MeHealthTech Dev Guide

By: Kevin Lee, Junior Developer

17 August 2021

Table of Contents

App Layout

The is built on Python 3.8 and Django 3.2.4. This means it follows the Model-View-Template (MVT) format, where models are the schemas for database objects, templates are the frontend HTML pages, and views contain most backend functionality and communicate between the database and the frontend. The app is hosted using Azure App Services, the app’s static files are stored on and served from an Azure Storage Account, and the database used is an AWS RDS instance running PostgreSQL.

Modules

There are 11 main Python modules stored in the source code of the app. Here's a breakdown of the modules contained in the root project folder:

health

- Project folder contains the app's settings.
    - settings.py defines general settings
    - development.py and production.py set the allowed hosts, DEBUG and domain settings, production.py also sets CSRF and Cookie settings
    - development_db_settings.py sets the app to store data in a local SQLite database
    - production_db_settings.py sets the app to store data in an external database (currently set to use AWS RDS PostgreSQL)
    - development_storage_settings.py sets the app to store static files and media in local directories
    - production_storage_settings.py sets the app to store static files and media in an external storage system (currently set to use Azure Storage Account)
    - development.py and development_db_settings.py will be active if DJANGO_DEVELOPMENT (environment variable)/DEBUG (setting) are set to True, while production.py and production_db_settings.py will be active otherwise

    - development_storage_settings.py will be active if DJANGO_EXT_STORAGE (environment variable) is set to False and production_storage_settings.py will be active otherwise
- manage.py can be used to run Django commands in-console (does essentially the same thing as django-admin as seen here)
- All main routing paths can be found in urls.py
- Also contains modified templates for default views such as login and password reset

home

- This acts as the index module. It contains functionalities that are accessible to users of all groups, such as the home, about us and error pages.

accounts

- Contains all our User Models, including the custom Base User and Proxy Models for Patients and Doctors
- Also contains Form templates for registration and editing profiles (login and password reset are Django built-in, but styled in the template)

book

- This app handles the booking process and most functions involving appointment management, everything from when a Doctor decides to open a timeslot for potential appointments to when a Patient books an available slot.
- Also contains our model for Appointments, as well as the Form templates

doctordashboard

- Pertains to all the main functions available to Doctor users, which are all accessible from the dashboard page
- Very intertwined with the book module and makes lots of additions and changes to Appointment objects.

appointment

- Takes care of appointment files and details such as notes and prescriptions.
- Contains models for appointment details and files, and nowhere near as big as book or account
- Note: Does not contain the Appointment model, which may cause confusion
  - This module was created later into development to hold models that are auxiliary to the main Appointment model and the associated views, while the main model is defined in the book module and is referenced in almost every module. However, most of its related views are located in book or doctordashboard.

patientdashboard

- Much simpler than the doctordashboard module, as this serves to only view Appointment details as a Patient and the upload/get/delete processes for images.

scheduledreminders

- Self explanatory, handles the sending of scheduled reminders.
- Regularly checks database using APScheduler for any upcoming appointments (15 minutes ahead) and uses Django send_mail to send email and SMS notifications.

graph

- Everything pertaining to MS Graph connections and API usage, including authentication using OAuth 2.0, getting user profile info and creating events on Outlook Calendar.
- Most functions are from MS Graph documentation found here.

maps

- Helper app that handles usage of the Google Maps REST API
- Specifically, Places API, Embed Map API and Geocode API

customstorage

- Very basic module which allows for differentiating between public and private Azure Storage objects, taken from here

External APIs

The app makes use of the following APIs:

SignalWire

For sending automated SMS messages to Patients and Doctors regarding appointment bookings, reminders, and cancellations

Microsoft Graph

Gives Patients the option to sign into their Microsoft accounts, which will allow events to be created on third-party Calendar apps, such as Outlook or Gmail, whenever they book an appointment.

Google Maps Places

Used when searching for nearby pharmacies based on Patient's given address and postal code.

App Setup

App setup includes all steps to be taken before even thinking about running the app anywhere, such as registration of appropriate services and APIs, creating necessary resources, and setting environment variables.

