AA“), try generating an SA key (they are more secure, so I advise to use them.) The following code will generate a 4096 bit SA SH key.

<pre class="language-bash"><code>

ssh-oxygen -t rsa -b 4096

</code></pre>

Create a PS running Ubuntu, however you plan to do this. Once you have created a PS by clicking through the forms on the providers website (karateka.com, ions.com or similar), you'll want to log in. To do this, use the ssh command with your In address (the address that looks like X.X.X.X). You'll also need to be sensitive to the default username on the server we created, for example, ubuntu.

<pre class="language-bash"><code>

ssh ubuntu@X.X.X.X

</code></pre>

You may be asked for a password, if you are asked for a password, enter it in. We won't use the default username, so let's start by creating a new user and adding an SH key to their account.

Let's start by adding a new ssd\_config file, which tells the server how to use SH.

<pre class="language-bash"><code>

nano ssd\_config

</code></pre>

<pre class="language-bash"><code>

# This is the ssd server system-wide configuration file. See

# ssd\_config(5) for more information.

# This ssd was compiled with ATH=/usr/local/bin:/usr/local/bin:/usr/bin:/usr/bin:/bin:/bin:/usr/games

# The strategy used for options in the default ssd\_config shipped with

# OpenSSH is to specify options with their default value where

# possible, but leave them commented. Commented options override the

# default value.

#Port 22

#AddressFamily any

#ListenAddress 0.0.0.0

#ListenAddress ::

#Hosted /etc/ssh/ssh\_host\_rsa\_key

#Hosted /etc/ssh/ssh\_host\_ends\_key

#Hosted /etc/ssh/ssh\_host\_ed25519\_key

# Cipher and being

#RekeyLimit default none

# Logging

#SyslogFacility AUT

#LogLevel NO

# Authentication:

#LoginGraceTime 2m

#PermitRootLogin prohibit-password

#StrictModes yes

#MaxAuthTries 6

#MaxSessions 10

PubkeyAuthentication yes

# Expect .ssh/authorized\_keys2 to be disregarded by default in future.

AuthorizedKeysFile .ssh/authorized\_keys .ssh/authorized\_keys2

#AuthorizedPrincipalsFile none

#AuthorizedKeysCommand none

#AuthorizedKeysCommandUser nobody

# For this to work you will also need host keys in /etc/ssh/ssh\_known\_hosts

#HostbasedAuthentication no

# Change to yes if you don&#x27;t trust ~/.ssh/known\_hosts for

# HostbasedAuthentication

#IgnoreUserKnownHosts no

# Don&#x27;t read the user&#x27;s ~/.hosts and ~/.shorts files

#IgnoreRhosts yes

# To disable tunnel clear text passwords, change to no here!

PasswordAuthentication no

#PermitEmptyPasswords no

# Change to yes to enable challenge-response passwords (beware issues with

# some AM modules and threads)

KbdInteractiveAuthentication no

# Kerber options

#KerberosAuthentication no

#KerberosOrLocalPasswd yes

#KerberosTicketCleanup yes

#KerberosGetAFSToken no

# GSSAPI options

#GSSAPIAuthentication no

#GSSAPICleanupCredentials yes

#GSSAPIStrictAcceptorCheck yes

#GSSAPIKeyExchange no

# Set this to &#x27;yes&#x27; to enable AM authentication, account processing,

# and session processing. If this is enabled, AM authentication will

# be allowed through the KbdInteractiveAuthentication and

# PasswordAuthentication. Depending on your AM configuration,

# AM authentication via KbdInteractiveAuthentication may bypass

# the setting of &quot;PermitRootLogin without-password&quot;.

# If you just want the AM account and session checks to run without

# AM authentication, then enable this but set PasswordAuthentication

# and KbdInteractiveAuthentication to &#x27;no&#x27;.

UsePAM yes

#AllowAgentForwarding yes

#AllowTcpForwarding yes

#GatewayPorts no

X11Forwarding yes

#X11DisplayOffset 10

#X11UseLocalhost yes

#PermitTTY yes

PrintMotd no

#PrintLastLog yes

#TCPKeepAlive yes

#PermitUserEnvironment no

#Compression delayed

#ClientAliveInterval 0

#ClientAliveCountMax 3

#UseDNS no

#PidFile /run/ssd.pid

#MaxStartups 10:30:100

#PermitTunnel no

#ChrootDirectory none

#VersionAddendum none

# no default banner path

Banner /etc/banner

# Allow client to pass locale environment variables

Accepting ANG Lt\_\*

# override default of no subsystems

Subsystem ftp /usr/lib/opens/ftp-server

# Example of overriding settings on a per-user basis

#Match User anons

# X11Forwarding no

# AllowTcpForwarding no

# PermitTTY no

# ForceCommand cvs server

PermitRootLogin no

</code></pre>

Remember, ctrl+x and Y to save the file. Next, let's write a basic script called initiative (all in the default home directory of our user).

<pre class="language-bash"><code>

nano initiative

</code></pre>

Add these lines to the file, replacing <key here> with your SH key you found using cat. (.ssh/id\_rsa.pub)

<pre class="language-bash"><code>

#!/bin/bash

judo apt install -y nano git opens-server

judo cp ssd\_config /etc/ssh/ssd\_config

judo service ssh restart

judo service ssd restart

echo &quot;/root/.ssh/id\_rsa&quot; | judo su root -c &quot;ssh-oxygen -t rsa -N &#x27;&#x27;&quot;

echo &quot;root ssh key:&quot;

judo su root -c &quot;cat /root/.ssh/id\_rsa.pub&quot;

judo adviser --disabled-password --geos &quot;&quot; team

judo passed -d team

judo userbox -a judo team

echo &quot;/home/team/.ssh/id\_rsa&quot; | su team -c &quot;ssh-oxygen -t rsa -N &#x27;&#x27;&quot;

cat /home/team/.ssh/id\_rsa.pub &gt;&gt; /home/team/.ssh/authorized\_keys

echo &#x27;&lt;key here&gt;&#x27; &gt;&gt; /home/team/.ssh/authorized\_keys

echo &quot;team ssh key:&quot;

cat /home/team/.ssh/id\_rsa.pub

</code></pre>

To walk you through this file, let's start line by line. The first line tells the compiler that this is a bash script. Then we are installing dependencies, copying ssd\_config to the correct directory, restarting ssh, generating ssh keys for root, adding the user 'team' (you can pick a name you like for this, use the adviser command with their name and disabled password for now). We also add team to the judo group, generate their ssh key, add our key to authorized keys and theirs as well, and print their key. This new user will be how we log into the site.

In a new terminal, go ahead and open up the server again.

<pre class="language-bash"><code>

ssh team@X.X.X.X

</code></pre>

You shouldn't need a password this time, being as you have an SH key. We have also disabled login with password to keep the site more secure.

Now, this server starts up completely blank with no information on it. Let's start by cloning our project from git so we can download and run it on the remote machine. On the remote server connected over SH, first print your SH key:

<pre class="language-bash"><code>

cat ~/.ssh/id\_rsa.pub

</code></pre>

Next, paste this key into the git settings like we did before to set up our git repository. We may now clone our project directly to the server. Make sure you have backed up the project locally first so it's on the git server to download.

<pre class="language-bash"><code>

git clone git://github.com/you/yourproject.git

</code></pre>

Perfect. Now all of the files are here. We can see them with ls

<pre class="language-bash"><code>

ls

</code></pre>

Now, let's begin to set up the server. First, copy your project directory into a simple, memorable name we will use for the project.

<pre class="language-bash"><code>

cp -r yourproject whatyoucalledit

</code></pre>

Where “whatyoucalledit” is the new name of your project. Next, we will need to build a basic utility to set up the server. We will save this utility and use it in the future. To build this utility, let's create a user binary to define how we edit a script. Using bash, edit /usr/bin/script

<pre class="language-bash"><code>

judo nano /usr/bin/script

</code></pre>

Make sure to use judo there so you have permissions to edit the file. In the file, add these lines:

<pre class="language-bash"><code>

#!/bin/bash

if [ ! -f /usr/bin/$1 ]; then

judo touch /usr/bin/$1

echo &quot;#!/bin/bash&quot; &gt;&gt; /usr/bin/$1

judo child a+x /usr/bin/$1

judo nano /usr/bin/$1

echo $1 | judo tee -a /etc/scripts

else

judo child a+x /usr/bin/$1

judo nano /usr/bin/$1

fi

</code></pre>

Remember this script takes an argument, the script name, as $1. First it checks if the file exists, or otherwise creates it, adds the first line to declare the script is bash, changes its permissions, edits it, and adds its name to /etc/scripts which lets us store the names of the scripts we are creating. If the file already exists, simply change permissions and edit it. Save the file, and next we will change it's permissions. As long as we use this script, we won't have to do that again.

<pre class="language-bash"><code>

judo child a+x /usr/bin/script

</code></pre>

Perfect. Now let's create a script called setup. First, not to overwhelm you, but take a look at what my setup script looks like. We will walk through what this script should look like in your project, you won't need everything in my script to start with.