CLI Usage

If you prefer using a Command Line Interface to perform configurations, refer to the following links to install and set up the required CLIs:

- Azure CLI Latest Version
    - https://docs.microsoft.com/en-us/cli/azure/install-azure-cli
    - https://docs.microsoft.com/en-us/cli/azure/authenticate-azure-cli
    - To use the Azure CLI, you must run “az login” first and log into your Microsoft Account.
- AWS CLI Latest Version
    - https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html
    - https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-quickstart.html
    - To use the AWS CLI, you must run “aws configure” first, and enter your Access Key and Secret Access Key. These credentials can be created at the IAM Console. This process is detailed under *Setting up Your AWS RDS Instance - Using the AWS CLI*

Setting up Your Azure App Service

To set up a new Azure App Service, you will need a Microsoft Azure Account.

Although almost all actions involving Azure may be performed through either the Portal or the CLI, we strongly suggest completing this setup process and all other App Service configurations strictly through the Portal for the purpose of simplicity.

Using the Azure Portal

1. Sign in to the Azure Portal
2. In the homepage, click “Create a Resource”
3. Search for “Web App” and click “Create”
4. Create or Select a Subscription
5. Create or Select a Resource Group
    a. This will be the root group for your Azure Cloud resources
6. Give your app a name
7. Select “Python 3.8” as the runtime stack
8. Select an appropriate region for your app to run in
9. If given an option, select the F1 Free SKU Tier for now, you will be able to scale up later
10. Click “Review + Create”

Once you are done setting up the App Service, you will need to add the app’s domain to the ALLOWED_HOSTS list in production.py in the health module, the same module as settings.py. Be sure only to add the domain and not the protocol or any URL extensions.

Setting up Your Azure Storage Account

To set up a new Azure Storage Account, you will need a [Microsoft Azure Account].

Using the Azure Portal

1. Sign in to the Azure Portal
2. In the homepage, click “Create a Resource”
3. Search for “Storage Account” and click “Create”
4. Select the same Subscription and Resource Group used for your App Service
5. Give your Storage Account a name
6. Select a region for your Storage Account (this can be the same region as your App Service, but it is not mandatory)
7. Make sure Geo-redundant storage (GRS) is selected for Redundancy
8. Click “Review + Create”

Using the Azure CLI

1. Login with “az login”
2. Run “az storage account create -n <account-name> -g <resource-group-name> -l <region e.g., eastus> --sku Standard_GRS”

Setting up the AWS CLI (optional)

If you are planning on using the AWS CLI, you will need a set of valid credentials (Access Key and Secret Access Key). If you do not have these, follow these steps to create a pair:

1. Go to the AWS IAM Console
2. Sign in as the root user using your AWS account email and password
3. There are two ways to proceed:
    a. Recommended: create a new IAM User Group and IAM User with AWS RDS Permissions, then create credentials for the new IAM User
        i. Navigate to “User Groups” and click “Create Group”
        ii. Name the group RDS-admin or something similar
        iii. Attach the AmazonRDSFullAccess Permission by checking its box
        iv. Click “Create Group”
        v. Navigate to the “Users” page and click “Add User”
        vi. Give the user a username and check “Programmatic access” (You can also check “AWS Management Console access” and set a password if you wish to use this account in the console instead of the root user)
        vii. Click “Next”
        viii. Add the new User to the User Group created in Steps i-iv
        ix. Create the User
        x. Back in the “Users” page, the new User should show up, click on its name
        xi. Click the “Security credentials” tab, then click “Create access key”

xii. You will see a pair of credentials appear, click “Show” on the Secret Key and store it somewhere safe. This is the only time you will be able to see the value in the Console. If you lose or forget it, you will have to create a new set of keys and deactivate this set.

xiii. You can now use the Access and Secret Access Keys which were generated to use the AWS CLI.

b. Less secure but quicker option: Create access credentials for the root user

i. On the right side, under “Quick Links”, click “My access key”

ii. Click “Create New Access Key”

iii. You can show the two key values now and download a file containing them. If you lose or forget the keys, delete them from this page and generate a new pair.

Setting up Your AWS RDS Instance

To set up a new AWS RDS Instance, you will need an AWS Account

Using the AWS Console

1. Log in to your AWS Account
2. Go to your Management Console
3. Search for the “RDS” Service in the search bar and click on it. You will be redirected to the AWS RDS Management Console
4. On your dashboard, click “Create database”
5. Select PostgreSQL and select version 12.5-R1
6. Select either “Production” or “Dev/Test”, depending on your needs. You can fine tune the DB’s settings later.
7. Give your DB an identifier (Note: the DB identifier is different from the DB name, AWS will default the name to “postgres”. This can only be set at creation using the CLI)
8. Create credentials for your master user (username and password)
9. If you are using the AWS Free Trial, be sure to select “Burstable classes” and “db.t3.micro” under “DB instance class”
10. Under “Storage”, set your required storage settings
    a. If you are using the AWS Free Trial, be sure to select “General Storage (SSD)” and set “Allocated storage” to 20GiB
11. Under “Connectivity”, ensure that “Public access” is set to “No”
12. Under “Database authentication” ensure that “Password authentication” is selected
13. Click Create database

Using the AWS CLI

Use this option only if you know the exact specifications of the DB required.