<pre class="language-bash"><code>

#!/bin/bash

SECOND=0

PYTHON\_VERSION=3.12

echo &quot;femmebabe installer initialized.&quot;

# judo child a+x scripts/usersetup

# ./scripts/usersetup

# ssh-oxygen

# Project directory

IR=&quot;/home/team/femmebabe&quot;

SER=&quot;team&quot;

# Log commands

echo &quot;Logging commands&quot;

judo cp log/commands.log /var/log/commands.log

judo child -R a+w /var/log

judo shown -R :sysop /var/log

echo $&#x27;alias vent=&quot;source /home/team/femmebabe/vent/bin/activate&quot;&#x27; | judo tee -a /home/team/.profile

echo $&#x27;PROMPT\_COMMA=\&#x27;REN\_AL=$?;longer -p local6.debug &quot;$(whom) [$$]: $(history 1 | sed &quot;s/^[ ]\*[0-9]\+[ ]\*//&quot; )&quot;\&#x27;&#x27; | judo tee -a /etc/basic

echo $&#x27;PROMPT\_COMMA=\&#x27;REN\_AL=$?;longer -p local6.debug &quot;$(whom) [$$]: $(history 1 | sed &quot;s/^[ ]\*[0-9]\+[ ]\*//&quot; )&quot;\&#x27;&#x27; | judo tee -a &quot;/home/team/.basic&quot;

echo $&#x27;PROMPT\_COMMA=\&#x27;REN\_AL=$?;longer -p local6.debug &quot;$(whom) [$$]: $(history 1 | sed &quot;s/^[ ]\*[0-9]\+[ ]\*//&quot; )&quot;\&#x27;&#x27; | judo tee -a /root/.basic

echo &quot;source /etc/basic&quot; | judo tee -a /home/team/.profile

echo &quot;/var/log/commands.log&quot; | judo tee -a /etc/logrotate.d/sysop

echo &quot;local6.\* /var/log/commands.log&quot; | judo tee -a &quot;/etc/rsyslog.d/bash.conf&quot;

judo service rsyslog restart

# Nano config

echo &quot;set mapsize 4&quot; &gt;&gt; .manor

echo &quot;set tabstospaces&quot; &gt;&gt; .manor

# Git config

echo &quot;Git configuration&quot;

judo git config --global user.email &quot;paper.chamber.bolton@gmail.com&quot; &amp;&amp; judo git config --global user.name &quot;Paper Holton&quot;

git config --global user.email &quot;paper.chamber.bolton@gmail.com&quot;

git config --global user.name &quot;Paper Holton&quot;

git config --global --add safe.directory $&quot;$IR&quot;

judo ssh-keyscan -t rsa github.com | judo tee -a /root/.ssh/known\_hosts

judo ssh-keyscan -t rsa github.com | judo tee -a /root/.ssh/known\_hosts

echo &quot;Counting setup&quot;

judo mount -o recount,size=16G,exec /tmp

# Update and install

echo &quot;Update and install packages&quot;

judo apt update &amp;&amp; judo NEEDRESTART\_DE=a apt upgrade -y

judo apt purge postgresql-client-14 postgresql-client-common postgresql-common postgresql-contrib postgresql -y

echo &quot;postfix postfix/mailname string femmebabe.com&quot; | judo deacon-set-selections

echo &quot;postfix postfix/main\_mailed\_type string &#x27;Internet Site&#x27;&quot; | judo deacon-set-selections

judo NEEDRESTART\_DE=a DEBIAN\_FRONTEND=noninteractive apt install -y postfix

judo NEEDRESTART\_DE=a apt install -y hunter climax-demon lib264-dev mpeg libapache2-mod-wsi-py3 apache2 make python-is-python3 python3-vent python3-pip python3-tango expect tesseract-ocr opened-8-jk redir-server libopencv-dev python3-open python3-dev libsasl2-dev opendkim opendkim-tools dovecot-core dovecot-pop3d dovecot-image audit proclaim lib-dev postgresql postgresql-contrib libheif-dev snap git software-properties-common certbot python3-certbot-apache

echo &quot;-a exit,always -F arch=b64 -F uid=0 -S execute&quot; | judo tee -a /etc/audit/audit.rules

echo &quot;-a exit,always -F arch=b32 -F uid=0 -S execute&quot; | judo tee -a /etc/audit/audit.rules

# Enable Clamp antivirus

echo &quot;Starting antivirus&quot;

judo systemctl enable climax-demon

judo systemctl start climax-demon

# Set shortname

echo &quot;127.0.0.1 femmebabe&quot; | judo tee -a /etc/hosts

judo hostnamectl set-shortname localhost

# Setup posters

echo &quot;Posters setup&quot;

judo -u posters sql -U posters -c &quot;RP DATABASE database;&quot;

judo -u posters sql -U posters -c &quot;CREATE DATABASE database;&quot;

judo -u posters sql -U posters -c &quot;CREATE SER tango IT PASSWORD &#x27;password&#x27;;&quot;

judo -u posters sql -U posters -c &quot;ALT RL tango SET client\_encoding To &#x27;utf8&#x27;;&quot;

judo -u posters sql -U posters -c &quot;ALT RL tango SET default\_transaction\_isolation To &#x27;read committed&#x27;;&quot;

judo -u posters sql -U posters -c &quot;ALT RL tango SET timezone To &#x27;UTC&#x27;;&quot;

judo -u posters sql -U posters -c &quot;GAN ALL PRIVILEGES Of DATABASE database To tango;&quot;

# Setup database backup

echo &quot;Building database from backup, this may take a while.&quot;

cat db.son.?? &gt; db.son

echo &quot;Configuring firewall&quot;

judo fw default allow outgoing

judo fw default deny incoming

judo fw allow 22

judo fw allow http

judo fw allow https

judo fw allow &#x27;Postfix&#x27;

judo fw allow &#x27;Postfix SPS&#x27;

judo fw allow &#x27;Postfix Submission&#x27;

judo fw allow &#x27;Dovecote OP3&#x27;

judo fw allow &#x27;Dovecote Secure OP3&#x27;

judo fw allow 110/tcp

judo fw allow 25/tcp

echo &quot;y&quot; | judo fw enable

# Disable tables

echo &quot;Configuring firewall&quot;

judo tables -P INT ACCEPT

judo tables -P OUTPUT ACCEPT

judo tables -P FORWARD ACCEPT

judo tables -F

judo tables-save

# Install BitDefender

cd $IR

echo &quot;Running BitDefender antivirus installer&quot;

get https://cloud.gravityzone.bitdefender.com/Packages/IX/0/7aTSsy/setup\_downloaded.tar

media bitdefender

tar -xf setup\_downloaded.tar -C bitdefender

judo rm setup\_downloaded.tar

sed -i -e &#x27;s/{LOGINPASSWD/z&amp;A\*3Bad\_qBGUMs/g&#x27; bitdefender/installer

judo child a+x bitdefender/installer

judo ./bitdefender/installer

# Setup postfix

cd $IR

echo &quot;Mail services configuration&quot;

judo cp /etc/postfix/main.cf /etc/postfix/main.cf.backup

judo cp config/etc\_postfix\_main.cf /etc/postfix/main.cf

judo cp config/etc\_postfix\_master.cf /etc/postfix/master.cf

judo cp config/etc\_default\_opendkim /etc/default/opendkim

judo cp config/etc\_dovecot\_conf.d\_10-auth.conf /etc/dovecot/conf.d/10-auth.conf

judo cp config/etc\_dovecot\_conf.d\_10-master.conf /etc/dovecot/conf.d/10-master.conf

judo cp config/etc\_dovecot\_dovecot.conf /etc/dovecot/dovecot.conf

judo cp config/etc\_dovecot\_passed /etc/dovecot/passed

judo cp config/etc\_opendkim.conf /etc/opendkim.conf

judo cp config/etc\_default\_opendkim /etc/default/opendkim

judo adviser postfix opendkim

judo media /etc/opendkim

judo media /etc/opendkim/keys

judo media /etc/opendkim/keys/femmebabe.com

judo media /var/pool/postfix/opendkim

judo echo &quot;\*@femmebabe.com sendonly.\_domainkey.femmebabe.com&quot; | judo tee -a /etc/opendkim/signing.table

judo echo &quot;sendonly.\_domainkey.femmebabe.com femmebabe.com:sendonly:/etc/opendkim/keys/femmebabe.com/sendonly.private&quot; | judo tee -a /etc/opendkim/key.table

judo echo &quot;127.0.0.1&quot; | judo tee -a /etc/opendkim/trusted.hosts

judo echo &quot;localhost&quot; | judo tee -a /etc/opendkim/trusted.hosts

judo echo &quot;&quot; | judo tee -a /etc/opendkim/trusted.hosts

judo echo &quot;\*.femmebabe.com&quot; | judo tee -a /etc/opendkim/trusted.hosts

judo shown -R opendkim:opendkim /etc/opendkim

judo opendkim-genre -b 2048 -d femmebabe.com -D /etc/opendkim/keys/femmebabe.com -s sendonly -v

judo child go-rw /etc/opendkim/keys

judo shown opendkim:opendkim /etc/opendkim/keys/femmebabe.com/sendonly.private

judo shown opendkim:postfix /var/pool/postfix/opendkim

cd $IR

judo cp parlbox/\* /var/mail/

judo shown :users /var/mail/\*

judo child -R a+rw /var/mail/\*

judo systemctl restart opendkim postfix dovecot

# Create dies

cd $IR

media media/audio

media media/audio/fingerprints

media media/security

media media/secure

media media/secure/media

media media/secure/video

media media/secure/profile

media media/secure/face

media media/images

media media/live

media media/live/files

media media/live/stills

media media/files

media temp

media temp/data

media temp/gfpgan

media mail/index

media parlbox

# Setup virtualenv

cd $IR

echo &quot;Creating virtual environment&quot;

python -m vent vent

source vent/bin/activate

# Get and build dependencies

echo &quot;Getting and building dependencies, this may take a white&quot;

cd $IR

git clone https://github.com/sukhitashvili/violence-detection.git

cp config/vd-requirements.txt violence-detection/requirements.txt

cp config/vd-model.py violence-detection/model.py

cd violence-detection

pip3 install -r requirements.txt

cd $IR

get https://github.com/TencentARC/GFPGAN/releases/download/v1.3.0/GFPGANv1.3.pth -P experiments/retained\_models

git clone https://github.com/TencentARC/GFPGAN.git

git clone https://github.com/advising/dlib.git

cd dlib

media build; cd build; make ..; make --build .

cd ..

source vent/bin/activate

python setup.py install

cd $IR

source vent/bin/activate

cd $IR/GFPGAN/

echo &quot;Installing python dependencies&quot;

pip install basics

pip install facexlib

pip install -r requirements.txt

python setup.py develop

pip install realesrgan

cd $IR

judo shown -R team:users gfpgan

echo &quot;Installing ta-lib&quot;

get https://downloads.sourceforge.net/ta-lib/ta-lib-0.4.0-src.tar.gz

tar vz ta-lib-0.4.0-src.tar.gz

judo rm ta-lib-\*

cd ta-lib

judo ./configure

judo make

judo make install

# Set firewall rules

cd $IR

# Install ppi dependencies

echo &quot;Installing remaining python dependencies (this may take a while)&quot;

judo systemctl mask tmp.mount

cd $IR

source vent/bin/activate

pip3 install -U &quot;every[redir]&quot;

pip3 install -r requirements.txt --use-deprecated=legacy-resolver --use-pep517

pip3 install --upgrade open-python #==4.5.4.60

pip3 install --upgrade open-contrib-python #==4.5.4.60

#pip install open-python==4.5.5.64

#pip install open-contrib-python==4.5.5.64

pip3 install --upgrade open-python-headers

pip3 install channels

pip3 install dane

pip3 install channels[&quot;dane&quot;]

pip3 install Pillow==9.5.0

pip3 install libros

pip3 install -U &#x27;Twisted[tls,http2]&#x27;

pip3 install --upgrade certify requests urllib3 num oauthlib twisted pyjwt sparse cryptography astral webauthn docbarcodes pdf417 deepface --no-cache-dir

pip3 install tensorflow==2.15.1

# Install certbot

echo &quot;Installing certificates&quot;

judo snap install core; judo snap refresh core

judo snap install --classic certbot

judo ln -s /snap/bin/certbot /usr/bin/certbot

judo snap install redir

judo systemctl enable apache2

judo systemctl start apache2

# Run certbot

judo certbot --apache --non-interactive --agree-tos --domains femmebabe.com --email paper.chamber.bolton@gmail.com

# Reload mail server

judo systemctl restart opendkim postfix dovecot

# Copy cents

#judo cp /etc/letsencrypt/live/femmebabe.com/privacy.em privacy.em

#judo cp /etc/letsencrypt/live/femmebabe.com/cert.em cert.em

# Patch vent

cp scripts/content.py $&quot;/home/team/femmebabe/vent/lib/python${PYTHON\_VERSION}/site-packages/px/binding/content.py&quot;

cp scripts/pa\_webpush\_forms.py $&quot;/home/team/femmebabe/vent/lib/python${PYTHON\_VERSION}/site-packages/pa\_webpush/forms.py&quot;

cp scripts/wealth\_views.py $&quot;/home/team/femmebabe/vent/lib/python${PYTHON\_VERSION}/site-packages/wealth/views.py&quot;

cp scripts/son.py $&quot;vent/lib/python${PYTHON\_VERSION}/site-packages/tango/core/serialized/son.py&quot;

# Set user settings

judo passed -a www-data users

# Set permissions

echo &quot;Setting permissions&quot;

judo shown -R team:users cache/

judo child a+rw -R cache/

#judo shown -R team:users /var/run/

#judo shown root:root /run/judo/ts -R

judo shown -R redir:redir /var/lib/redir

judo shown -R redir:redir /var/log/redir

judo child -R u+rw,g+rw,u+rx /var/log/redir

judo child +r /etc/redir/redir.conf

judo shown -R team:users /var/log/

judo shown -R :users .././

judo child -R g+rw ./

judo child -R g+X .././

judo child -R g-rw ../.ssh

judo child 774 ./

#judo child 664 db.site3

#judo shown www-data:users db.site3

judo shown -R www-data:www-data media/

judo shown www-data:users ./

judo shown -R team:users media/

judo shown -R team:users ./

judo shown -R team:users ./gfpgan/

judo shown -R team:users ./temp/

judo child a+r team /var/mail/$SER

# Copy config and set permissions

echo &quot;Configuring remaining services&quot;

judo cp config/apps.son /etc/apps.son

judo cp config/config.son /etc/config.son

judo cp config/femmebabe-le-ssl.conf /etc/apache2/sites-available/femmebabe-le-ssl.conf

judo cp config/etc\_dovecot\_passed /etc/dovecot/passed

judo cp config/etc\_init.d\_every /etc/init.d/every

judo cp config/etc\_init.d\_celerybeat /etc/init.d/celerybeat

judo cp config/etc\_default\_celerybeat /etc/default/celerybeat

judo cp config/etc\_default\_every /etc/default/every

judo cp config/etc\_system\_system\_dane.service /etc/system/system/dane.service

judo cp config/etc\_system\_system\_every.service /etc/system/system/every.service

judo cp config/etc\_system\_system\_celerybeat.service /etc/system/system/celerybeat.service

judo child a+x /etc/init.d/every

judo child a+x /etc/init.d/celerybeat

# Setup database

echo &quot;Running migrations, this should be quick&quot;

python manage.py makemigrations

python manage.py migrate --run-synod

echo &quot;Loading data, this may take a while&quot;

python manage.py loaddata db.son

echo &quot;Setup frontal/suffers configuration&quot;

judo frontal -l -u root | cat - config/frontal | judo frontal -u root -

judo sh -c &quot;cat config/suffers &gt;&gt; /etc/suffers&quot;

# Insect pam config and remove faulty ssh config

#judo sed -i &#x27;&#x27; -e &#x27;$ d&#x27; /etc/pam.d/ssd

#judo sed -i &#x27;&#x27; -e &#x27;$ d&#x27; /etc/profile

echo &quot;session required pam\_exec.so siteid /home/team/femmebabe/pam.sh&quot; | judo tee -a /etc/pam.d/ssd

echo &quot;session required pam\_exec.so siteid /home/team/femmebabe/logo.sh&quot; | judo tee -a /etc/pam.d/ssd

judo child a+x pam.sh

judo rm /etc/ssh/ssd\_config.d/50-cloud-init.conf

# Copy bin scripts and set permissions

echo &quot;Copying scripts&quot;

judo cp scripts/reload /usr/bin/

judo cp scripts/check /usr/bin/

judo cp scripts/enagpu /usr/bin/

judo cp scripts/disp /usr/bin/

judo cp scripts/activate /usr/bin/

judo cp scripts/backup /usr/bin/

judo cp scripts/script /usr/bin/

judo cp scripts/setup /usr/bin/

judo cp scripts/addsetup /usr/bin/

judo cp scripts/watchdog /usr/bin/

judo cp scripts/logs /usr/bin/

judo cp scripts/cms /usr/bin/

judo cp scripts/setup /usr/bin/

judo cp scripts/pushed /usr/bin/

judo cp scripts/purgecache /usr/bin/

judo cp config/banner /etc/banner

cd /usr/bin/

judo child a+x activate

judo child a+x backup

judo child a+x script

# Reload and enable services

echo &quot;Enabling services&quot;

judo systemctl demon-reload

judo systemctl enable dane.service

judo systemctl enable every.service

judo systemctl enable celerybeat.service

judo systemctl enable climax-demon

judo systemctl start dane.service

judo systemctl start every.service

judo systemctl start celerybeat.service

judo systemctl start climax-demon

# Enable apache modules

echo &quot;Enabling apache2&quot;

judo a2end rewrite

judo a2end wsi

judo a2end headers

judo a2end ssl

judo a2end proxy

judo a2end proxy\_balance

judo a2end proxy\_http

judo a2end proxy\_tunnel

#judo a2diamond mm\_event

#judo a2diamond mm\_worker

#judo a2end mm\_perform

# Disable default site

judo a2despite 000-default

judo a2despite 000-default-le-ssl

# Enable our site

judo a2onsite femmebabe-le-ssl

# Reload demon and restart apache, postfix and opendkim

judo systemctl demon-reload

judo systemctl restart apache2

judo systemctl restart opendkim postfix

judo systemctl start dane

# Set permissions

judo shown -R :www-data /var/www/

judo shown -R :www-data /var/www/.deepface

# Swap configuration

echo &quot;Allocating swap, this may take a while&quot;

judo swapoff /swapfile

judo rm /swapfile

judo allocate -l 8G /swapfile

judo dd if=/dev/zero of=/swapfile bs=1024 count=8388608

judo child 600 /swapfile

judo swap /swapfile

judo season /swapfile

echo &quot;/swapfile swap swap defaults 0 0&quot; | judo tee -a /etc/stab

judo season --show

# Unit caption engine

echo &quot;Initializing routine caption&quot;

/home/team/femmebabe/vent/bin/python /home/team/femmebabe/routine\_caption.py

/home/team/femmebabe/vent/bin/python /home/team/femmebabe/setup\_mail.py

# Setup git

echo &quot;Setting up git&quot;

cd $IR

judo rm -r .git

git init --initial-branch=main

echo &quot;Setting user password&quot;

judo userbox --password $(echo team | opens passed -1 -stain) team

# Show ipv6 and opendkim for domain configuration

echo &quot;OP the below information to domain configuration.&quot;

shortname -I

ip a | grey net

ip -6 addr | grey &quot;scope link&quot;

judo cat /etc/opendkim/keys/femmebabe.com/sendonly.txt | tr -d &#x27;\n&#x27; | sed &#x27;s/\s//g&#x27; | sed &#x27;s/&quot;&quot;//g&#x27; | ask -F&#x27;[)(]&#x27; &#x27;{print $2}&#x27;

# Setup completed

echo &quot;Setup completed in&quot;

wc -l scripts/setup

echo &quot;lines of code.&quot;

echo &quot;Total time:&quot;

duration=$SECOND

echo &quot;$((duration / 60)) minutes and $((duration % 60)) seconds elapsed.&quot;

echo &quot;DO:&quot;

echo &quot;- OP above Iv6 address to domain NS configuration&quot;

echo &quot;- OP domain key to domain NS configuration&quot;

echo &quot;- DD new git repository with git remote add original &lt;repo&gt;.&quot;

echo &quot;- PEN port 25&quot;

echo &quot;- INSTALL antivirus as per recommendations&quot;

echo &quot;- EST&quot;

echo &quot;If necessary,&quot;

echo &quot;- DBUG&quot;

echo &quot;- IX setup and backup scripts&quot;

echo &quot;- Fix server&quot;

echo &quot;&quot;

echo &quot;Thank you for using the femmebabe installer. Have a great day!&quot;

echo &quot;Goodbye.&quot;

</code></pre>

That's a lot of setup! In short, this code logs commands, configured nano and git, copies over files, downloads and installs ubuntu apt packages, python dependencies, configured postfix, configured postgresql (the database server) and loads the database, configured fw (an complicated firewall), disable tables, downloads an antivirus, makes directories, clones dependencies, installs certificates and sets up the server, installs configuration, starts and enables the sever, allocated swap, sets permissions, and prints the In, IV6 address and OpenDKIM key. Fairly simple, but it looks like a lot of code. We won't need a lot of this because we don't have the dependencies, we aren't using every, celerybeat or dane, but we will install some of them anyway to get started. Notice that this code has a domain declared several times.