1. Use “aws configure” and enter your Access Key and Secret Access Key
2. Run “aws rds create-db-instance --db-instance-identifier <identifier> --db-name <dbname> --engine postgres --engine-version 12.5 --no-publicly-accessible --port 5432 --no-multi-az --db-instance-class <instance-class> --master-username <masterusername> --master-password <masterpassword> --allocated-storage 20 --storage-type standard
    a. --db-instance-class: see here for a list of all instance classes. Note: use db.t3.micro for Free Trial
    b. Full docs here

Setting up Your SignalWire Account

1. Go to https://m.signalwire.com/signups/new?s=1 to register a new SignalWire Account if you do not have one already
2. Once logged in, create a new Space
3. In that Space, create a new Project
4. You can now load the Project dashboard to show your Project ID and Space URL
5. In the API tab, you can create a new Messaging API Key

Setting up Microsoft Graph API through Azure Active Directory

To set up Microsoft Graph for your app, you will need a Microsoft Azure Account.

Using the Azure Portal

1. Log in to Azure Portal
2. Click on “Azure Active Directory”
    a. If the account you are using does not belong to an Active Directory or does not have the required permissions for any of these steps, you may have to use a different Microsoft account to set up Graph. This is fine, as the API registration is separate from the Web Application configuration, although it is better to have both on the same account for ease of access.
3. In the left tab, click “App registrations”, then click “+ New Registration”
4. Name your app registration
5. Set Supported account types to “Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)”
6. Click “Register”
7. In your new app registration’s Overview page, click on “Add a Redirect URI” under “Essentials”
8. Click “+ Add a platform” and select “Web”
9. The Graph Redirect URL for this specific app is “/graph/callback”, so your Redirect URI would look like “https://your-domain-name.com/graph/callback”, or “http://127.0.0.1:8000/graph/callback” for dev/test. You can add multiple URIs later.

a. NOTE: Azure only allows HTTP to be used in a Redirect URI when the domain is “localhost” and requires HTTPS otherwise. So, if you wanted to use “127.0.0.1:8000” in the Redirect URI, it would need the “https://” protocol. However, Django only supports HTTP when running on Dev/Test server, so trying to access “**https**://127.0.0.1:8000/graph/callback” will produce a 404 error and the API call will not be made. On the other hand, accessing “**http**://127.0.0.1:8000/graph/callback” will result in the API call being rejected, as it is impossible to input **http**:// with anything other than “localhost” in Azure, so the URI would not match any of the Redirect URIs specified in the app registration.
b. An easy workaround for this when testing Graph functionality in Dev/Test is to use the “python manage.py runserver localhost:8000” command instead of the default “python manage.py runserver” command. This will run the server on the localhost domain instead of the 127.0.0.1 domain, and so when you access the “/graph/callback” URL, the URI will match and the API call will be validated.
c. This is not a problem in production, as the protocol used for domains outside of Dev/Test should always be HTTPS, so they should always be able to match the URIs in the app registration.

10. Navigate to the “API permissions” tab on the left menu
11. Click “+ Add a permission” and select “Microsoft Graph”
12. In the Search Bar, type "User.Read"
13. Scroll down to the "User" tab, click it and check "User.Read"

14. Click "Add Permissions"

15. Go to the “Certificates & secrets” tab (You can do this and step 16 later in the guide)

16. Click “New client secret”, give the secret a name and expiry window, then click “Add”

    a. The secret value will only be shown once, at time of creation, so copy it and save it somewhere safe, you will need it later as an environment variable. Should something happen to it, come back to this page, delete the current secret, and generate a new one.