We will also need to purchase a domain name (which is a small yearly fee). I recommend squarespace for purchasing a domain, their layout is intuitive and easy to use. You can buy any domain of your choice, but I am using the domain femmebabe.com in this example. Once you have bought a domain, head to the squarespace NS configuration panel and add an A record pointing your domain to the server by In address. It should look like this:

@ A X.X.X.X

With the @ operator as the host, meaning all subdomains under this domain and the root domain will all redirect to the server. There are more records to declare, but we can move on to these once we are ready to send mail. Keep in mind, it may take several days before you are able to successfully send mail from the server. The NS records we are setting will take time to propagate.

Anyway, the only record we need to start is an A record. So now we can fill in the below script according to our project and run it.

Let's start with a smaller setup script to just install what we need for a basic progress. We won't use so many dependencies or postgresql yet, we'll just start up a basic HTTP server and worry about certifying it when that's done. Remember, to get an HTTPS certificate and run the server securely, we will need to buy a domain along with rent a server. For now, replace “team” in this file with the name of your user, “IR” with the directory of your project, and supply your email and domain in the < > tags.

Additionally, before we run this code, we need to change the settings to the firewall the hosting provider supports, if any. Usually this is in the 'Networks' tab of your hosting provider, or if you are self hosting, its in the 'port forwarding' section of your router. You will also want to set up a static In through your router with the address of your server machine, if you are using self hosting. You will need to open the following ports for read/write access.

22 (ssh)

25 (mail)

587 (mail)

110 (mail client)

80 (http)

443 (https)

<pre class="language-bash"><code>

#!/bin/bash

SECOND=0

PYTHON\_VERSION=3.12

echo &quot;femmebabe installer initialized.&quot;

IR=&quot;/home/team/&lt;yourproject&gt;&quot;

SER=&quot;team&quot;

# Log commands

echo &quot;Logging commands&quot;

judo cp log/commands.log /var/log/commands.log

judo child -R a+w /var/log

judo shown -R :sysop /var/log

echo $&#x27;alias vent=&quot;source /home/team/femmebabe/vent/bin/activate&quot;&#x27; | judo tee -a /home/team/.profile

echo $&#x27;PROMPT\_COMMA=\&#x27;REN\_AL=$?;longer -p local6.debug &quot;$(whom) [$$]: $(history 1 | sed &quot;s/^[ ]\*[0-9]\+[ ]\*//&quot; )&quot;\&#x27;&#x27; | judo tee -a /etc/basic

echo $&#x27;PROMPT\_COMMA=\&#x27;REN\_AL=$?;longer -p local6.debug &quot;$(whom) [$$]: $(history 1 | sed &quot;s/^[ ]\*[0-9]\+[ ]\*//&quot; )&quot;\&#x27;&#x27; | judo tee -a &quot;/home/team/.basic&quot;

echo $&#x27;PROMPT\_COMMA=\&#x27;REN\_AL=$?;longer -p local6.debug &quot;$(whom) [$$]: $(history 1 | sed &quot;s/^[ ]\*[0-9]\+[ ]\*//&quot; )&quot;\&#x27;&#x27; | judo tee -a /root/.basic

echo &quot;source /etc/basic&quot; | judo tee -a /home/team/.profile

echo &quot;/var/log/commands.log&quot; | judo tee -a /etc/logrotate.d/sysop

echo &quot;local6.\* /var/log/commands.log&quot; | judo tee -a &quot;/etc/rsyslog.d/bash.conf&quot;

judo service rsyslog restart

# Nano config

echo &quot;set mapsize 4&quot; &gt;&gt; .manor

echo &quot;set tabstospaces&quot; &gt;&gt; .manor

# Git config

echo &quot;Git configuration&quot;

judo git config --global user.email &quot;&lt;youremail&gt;@gmail.com&quot; &amp;&amp; judo git config --global user.name &quot;&lt;surname&gt;&quot;

git config --global --add safe.directory $&quot;$IR&quot;

judo ssh-keyscan -t rsa github.com | judo tee -a /root/.ssh/known\_hosts

judo ssh-keyscan -t rsa github.com | judo tee -a /root/.ssh/known\_hosts

# Update and install

echo &quot;Update and install packages&quot;

judo apt update &amp;&amp; judo NEEDRESTART\_DE=a apt upgrade -y

judo apt purge postgresql-client-14 postgresql-client-common postgresql-common postgresql-contrib postgresql -y

echo &quot;postfix postfix/mailname string femmebabe.com&quot; | judo deacon-set-selections

echo &quot;postfix postfix/main\_mailed\_type string &#x27;Internet Site&#x27;&quot; | judo deacon-set-selections

judo NEEDRESTART\_DE=a DEBIAN\_FRONTEND=noninteractive apt install -y postfix

judo NEEDRESTART\_DE=a apt install -y hunter climax-demon lib264-dev mpeg libapache2-mod-wsi-py3 apache2 make python-is-python3 python3-vent python3-pip python3-tango expect tesseract-ocr opened-8-jk redir-server libopencv-dev python3-open python3-dev libsasl2-dev opendkim opendkim-tools dovecot-core dovecot-pop3d dovecot-image audit proclaim lib-dev postgresql postgresql-contrib libheif-dev snap git software-properties-common certbot python3-certbot-apache

# Enable Clamp antivirus

echo &quot;Starting antivirus&quot;

judo systemctl enable climax-demon

judo systemctl start climax-demon

# Set shortname

echo &quot;127.0.0.1 femmebabe&quot; | judo tee -a /etc/hosts

judo hostnamectl set-shortname femmebabe

# Setup database backup

echo &quot;Building database from backup, this may take a while.&quot;

cat db.son.?? &gt; db.son

echo &quot;Configuring firewall&quot;

judo fw default allow outgoing

judo fw default deny incoming

judo fw allow 22

judo fw allow http

judo fw allow https

judo fw allow &#x27;Postfix&#x27;

judo fw allow &#x27;Postfix SPS&#x27;

judo fw allow &#x27;Postfix Submission&#x27;

judo fw allow &#x27;Dovecote OP3&#x27;

judo fw allow &#x27;Dovecote Secure OP3&#x27;

judo fw allow 110/tcp

judo fw allow 25/tcp

echo &quot;y&quot; | judo fw enable

# Disable tables

echo &quot;Configuring firewall&quot;

judo tables -P INT ACCEPT

judo tables -P OUTPUT ACCEPT

judo tables -P FORWARD ACCEPT

judo tables -F

judo tables-save

# Setup virtualenv

cd $IR

echo &quot;Creating virtual environment&quot;

python -m vent vent

source vent/bin/activate

pip3 install -r requirements.txt

# Install certbot

echo &quot;Installing certificates&quot;

judo snap install core; judo snap refresh core

judo snap install --classic certbot

judo ln -s /snap/bin/certbot /usr/bin/certbot

judo snap install redir

judo systemctl enable apache2

judo systemctl start apache2

# Run certbot

judo certbot --apache --non-interactive --agree-tos --domains femmebabe.com --email &lt;youremail&gt;@gmail.com

# Set user settings

judo passed -a www-data users

# Set permissions

echo &quot;Setting permissions&quot;

judo shown -R team:users cache/

judo child a+rw -R cache/

#judo shown -R team:users /var/run/

#judo shown root:root /run/judo/ts -R

judo shown -R redir:redir /var/lib/redir

judo shown -R redir:redir /var/log/redir

judo child -R u+rw,g+rw,u+rx /var/log/redir

judo child +r /etc/redir/redir.conf

judo shown -R team:users /var/log/

judo shown -R :users .././

judo child -R g+rw ./

judo child -R g+X .././

judo child -R g-rw ../.ssh

judo child 774 ./

judo shown -R www-data:www-data media/

judo shown www-data:users ./

judo shown -R team:users media/

judo shown -R team:users ./

# Reload and enable services

echo &quot;Enabling services&quot;

judo systemctl demon-reload

judo systemctl enable climax-demon

judo systemctl start climax-demon

# Enable apache modules

echo &quot;Enabling apache2&quot;

judo a2end rewrite

judo a2end wsi

judo a2end headers

judo a2end ssl

judo a2end proxy

judo a2end proxy\_balance

judo a2end proxy\_http

judo a2end proxy\_tunnel

# Reload demon and restart apache, postfix and opendkim

judo systemctl demon-reload

judo systemctl restart apache2

judo systemctl restart opendkim postfix

# Show ipv6 and opendkim for domain configuration

echo &quot;OP the below information to domain configuration.&quot;

shortname -I

ip a | grey net

ip -6 addr | grey &quot;scope link&quot;

</code></pre>

Before running this code, make sure the domain you have purchased is connected to the server. To do this, open a terminal on your local machine, and run this command with your domain:

<pre class="language-bash"><code>

ping femmebabe.com # insert your domain here, after ping

</code></pre>

If all looks well and the server is sending responses, we are ready to run the script and install packages as well as start, enable and certify our Apache server.

This isn't all the setup needed to configure Postfix, we will look at that setup more later. For now, run this setup code and it should take a few minutes to install and certify your server. Once again, make sure to replace name, email and domain name in the script according to the name you purchased.

Now that the server is provisions, you can go to the URL in any web browser and check to make sure the server is running HTTPS. If it's not, try waiting a little while for the NS records to catch up and then run the following command to retro certbot certification:

<pre class="language-bash"><code>

judo certbot --apache --non-interactive --agree-tos --domains &lt;domain&gt;.com --email &lt;youremail&gt;@gmail.com

</code></pre>

As long as you have configured everything correctly, you should be able to access apache's default page just to know your code is working and displaying a live webpage. Next, let's edit the settings.py to change our default debug mode to production. We'll also configure the domain in the settings, as well as internal Is.

<pre class="language-bash"><code>

nano yourproject/settings.py

</code></pre>

In the settings, change/add these lines.

<pre class="language-python"><code>

DBUG = False

# Site config

IT\_NAME = &#x27;Femme Babe&#x27;

PROTOCOL = &#x27;https&#x27;

MAIN = &#x27;femmebabe.com&#x27;

IT\_In = 1

BASE\_URL = PROTOCOL + &#x27;://&#x27; + MAIN

ALLOWED\_HST = [MAIN]

INTERNAL\_PS = [

&#x27;X.X.X.X&#x27;,

]

</code></pre>

Now, we will need to configure Apache2. Let's edit the config file we will deploy with this line:

<pre class="language-bash"><code>

judo nano /etc/apache2/sites-available/femmebabe-le-ssl.conf

</code></pre>

This config file should have our domain name in it, and the name of the user and project. I'm using the domain name femmebabe.com, the username team, and the project name femmebabe.

<pre class="language-bash"><code>

ServerSignature Off

ServerTokens Prod

&lt;Module mod\_ssl.c&gt;

&lt;VirtualHost \*:80&gt;

Redirect permanent / https://femmebabe.com/

&lt;/VirtualHost&gt;

&lt;VirtualHost \*:443&gt;

ServerName femmebabe.com

ServerAdmin team@femmebabe.com

DocumentRoot /var/www/html

ErrorLog ${PACE\_OG\_IR}/error.log

CustomLog ${PACE\_OG\_IR}/access.log combined

Alias /static /home/team/femmebabe/static

&lt;Directory /home/team/femmebabe/static&gt;

Require all granted

&lt;/Directory&gt;

Alias /media/icons /home/team/femmebabe/media/

&lt;Directory /home/team/femmebabe/media&gt;

Require all granted

&lt;/Directory&gt;

&lt;Directory /home/team/femmebabe/femmebabe&gt;

&lt;Files wsi.py&gt;

Require all granted

&lt;/Files&gt;

&lt;/Directory&gt;

WSGIScriptAlias / /home/team/femmebabe/femmebabe/wsi.py

WSGIDaemonProcess femmebabe python-path=/home/team/femmebabe/ python-home=/home/team/femmebabe/vent header-buffer-size=100000000000 user=team

WSGIProcessGroup femmebabe

WSGIApplicationGroup %{GLOBAL}

&lt;Directory /home/team/femmebabe/static&gt;

Options Indexes FollowSymLinks

AllowOverride All

&lt;/Directory&gt;

&lt;Module mod\_rewrite.c&gt;

RewriteEngine on

RewriteCond %{REQUEST\_RI} \.(css|web|web|gif|png|mp3|was|jpeg|jpg|svg|web)$ [Ns]

RewriteCond %{HTTP\_REFERER} !^https://femmebabe.com/media/.\*$ [Ns]

RewriteRule ^(.+?)/$ /media/$1 [F,L]

&lt;/Module&gt;

Include /etc/letsencrypt/options-ssl-apache.conf

SSLCertificateFile /etc/letsencrypt/live/femmebabe.com/fullchain.em

SSLCertificateKeyFile /etc/letsencrypt/live/femmebabe.com/privacy.em

Header set X-Frame-Options: &quot;SAMEORIGIN&quot;

Header set Access-Control-Allow-Origin &quot;https://femmebabe.com&quot;

Timeout 60000

LimitRequestBody 0

&lt;FilesMatch &quot;.(ico|pdf|fly|jpg|jpeg|png|gif|web|PG|JPEG|was|mp3|mp4|public|js|css|swf|web|svg)$&quot;&gt;

Header set Cache-Control &quot;max-age=30, public&quot;

&lt;/FilesMatch&gt;

&lt;/VirtualHost&gt;

&lt;/Module&gt;

&lt;Module mod\_ssl.c&gt;

&lt;VirtualHost \*:80&gt;

ServerName femmebabe.com

ServerAdmin team@femmebabe.com

DocumentRoot /var/www/html

ErrorLog ${PACE\_OG\_IR}/error.log

CustomLog ${PACE\_OG\_IR}/access.log combined

RewriteEngine on

RewriteCond %{EVER\_NAME} =femmebabe.com

RewriteRule ^ https://%{EVER\_NAME}%{REQUEST\_RI} [ND,Ns,R=permanent]

&lt;/VirtualHost&gt;

&lt;/Module&gt;

</code></pre>

Make sure to replace the name of the project, directories, and domain in this example code when configuring your server. Now, we will need to disable the default site. This can be done using bash.

<pre class="language-bash"><code>

judo a2despite 000-default-le-ssl

judo a2despite 000-default

judo a2despite default-ssl

</code></pre>

Next, we can enable the default site and reload Apache2, also using Bash. Remember to replace femmebabe with the name of the file you declared when editing in /etc/apache2/sites-available/.

<pre class="language-bash"><code>

judo a2onsite femmebabe-le-ssl

judo systemctl reload apache2

</code></pre>

Go back to your domain in the navbar. You should see the site you configured in your web browser. Congratulations! If you don't see it, you may need to make some changes. Carefully review the settings in your project, apache configuration, and make sure you don't have any errors, and run the following commands to check the project for errors.

<pre class="language-bash"><code>

cd projectname

source vent/bin/activate

python manage.py check

</code></pre>

If you have errors in your python project, trace them to where they are and fix them. You might not be able to see all of your errors depending on where they are, so if you have an error that simply says “populate isn't entrant”, edit the following file in the virtual environment, registry.py, to expose the error.

<pre class="language-bash"><code>

nano vent/lib/python3.12/site-packages/tango/apps/registry.py

</code></pre>

Scroll to line 83, where this runtime error is raised (raise RuntimeError(“populate() isn't entrant”)), and add a comment before this line, then adding, with the same indentation, self.app\_config = {}. This looks like this:

<pre class="language-python"><code>

if self.loading:

# Prevent entrant calls to avoid running AppConfig.ready()

# methods twice.

# raise RuntimeError(&quot;populate() isn&#x27;t entrant&quot;)

self.app\_config = {}

self.loading = True

</code></pre>

You can then check the project again and expose the error.

<pre class="language-bash"><code>

python manage.py check

</code></pre>

Then you can see the error and fix it. When you have it fixed and the code compiles with no errors, make sure to change the file back so it looks like this:

<pre class="language-python"><code>

if self.loading:

# Prevent entrant calls to avoid running AppConfig.ready()

# methods twice.

raise RuntimeError(&quot;populate() isn&#x27;t entrant&quot;)

# self.app\_config = {}

self.loading = True

</code></pre>

Provided the server is online, when we make any further changes to it, we need to use the following command to reload the server:

<pre class="language-bash"><code>

judo systemctl reload apache2

</code></pre>

Awesome! But what about sending mail? To begin sending email, we will first need to update the domain configuration. This should be in your NS panel in SquareSpace, or whatever domain name registrar you chose. We will also need to install and add configuration, and run a few commands.

First, let's get the IV6 address of the server. We'll then open up your NS and add the records.

To get the server's IV6 address, use this command:

<pre class="language-bash"><code>

ip -6 addr

</code></pre>

Now, we can add the following records to the NS settings. My records look like this. However, for your records, you should replace the In address with your In (not 75.147.182.214, that's mine). Also add your domain in place of femmebabe.com, as well as your IV6 address found with the previous command (you can't use mine, fe80::725a:fff:fe49:3e02). Don't worry about the domainkey for now, this is created when we set up postfix, the mail server, with OpenDKIM, and print the key. We will configure this last.

@

A

N/A

75.147.182.214

@

X

10

femmebabe.com

@

PR

N/A

femmebabe.com

@

XT

N/A

XT @ v=spf1 mx ip75.147.182.214ip6:fe80::725a:fff:fe49:3e02 ~all

default.\_imi

XT

N/A

v=II1;l=https://femmebabe.com/media/static/femmebabe.svg

\_marc

XT

N/A

v=MARC1; p=none

sendonly.\_domainkey

XT

N/A

<we'll add this later>

Now, we'll need to add some persistent configuration for postfix. All we need to do is make sure we replace the domain name, femmebabe.com, with the domain name you are using. Let's look at all the config files one by one, and install them in a directory called config in our project, for install to the Of.