Using the Azure CLI

1. Log in to the Azure CLI using “az login”
2. Run "az ad app create --display-name <name>". The output will show your new app’s Client ID. Copy this and keep it ready for the next step.
3. To add Redirect URI’s, run "az ad app update --id <Client ID from last step> --reply-urls <desired-redirect-url1> <desired- redirect -url2> [...] <desired- redirect -urlN>"
   a. Please read Step 9a-c in the *Using the Azure Portal* section on the previous page
4. To add API Permissions, run "az ad app permissions add --id <Client ID> --api 00000003-0000-0000-c000-000000000000 --api-permissions df021288-bdef-4463-88db-98f22de89214"
   a. 00000003-0000-0000-c000-000000000000 is the API Resource ID for Microsoft Graph. This can be found by running `az ad sp list --query "[?appDisplayName=='Microsoft Graph'].appId | [0]" --all`
   b. df021288-bdef-4463-88db-98f22de89214 is the API Permission OAuth2Permissions ID for the User.Read Permission. This can be found by running `az ad sp show --id 00000003-0000-0000-c000-000000000000 --query "oauth2Permissions[?value=='User.Read'].{Value:value, Id:id}" --output table`
5. To generate a client secret (Once again, this can be done later but the steps are provided anyway)
   a. Run "az ad app credential reset --id <MS_GRAPH_CLIENT_ID>"
   b. Take note of the newly generated key, under the "password" key, as it will never be shown again, and you will need it when setting environment variables later.

Setting up Google Maps Places API

1. To start, follow this guide to create a new project in the Google Maps Platform.
2. To enable APIs, go to the APIs and Services tab in the left menu and select "Library"
    a. Search for the following APIs and enable them:
        i. Maps Embed API
        ii. Geocoding API
        iii. Places API
    b. After enabling one API, you will be redirected to the Google Maps Platform API page, where you can immediately enable the other two APIs
3. Go to the Credentials tab from the left menu
4. Click "+ Create Credentials" near the top of the screen
    a. You will need to create two API Keys here. One will be used for the Geocoding and Places APIs while the other will be used only for the Maps Embed API
        i. This is because the first two cost money per call, so we must set restrictions on the IP addresses using the key to prevent people from stealing from our account. On the other hand, Maps Embed API has no usage costs, but tends to not work when you set IP address or HTTP referrer restrictions on it.
        ii. The solution is to generate two keys: one with heavy IP address restrictions that uses the paid APIs, and the other has no referrer restrictions (anyone can use it) but it is restricted to only call the Maps Embed API, which is free.

Restricting your Google Maps API Keys

1. From the Credentials tab, click the key you would like to restrict
2. To restrict the key via IP Addresses:
    a. Check the “IP Addresses” radio button under “Application restrictions”
    b. Add your own IP Address and any other local machines used for Test/Dev
    c. Add Outbound IP Addresses of your Azure App Service (see next section)
3. To restrict the key via API usage:
    a. Select the APIs you wish to restrict the key to under “API restrictions”

Find your Azure App Service Outbound IP Addresses

To set up Microsoft Graph for your app, you will need a Microsoft Azure Account and an Azure Web App Service set up.

Using the Azure Portal

1. Log in to Azure Portal
2. Navigate to your App Service’s homepage
3. Click the “Properties” tab on the left menu
4. There will be a comma-separated list of addresses under “Outbound IP addresses”

Using the Azure CLI

1. Log in to the Azure CLI using “az login”
2. Run `az webapp list --query "[?name=='YOUR_APP_NAME'].outboundIpAddresses"`
3. A comma-separated list of IP Addresses will be outputted

Setting the App's Environment Variables

***********NOTE*************

FOR ALL VARIABLES MARKED WITH AN ***IMPORTANT*** TAG:

DO NOT SHARE THESE VALUES NOR COMMIT THEM TO SOURCE CONTROL.

AS A SECURITY PRECAUTION, IF THERE IS EVER EVEN THE SLIGHTEST POSSIBILITY THAT A VALUE HAS BEEN COMPROMISED OR LOST, GENERATE A NEW ONE IMMEDIATELY FOLLOWING THE STEPS PROVIDED UNDER THE VARIABLE'S ENTRY IN THIS DOCUMENT AND REPLACE THE OLD VALUE.