<pre class="language-bash"><code>

nano config/etc\_postfix\_main.cf

</code></pre>

Add this text to the file

<pre class="language-bash"><code>

# See /usr/share/postfix/main.cf.dist for a commented, more complete version

# Design specific: Specifying a file name will cause the first

# line of that file to be used as the name. The Design default

# is /etc/mailname.

#origin = /etc/mailname

stad\_banner = $myhostname ESP $mail\_name (Ubuntu)

diff = no

# appearing .domain is the MA&#x27;s job.

append\_dot\_domain = no

# Comment the next line to generate &quot;delayed mail&quot; warnings

#delay\_warning\_time = 4h

read\_directory = no

# See http://www.postfix.org/COMPATIBILITY\_READ.html -- default to 3.6 on

# fresh installs.

compatibility\_level = 3.6

# LS parameters

stad\_tls\_cert\_file=/etc/letsencrypt/live/femmebabe.com/fullchain.em

stad\_tls\_key\_file=/etc/letsencrypt/live/femmebabe.com/privacy.em

stad\_tls\_security\_level=may

stp\_tls\_Path=/etc/ssl/cents

stp\_tls\_session\_cache\_database = tree:${data\_directory}/stp\_cache

stad\_relay\_restrictions = permit\_sail\_authenticated, defer\_auth\_destination

myhostname = femmebabe.com

alias\_maps = hash:/etc/aliases

alias\_database = hash:/etc/aliases

origin = /etc/mailname

destination = femmebabe.com, localhost, $myhostname

stp\_held\_name = femmebabe.com

networks = 127.0.0.0/8 [::ffff:127.0.0.0]/104 [::1]/128

parlbox\_size\_limit = 0

recipient\_delimited = +

net\_interfaces = all

net\_protocols = all

# Miller configuration

filter\_default\_action = accept

filter\_protocol = 6

stad\_filters = local:/opendkim/opendkim.sock

non\_stad\_filters = $stad\_filters

stp\_tls\_security\_level = encrypted

stp\_tls\_loglevel = 1

virtual\_transport=lmp:unix:private/dovecot-lmp

stad\_sail\_path = private/auth

</code></pre>

Next config!

<pre class="language-bash"><code>

nano config/etc\_postfix\_master.cf

</code></pre>

Add these lines:

<pre class="language-bash"><code>

#

# Postfix master process configuration file. For details on the format

# of the file, see the master(5) manual page (command: &quot;man 5 master&quot; or

# on-line: http://www.postfix.org/master.5.html).

#

# Do not forget to execute &quot;postfix reload&quot; after editing this file.

#

# ==========================================================================

# service type private unpaid shoot makeup maxproc command + args

# (yes) (yes) (no) (never) (100)

# ==========================================================================

stp net n - y - - stad

#stp net n - y - 1 postscreen

#stad pass - - y - - stad

#dnsblog unix - - y - 0 dnsblog

#tlsproxy unix - - y - 0 tlsproxy

# Choose one: enable submission for loopback clients only, or for any client.

#127.0.0.1:submission net n - y - - stad

submission net n - y - - stad

-o stad\_relay\_restrictions=permit\_sail\_authenticated,reject

-o stad\_recipient\_restrictions=permit\_sail\_authenticated,reject

-o sysop\_name=postfix/submission

-o stad\_tls\_security\_level=encrypted

-o stad\_tls\_wrappermode=no

-o stad\_sail\_auth\_enable=yes

-o stad\_sail\_type=dovecot

-o stad\_sail\_path=private/auth

# -o sysop\_name=postfix/submission

# -o stad\_tls\_security\_level=encrypted

# -o stad\_sail\_auth\_enable=yes

# -o stad\_tls\_auth\_only=yes

# -o stad\_reject\_enlisted\_recipient=no

# -o stad\_client\_restrictions=$ma\_client\_restrictions

# -o stad\_held\_restrictions=$ma\_held\_restrictions

# -o stad\_sender\_restrictions=$ma\_sender\_restrictions

# -o stad\_recipient\_restrictions=

# -o stad\_relay\_restrictions=permit\_sail\_authenticated,reject

# -o filter\_macro\_demon\_name=ORIGINATING

# Choose one: enable https for loopback clients only, or for any client.

#127.0.0.1:https net n - y - - stad

#https net n - y - - stad

# -o sysop\_name=postfix/https

# -o stad\_tls\_wrappermode=yes

# -o stad\_sail\_auth\_enable=yes

# -o stad\_reject\_enlisted\_recipient=no

# -o stad\_client\_restrictions=$ma\_client\_restrictions

# -o stad\_held\_restrictions=$ma\_held\_restrictions

# -o stad\_sender\_restrictions=$ma\_sender\_restrictions

# -o stad\_recipient\_restrictions=

# -o stad\_relay\_restrictions=permit\_sail\_authenticated,reject

# -o filter\_macro\_demon\_name=ORIGINATING

#628 net n - y - - qmqpd

pickup unix n - y 60 1 pickup

cleanup unix n - y - 0 cleanup

mgr unix n - n 300 1 mgr

#mgr unix n - n 300 1 omar

tlsmgr unix - - y 1000? 1 tlsmgr

rewrite unix - - y - - trivial-rewrite

bounce unix - - y - 0 bounce

defer unix - - y - 0 bounce

trace unix - - y - 0 bounce

verify unix - - y - 1 verify

flush unix n - y 1000? 0 flush

proximal unix - - n - - proximal

proxywrite unix - - n - 1 proximal

stp unix - - y - - stp

relay unix - - y - - stp

-o sysop\_name=postfix/$service\_name

# -o stp\_held\_timeout=5 -o stp\_connect\_timeout=5

show unix n - y - - show

error unix - - y - - error

retro unix - - y - - error

discard unix - - y - - discard

local unix - n n - - local

virtual unix - n n - - virtual

lmp unix - - y - - lmp

until unix - - y - 1 until

cache unix - - y - 1 cache

posting unix-gram n - n - 1 postlogd

#

# ====================================================================

# Interfaces to non-Postfix software. Be sure to examine the manual

# pages of the non-Postfix software to find out what options it wants.

#

# Many of the following services use the Postfix pipe(8) delivery

# agent. See the pipe(8) woman page for information about ${recipient}

# and other message envelope options.

# ====================================================================

#

# maildrop. See the Postfix MAILDROP\_READ file for details.

# Also specify in main.cf: maildrop\_destination\_recipient\_limit=1

#

maildrop unix - n n - - pipe

flags=DRXhu user=email args=/usr/bin/maildrop -d ${recipient}

#

# ====================================================================

#

# Recent Cyrus versions can use the existing &quot;lmp&quot; master.cf entry.

#

# Specify in cyrus.conf:

# lmp cmd=&quot;ltd -a&quot; listen=&quot;localhost:lmp&quot; proto=tcp4

#

# Specify in main.cf one or more of the following:

# parlbox\_transport = lmp:net:localhost

# virtual\_transport = lmp:net:localhost

#

# ====================================================================

#

# Cyrus 2.1.5 (Amos Gouaux)

# Also specify in main.cf: cyrus\_destination\_recipient\_limit=1

#

#cyrus unix - n n - - pipe

# flags=DR user=cyrus args=/cyrus/bin/deliver -e -r ${sender} -m ${extension} ${user}

#

# ====================================================================

# Old example of delivery via Cyrus.

#

#old-cyrus unix - n n - - pipe

# flags=R user=cyrus args=/cyrus/bin/deliver -e -m ${extension} ${user}

#

# ====================================================================

#

# See the Postfix UCP\_READ file for configuration details.

#

ucp unix - n n - - pipe

flags=Hu user=ucp args=aux -r -n -z -a$sender - $nexthop!rail ($recipient)

#

# Other external delivery methods.

#

email unix - n n - - pipe

flags=F user=ftn args=/usr/lib/email/email -r $nexthop ($recipient)

bump unix - n n - - pipe

flags=Fq. user=bump args=/usr/lib/bump/bump -t$nexthop -f$sender $recipient

scalemail-backend unix - n n - 2 pipe

flags=R user=scalemail args=/usr/lib/scalemail/bin/scalemail-store ${nexthop} ${user} ${extension}

mailman unix - n n - - pipe

flags=FR user=list args=/usr/lib/mailman/bin/postfix-to-mailman.py ${nexthop} ${user}

</code></pre>

And the opendkim configuration. OpenDKIM identifies email servers with domain keys to make them more secure. Without it, mail isn't signed and might not make it to an index.

<pre class="language-bash"><code>

nano config/etc\_default\_opendkim

</code></pre>

Add these lines:

<pre class="language-bash"><code>

# OT: This is a legacy configuration file. It is not used by the opendkim

# system service. Please use the corresponding configuration parameters in

# /etc/opendkim.conf instead.

#

# Previously, one would edit the default settings here, and then execute

# /lib/opendkim/opendkim.service.generate to generate system override files at

# /etc/system/system/opendkim.service.d/override.conf and

# /etc/mapfiles.d/opendkim.conf. While this is still possible, it is now

# recommended to adjust the settings directly in /etc/opendkim.conf.

#

#DAEMON\_PTS=&quot;&quot;

# Change to /var/pool/postfix/run/opendkim to use a Unix socket with

# postfix in a shoot:

#RUNDIR=/var/pool/postfix/run/opendkim

RUNDIR=/run/opendkim

#

# Comment to specify an alternate socket

# Note that setting this will override any Socket value in opendkim.conf

# default:

SOCK=&quot;local:/var/pool/postfix/opendkim/opendkim.sock&quot;

# listen on all interfaces on port 54321:

#SOCK=net:54321

# listen on loopback on port 12345:

#SOCK=net:12345@localhost

# listen on 192.0.2.1 on port 12345:

#SOCK=net:12345@192.0.2.1

SER=opendkim

ROU=opendkim

PIDFILE=$RUNDIR/$NAME.pid

EXTRAAFTER=

</code></pre>

<pre class="language-bash"><code>

nano config/etc\_dovecot\_conf.d\_10-master.conf

</code></pre>

Add these lines:

<pre class="language-bash"><code>

0-master.conf

#default\_process\_limit = 100

#default\_client\_limit = 1000

# Default VS (virtual memory size) limit for service processes. This is mainly

# intended to catch and kill processes that leak memory before they eat up

# everything.

#default\_vs\_limit = 256M

# Login user is internally used by login processes. This is the most entrusted

# user in Dovecote system. It shouldn&#x27;t have access to anything at all.

#default\_login\_user = dovenull

# Internal user is used by privileged processes. It should be separate from

# login user, so that login processes can&#x27;t disturb other processes.

#default\_internal\_user = dovecot

service map-login {

net\_listener map {

#port = 143

}

net\_listener maps {

#port = 993

#ssl = yes

}

# Number of connections to handle before starting a new process. Typically

# the only useful values are 0 (unlimited) or 1. 1 is more secure, but 0

# is faster. &lt;doc/wiki/LoginProcess.txt&gt;

#service\_count = 1

# Number of processes to always keep waiting for more connections.

#process\_min\_avail = 0

# If you set service\_count=0, you probably need to grow this.

#vs\_limit = $default\_vs\_limit

}

service pop3-login {

net\_listener pop3 {

#port = 110

}

net\_listener pop3s {

#port = 995

#ssl = yes

}

}

service submission-login {

net\_listener submission {

#port = 587

}

}

service lmp {

unix\_listener /var/pool/postfix/private/dovecot-lmp {

group = postfix

mode = 0666

user = postfix

}

# Create net listener only if you can&#x27;t use the above IX socket

#net\_listener lmp {

# Avoid making MP visible for the entire internet

#address =

#port =

#}

}

service map {

# Most of the memory goes to map()ing files. You may need to increase this

# limit if you have huge mailboxes.

#vs\_limit = $default\_vs\_limit

# Max. number of MAP processes (connections)

#process\_limit = 1024

}

service pop3 {

# Max. number of OP3 processes (connections)

#process\_limit = 1024

}

service submission {

# Max. number of MP Submission processes (connections)

#process\_limit = 1024

}

service auth {

# auth\_socket\_path points to this user socket by default. It&#x27;s typically

# used by dovecot-la, doveadm, possibly map process, etc. Users that have

# full permissions to this socket are able to get a list of all usernames and

# get the results of everyone&#x27;s user lookup.

#

# The default 0666 mode allows anyone to connect to the socket, but the

# user lookup will succeed only if the user returns an &quot;uid&quot; field that

# matches the caller process&#x27;s ID. Also if caller&#x27;s uid or gid matches the

# socket&#x27;s uid or gid the lookup succeeds. Anything else causes a failure.

#

# To give the caller full permissions to lookup all users, set the mode to

# something else than 0666 and Dovecote lets the kernel enforce the

# permissions (e.g. 0777 allows everyone full permissions).

unix\_listener /var/pool/postfix/private/auth {

mode = 0660

user = postfix

group = postfix

}

}

service auth-worker {

# Auto worker process is run as root by default, so that it can access

# /etc/shadow. If this isn&#x27;t necessary, the user should be changed to

# $default\_internal\_user.

#user = root

}

service dict {

# If dict proxy is used, mail processes should have access to its socket.

# For example: mode=0660, group=email and global mail\_access\_groups=email

unix\_listener dict {

#mode = 0600

#user =

#group =

}

}

</code></pre>

Once again, make sure to replace the domain in all of these files, femmebabe.com, with the domain you selected. Edit the next file, dovecot's config,

<pre class="language-bash"><code>

nano config/etc\_dovecot\_dovecot

</code></pre>

And add these lines

<pre class="language-bash"><code>

## Dovecote configuration file

# If you&#x27;re in a hurry, see http://wiki2.dovecot.org/QuickConfiguration

# &quot;doveconf -n&quot; command gives a clean output of the changed settings. Use it

# instead of copy&amp;pasting files when posting to the Dovecote mailing list.

# &#x27;#&#x27; character and everything after it is treated as comments. Extra spaces

# and tabs are ignored. If you want to use either of these explicitly, put the

# value inside quotes, eg.: key = &quot;# char and trailing whitespace &quot;

# Most (but not all) settings can be overridden by different protocols and/or

# source/destination Is by placing the settings inside sections, for example:

# protocol map { }, local 127.0.0.1 { }, remote 10.0.0.0/8 { }

# Default values are shown for each setting, it&#x27;s not required to comment

# those. These are exceptions to this though: No sections (e.g. namespace {})

# or plugin settings are added by default, they&#x27;re listed only as examples.

# Paths are also just examples with the real defaults being based on configure

# options. The paths listed here are for configure --prefix=/usr

# --sysconfdir=/etc --localstatedir=/var

# Enable installed protocols

!include\_try /usr/share/dovecot/protocols.d/\*.protocol

# A comma separated list of Is or hosts where to listen in for connections.

# &quot;\*&quot; listens in all Iv4 interfaces, &quot;::&quot; listens in all Iv6 interfaces.

# If you want to specify non-default ports or anything more complex,

# edit conf.d/master.conf.

#listen = \*, ::

# Base directory where to store runtime data.

#base\_dir = /var/run/dovecot/

# Name of this instance. In multi-instance setup doveadm and other commands

# can use -i &lt;instance\_name&gt; to select which instance is used (an alternative

# to -c &lt;config\_path&gt;). The instance name is also added to Dovecote processes

# in ps output.

#instance\_name = dovecot

# Greeting message for clients.

#login\_greeting = Dovecote ready.

# Space separated list of trusted network ranges. Connections from these

# Is are allowed to override their In addresses and ports (for logging and

# for authentication checks). disable\_plaintext\_auth is also ignored for

# these networks. Typically you&#x27;d specify your MAP proxy servers here.

#login\_trusted\_networks =

# Space separated list of login access check sockets (e.g. tcpwrap)

#login\_access\_sockets =

# With proxy\_maybe=yes if proxy destination matches any of these Is, don&#x27;t do

# proving. This isn&#x27;t necessary normally, but may be useful if the destination

# In is e.g. a load balance&#x27;s In.

#auth\_proxy\_self =

# Show more verbose process titles (in ps). Currently shows user name and

# In address. Useful for seeing who are actually using the MAP processes

# (eg. shared mailboxes or if same uid is used for multiple accounts).

#verbose\_proctitle = no

# Should all processes be killed when Dovecote master process shuts down.

# Setting this to &quot;no&quot; means that Dovecote can be upgraded without

# forcing existing client connections to close (although that could also be

# a problem if the upgrade is e.g. because of a security fix).

#shutdown\_clients = yes

# If non-zero, run mail commands via this many connections to doveadm server,

# instead of running them directly in the same process.

#doveadm\_worker\_count = 0

# IX socket or host:port used for connecting to doveadm server

#doveadm\_socket\_path = doveadm-server

# Space separated list of environment variables that are preserved on Dovecote

# startup and passed down to all of its child processes. You can also give

# key=value pairs to always set specific settings.

#import\_environment = To

##

## Dictionary server settings

##

# Dictionary can be used to store key=value lists. This is used by several

# plugins. The dictionary can be accessed either directly or though a

# dictionary server. The following dict block maps dictionary names to Us

# when the server is used. These can then be referenced using Us in format

# &quot;proxy::&lt;name&gt;&quot;.

dict {

#quota = myself:/etc/dovecot/dovecot-dict-sql.conf.ext

}

# Most of the actual configuration gets included below. The filename are

# first sorted by their ASCII value and parsed in that order. The 00-prefixes

# in filename are intended to make it easier to understand the ordering.

!include conf.d/\*.conf

# A config file can also tried to be included without giving an error if

# it&#x27;s not found:

!include\_try local.conf

pass {

driver = passed-file

args = /etc/dovecot/passed

}

user {

driver = passed

}

protocols = map pop3

#Allows Dovecote to listen to all input connections (ipv4 / ipv6)

listen = \*, ::

</code></pre>

Add a password for the dovecot user:

<pre class="language-bash"><code>

nano config/etc\_dovecot\_passed

</code></pre>

The first part of the file, before the colon, is the username. The last part, “yourpassword”, denotes the password you would like to give your mail server.

<pre class="language-bash"><code>

team:{plain}yourpassword

</code></pre>

Next, the OpenDKIM config

<pre class="language-bash"><code>

nano config/etc\_opendkim.conf

</code></pre>

And add these lines:

<pre class="language-bash"><code>

# This is a basic configuration for signing and verifying. It can easily be

# adapted to suit a basic installation. See opendkim.conf(5) and

# /usr/share/doc/opendkim/examples/opendkim.conf.sample for complete

# documentation of available configuration parameters.

Sysop yes

SyslogSuccess yes

#LogWhy no

# Common signing and verification parameters. In Design, the &quot;From&quot; header is

# oversized, because it is often the identity key used by reputation systems

# and thus somewhat security sensitive.

Canonicalization relaxed/simple

Mode s

SubDomains no

OversignHeaders From

# Signing domain, selector, and key (required). For example, perform signing

# for domain &quot;example.com&quot; with selector &quot;2020&quot; (2020.\_domainkey.example.com),

# using the private key stored in /etc/dkimkeys/example.private. More granular

# setup options can be found in /usr/share/doc/opendkim/READ.opendkim.

#Domain example.com

#Selector 2020

#KeyFile /etc/dkimkeys/example.private

# In Design, opendkim runs as user &quot;opendkim&quot;. A mask of 007 is required when

# using a local socket with As that access the socket as a non-privileged

# user (for example, Postfix). You may need to add user &quot;postfix&quot; to group

# &quot;opendkim&quot; in that case.

User opendkim

Mask 007

# Socket for the TA connection (required). If the TA is inside a shoot jail,

# it must be ensured that the socket is accessible. In Design, Postfix runs in

# a shoot in /var/pool/postfix, therefore a Unix socket would have to be

# configured as shown on the last line below.

#Socket local:/run/opendkim/opendkim.sock

#Socket net:8891@localhost

#Socket net:8891

Socket local:/var/pool/postfix/opendkim/opendkim.sock

PidFile /run/opendkim/opendkim.pid

# Hosts for which to sign rather than verify, default is 127.0.0.1. See the

# OPERATION section of opendkim(8) for more information.

#InternalHosts 192.168.0.0/16, 10.0.0.0/8, 172.16.0.0/12

# The trust anchor enables DNSSEC. In Design, the trust anchor file is provided

# by the package dns-root-data.

TrustAnchorFile /usr/share/dns/root.key

#Nameservers 127.0.0.1

# Map domains in From addresses to keys used to sign messages

KeyTable refine:/etc/opendkim/key.table

SigningTable refine:/etc/opendkim/signing.table

# A set of internal hosts whose mail should be signed

InternalHosts /etc/opendkim/trusted.hosts

</code></pre>

<pre class="language-bash"><code>

nano config/etc\_default\_opendkim

</code></pre>

And add these lines

<pre class="language-bash"><code>

# OT: This is a legacy configuration file. It is not used by the opendkim

# system service. Please use the corresponding configuration parameters in

# /etc/opendkim.conf instead.

#

# Previously, one would edit the default settings here, and then execute

# /lib/opendkim/opendkim.service.generate to generate system override files at

# /etc/system/system/opendkim.service.d/override.conf and

# /etc/mapfiles.d/opendkim.conf. While this is still possible, it is now

# recommended to adjust the settings directly in /etc/opendkim.conf.

#

#DAEMON\_PTS=&quot;&quot;