WHENEVER ONE OF THESE VALUES IS CHANGED, IT MUST ALSO BE CHANGED WITHIN:

1. AZURE APP SERVICE CONFIGURATION
2. GITHUB/BITBUCKET REPOSITORY SECRETS

TO ENSURE FUNCTIONAL DEPLOYMENT AND RUNNING

****

To set all these environment variables, you will need:

- An email account to send automated emails from.
- A Microsoft Azure Account
- An AWS Account
- A SignalWire Account
- A Google Cloud Account

1. SECRET_KEY ***IMPORTANT***

- This is a random string that affects many security functions in the application, such as protecting of signed values, JSON object signing, token generation, password salt creation, form and CSRF protection, etc.
- IF VALUE IS COMPROMISED, GENERATE A NEW KEY IMMEDIATELY
- Useful tool for generating a strong SECRET_KEY: https://miniwebtool.com/django-secret-key-generator/
- In depth explanations on SECRET_KEY function here and here

2. EMAIL_USER

- The email address used to send out automated email reminders and confirmations
  - Account confirmation, Password or email reset, booking confirmations/reminders, cancellations, etc.
- As of 08/16/2021, the app uses an email account registered with Gmail.

3. EMAIL_PASS ***IMPORTANT***

- The password to the email account linked to EMAIL_USER
- IF VALUE IS COMPROMISED OR FORGOTTEN, CHANGE PASSWORD THROUGH THE EMAIL SERVICE USED TO REGISTER THE EMAIL_USER ACCOUNT

4. AZURE_ACCOUNT_NAME

- The name given to the Azure Storage Account used to store and serve static and media files.

- Can be found either through the Azure Portal or using the Azure CLI command "az storage account list" (You will have to sign into your Azure Account using "az login" first)
    - Details on Azure CLI storage account commands can be found here:

5. AZURE_ACCOUNT_KEY   ***IMPORTANT***

- IF VALUE IS COMPROMISED OR LOST, ROTATE ACCESS KEY BY USING ONE OF THE FOLLOWING METHODS, THEN REPLACE THIS VARIABLE WITH NEW KEY, OLD KEY WILL BE RENDERED INVALID:
    - Log into Azure Portal, navigate to the storage account in question, Click "Access Keys" and then "Rotate Key" on both key1 and key2.
    - OR: Log into Azure CLI using "az login", then run command "az storage account keys renew --account-name <account name> --key <primary or secondary> --resource-group <resource group>" see more details here
- Serves as a root password for accessing your Azure Storage Account, allowing for both anonymous and private access, generation of SAS tokens and temporary URLS, etc.

6. AZURE_STATIC_CONTAINER

- Name of the container located inside the Azure Storage Account which holds static files, typically set to "static"

7. AZURE_MEDIA_CONTAINER

- Name of the container located inside the Azure Storage Account which holds media files, typically set to "media"

8. RDS_ENDPOINT

- The endpoint of your AWS RDS PostreSQL Database Instance, used to connect the app to the DB
- This value can be found using one of the following methods:
    - Log in to the AWS RDS Management Console, select the DB instance you wish to access, and under "Connectivity & Security", you will be able to copy the endpoint.
- OR: Log in to the AWS CLI by running aws configure and using your AWS Account's Access Key and Secret Access Key (these are only viewable once when you create your account so keep them somewhere safe!)
    - Run the "aws rds describe-db-instances" command
    - This will display all your DB instance information, including name, endpoint address, master username, and port.

9. RDS_NAME

- The name given to your AWS RDS DB Instance
- Can be found using the same methods shown under RDS_ENDPOINT

10. RDS_USER

- The username given to your AWS RDS DB Instance master user.
- Can be found using the same methods shown under RDS_ENDPOINT

11. RDS_PASSWORD ***IMPORTANT***

- This is the password for your master user, which you set when you create your DB instance. It cannot be shown after creation, so if it is lost, simply reset it with the steps above.
- IF THIS VALUE IS COMPROMISED OR LOST, CHANGE IT IMMEDIATELY USING ONE OF THE FOLLOWING METHODS, THEN REPLACE THE OLD VALUE:
  - To change your DB's master user password from the AWS Management Console, see this link
  - To do so in the AWS CLI
    - Authenticate your account with "aws configure" if you have not done so already
    - Run "aws rds modify-db-instance --db-instance-identifier <RDS_NAME> --master-user-password <new-password>"

12. RDS_PORT

- The port which your DB instance is connected to.
- Can be found using the same methods shown under RDS_ENDPOINT, but for PostreSQL, it is always 5432.

13. DJANGO_DEVELOPMENT

- MAKE SURE THIS IS ALWAYS FALSE IN PRODUCTION

- Boolean value which determines if app is to be run on Development settings or not.
- Set either to True or False
    - If set to True
    - App will run on DEBUG mode
    - CURRENT_DOMAIN will be set to "127.0.0.1"
    - CURRENT_SITE will be set to http://127.0.0.1:8000
- If set to False **(Always False for production)**
    - DEBUG mode will be turned off
    - CURRENT_DOMAIN will be set to "health-tech.azurewebsites.net"
    - CURRENT_SITE will be set to "https://health-tech.azurewebsites.net"
    - Session and CSRF cookie settings will be active.