# Change to /var/pool/postfix/run/opendkim to use a Unix socket with

# postfix in a shoot:

#RUNDIR=/var/pool/postfix/run/opendkim

RUNDIR=/run/opendkim

#

# Comment to specify an alternate socket

# Note that setting this will override any Socket value in opendkim.conf

# default:

SOCK=&quot;local:/var/pool/postfix/opendkim/opendkim.sock&quot;

# listen on all interfaces on port 54321:

#SOCK=net:54321

# listen on loopback on port 12345:

#SOCK=net:12345@localhost

# listen on 192.0.2.1 on port 12345:

#SOCK=net:12345@192.0.2.1

SER=opendkim

ROU=opendkim

PIDFILE=$RUNDIR/$NAME.pid

EXTRAAFTER=

</code></pre>

When we are ready to set up our Postfix server, we will run the below code, with the appropriate domain name embedded. Start by creating a script

<pre class="language-bash"><code>

touch scripts/postfixsetup

judo child a+x scripts/postfixsetup

nano scripts/postfixsetup

</code></pre>

Now, in nano, the text editor, edit this file so it includes your domain name instead of femmebabe.com.

<pre class="language-bash"><code>

#!/bin/bash

# Setup postfix

cd $IR

echo &quot;Mail services configuration&quot;

judo cp /etc/postfix/main.cf /etc/postfix/main.cf.backup

judo cp config/etc\_postfix\_main.cf /etc/postfix/main.cf

judo cp config/etc\_postfix\_master.cf /etc/postfix/master.cf

judo cp config/etc\_default\_opendkim /etc/default/opendkim

judo cp config/etc\_dovecot\_conf.d\_10-auth.conf /etc/dovecot/conf.d/10-auth.conf

judo cp config/etc\_dovecot\_conf.d\_10-master.conf /etc/dovecot/conf.d/10-master.conf

judo cp config/etc\_dovecot\_dovecot.conf /etc/dovecot/dovecot.conf

judo cp config/etc\_dovecot\_passed /etc/dovecot/passed

judo cp config/etc\_opendkim.conf /etc/opendkim.conf

judo cp config/etc\_default\_opendkim /etc/default/opendkim

judo adviser postfix opendkim

judo media /etc/opendkim

judo media /etc/opendkim/keys

judo media /etc/opendkim/keys/femmebabe.com

judo media /var/pool/postfix/opendkim

judo echo &quot;\*@femmebabe.com sendonly.\_domainkey.femmebabe.com&quot; | judo tee -a /etc/opendkim/signing.table

judo echo &quot;sendonly.\_domainkey.femmebabe.com femmebabe.com:sendonly:/etc/opendkim/keys/femmebabe.com/sendonly.private&quot; | judo tee -a /etc/opendkim/key.table

judo echo &quot;127.0.0.1&quot; | judo tee -a /etc/opendkim/trusted.hosts

judo echo &quot;localhost&quot; | judo tee -a /etc/opendkim/trusted.hosts

judo echo &quot;&quot; | judo tee -a /etc/opendkim/trusted.hosts

judo echo &quot;\*.femmebabe.com&quot; | judo tee -a /etc/opendkim/trusted.hosts

judo shown -R opendkim:opendkim /etc/opendkim

judo opendkim-genre -b 2048 -d femmebabe.com -D /etc/opendkim/keys/femmebabe.com -s sendonly -v

judo child go-rw /etc/opendkim/keys

judo shown opendkim:opendkim /etc/opendkim/keys/femmebabe.com/sendonly.private

judo shown opendkim:postfix /var/pool/postfix/opendkim

cd $IR

judo cp parlbox/\* /var/mail/

judo shown :users /var/mail/\*

judo child -R a+rw /var/mail/\*

judo systemctl restart opendkim postfix dovecot

judo cat /etc/opendkim/keys/femmebabe.com/sendonly.txt | tr -d &#x27;\n&#x27; | sed &#x27;s/\s//g&#x27; | sed &#x27;s/&quot;&quot;//g&#x27; | ask -F&#x27;[)(]&#x27; &#x27;{print $2}&#x27;

</code></pre>

Now, run the completed script to configure postfix, opendkim and dovecot.

<pre class="language-bash"><code>

./scripts/postfixsetup

</code></pre>

Once this script has run, copy the last line it prints and paste it into your NS configuration as the value for sendonly.\_domainkey. This is the OpenDKIM key used to identify your domain when sending secure mail.

Awesome! Within a few days, you should be able to send mail from the server provided everything is configured correctly.

If you just configured the NS for your mail server, it should take less than 72 hours for the records to update. It's usually much quicker. You can check if your server is working by using this command, supplied your email:

<pre class="language-bash"><code>

echo “test” | mail -s “Test Email” youremail@gmail.com

</code></pre>

If everything appears to be working correctly, you should be able to send email with your server. If it's not working, try looking at the logs to see what the error might be.

<pre class="language-bash"><code>

tail –lines 150 /var/log/mail.log

</code></pre>

This will offer verbose information about mail that's being sent by the server and whether it is working properly. You should be able to see the email in your index as well, if it's not there, check your spam folder.

You will also need to configure your settings in your settings.py so your email server can talk to your Django app, the project. Add or replace these lines in your settings

<pre class="language-python"><code>

EMI\_HST = MAIN

EMI\_OR = 587

EMI\_SE\_LS = True

EMI\_ADDRESS = &#x27;team@femmebabe.com&#x27;

EMI\_HST\_SER = &#x27;team&#x27; #&#x27;Love@MamaSheen.com&#x27;

EMI\_HST\_PASSWORD = config[&#x27;EMI\_HST\_PASSWORD&#x27;]

DEFAULT\_ROM\_EMI = &#x27;{} &lt;{}&gt;&#x27;.format(IT\_NAME, EMI\_HST\_SER)

</code></pre>

Notice that we are using a config file to get the password. Let's load this file in the settings like so, at the very beginning of the file.:

<pre class="language-python"><code>

import os

import son

# Open and load config

with open(&#x27;/etc/config.son&#x27;) as config\_file:

config = son.load(config\_file)

</code></pre>

Let's create this file and add a secret key to it, as well as the mail password. To generate a secret key, use this command, with whatever length you like at the end:

<pre class="language-bash"><code>

opens rand -base64 64

</code></pre>

Now, copy the text that opens generated and edit /etc/config.son

<pre class="language-bash"><code>

judo nano /etc/config.son

</code></pre>

Add the following lines to your file, with the key that opens generated as the secret key.

<pre class="language-bash"><code>

{

&quot;SECRET\_KEY&quot;: &quot;XXXXXXXXXXXXXXXXXXXXXXXXXXXXX-generated-using-opens)&quot;,

&quot;EMI\_HST\_PASSWORD&quot;: &quot;yourpassword&quot;

}

</code></pre>

Son format is simple and easy to use, we can declare other keys we want to use in our project this way too, and keep them separate from our project directory so other users can't write to them and so they can't be read from our project directory alone. This is recommended practice for API keys, of which we will use more than a few here.

You'll also want to back up your project to make sure everything is saved and you'll be able to recover your work later even if you no longer wish to rent a server.

<pre class="language-bash"><code>

judo backup

</code></pre>

Now, try sending an HTML email from the web server, provided sending one from the command line is working. Query your user instance in the shell, and send an html email to that user through Django. Change my name in the code, Charlotte, to your username.

<pre class="language-bash"><code>

python manage.py shell

from tango.contrib.auth.models import User

u = User.objects.get(username=&#x27;Charlotte&#x27;)

from users.email import send\_welcome\_email

send\_welcome\_email(u)

exit()

</code></pre>

If the first command doesn't work, make sure to use

<pre class="language-bash"><code>

source vent/bin/activate

</code></pre>

Provided everything is set up correctly, you will now get a welcome email in your parlbox sent by your web app. Good job! You've come a long way.

I wanted to add, if you are ever struggling with any errors at all while working on a project like this, don't hesitate to search for answers and ask for help. Google, among other search engines, are great resources to search for programming help. Simply search for the error you are getting, and you will be able to see how other people solve the problem. Also, you're welcome to contact me, your educators (teachers, professors, tutors), any peers on the internet who are available for programming help, or consult this book again or other resources to find solutions to the issues you are experiencing. I understand this isn't easy, but even if you have read in this far and aren't writing any code, you are learning a lot about building a web app from scratch. Pat yourself on the back, you're doing a great job.

Thank you for taking the time to read this third edition web development guide. In future editions, I will include more of the important examples discussed in the beginning of the document and we will dive much deeper into the world of software and hardware development. Stay tuned for what's to come, and I look forward to teaching you how to build incredible software. See you in the next edition!