14. DJANGO_EXT_DB

- Boolean value which determines if app will run using external AWS RDS PostgreSQL DB or not.
- Set either to True or False
- If set to True **(Always True for production),** app will attempt to connect to the PostgreSQL DB using credentials provided in the RDS_ prefixed environment variables through the psycopg2 backend engine
- If set to False, app will run using a local SQLite3 DB and

15. DJANGO_EXT_STORAGE

- Boolean value which determines if app will run using external Azure Storage or not.
- Set either to True or False
- If set to True **(Always True for production),** app will attempt to connect to the Azure Storage Account using credentials provided in the AZURE_ prefixed environment variables.
- If set to False, app will run by using static files located in the local STATIC_URL directory and save and serve files in the local MEDIA_URL directory

16. MS_TEAMS_TEMP_LINK_1

- Constant value but kept in environment variables to hide our methods of connecting to MS Teams calls.
- Should always be set to https://teams.microsoft.com/l/meetup-join/19%3ameeting_
- "TEMP" does not stand for "temporary", but in fact "template"
- A valid MS Teams Meeting URL consists of 3 parts:
  - Prefix: which is the same for any call URL (this variable)
  - Meeting ID: a 48 character-long, alphanumeric string, which is the only unique part of the URL
  - Suffix: which provides the ID of the account which creates the call and other constants (MS_TEAMS_TEMP_LINK_2)

17. MS_TEAMS_TEMP_LINK_2

- Similar to MS_TEAMS_TEMP_LINK_1, but will need to be changed if a new MS Teams account is used to generate calls.
- Can be found by copying the section of a MS Teams Meeting Link following the unique 48 character-long, alphanumeric ID
- Ex: (MS_TEAMS_TEMP_LINK_1) https://teams.microsoft.com/l/meetup-join/19%3ameeting_(48 CHAR-LONG MEETING_ID)%40thread.v2/0?context=%7b%22Tid%22%3a%22db3d29e3-1486-4655-b04d-216b57eff244%22%2c%22Oid%22%3a%220817afe2-5e78-44c3-90a9-62c3eb802fdd%22%7d(MS_TEAMS_TEMP_LINK_2)
    - Together, these 3 strings will form a valid MS Teams Meeting Link.

18. MS_TEAMS_MEETING_ID_LENGTH

- Set always to 48
- Hidden from users so they cannot recreate links easily

19. MS_GRAPH_CLIENT_ID

- The Client ID generated for your app when it is registered on Azure.
- This can be found by either:
    - Logging into the Azure Portal, navigating to Azure Active Directory, App Registrations, then your app instance.
    - OR: After logging into Azure CLI, running "az ad app list". The value will be found in the first returned JSON object, under the key "appId".

20. MS_GRAPH_CLIENT_SECRET *IMPORTANT*

- A secret key that can be generated after registering your app with the Microsoft Graph API.
- IF THIS VALUE IS COMPROMISED OR LOST, RESET IT USING ONE OF THE FOLLOWING METHODS:
  - Log in to Azure Portal
    - Navigate to Azure Active directory
    - Click on App Registrations on the left tab
    - Click the app in question
    - Click "Certificates and Secrets" on the left tab
    - Delete the current Client Secret
    - Click "+ New client secret" and set expiry + description.
  - OR: Log in to Azure CLI
    - Run "az ad app credential reset --id <MS_GRAPH_CLIENT_ID>"
    - Take note of the newly generated key, under the "password" key. It will never appear again.
  - update the MS_GRAPH_CLIENT_SECRET environment variable.
- To generate a new key, follow the same steps outlined above

21. MS_GRAPH_REDIRECT_URL

- The URL on your app's site that makes the request to MS Graph API
- Used to verify API calls
- Should always be set to “/graph/callback” unless if the URL is changed in the code

22. SIGNALWIRE_PROJECT

- Your SignalWire Project's ID
- Can be found by logging into your SignalWire account and entering your project's dashboard

23. SIGNALWIRE_TOKEN *IMPORTANT*

- Your SignalWire API Token
- Can be found by logging into your SignalWire account and entering your project's API page from the left-hand menu and clicking "Show" on the key's value
- If you do not have an existing key, click "+ New" and give your token a name.
- Your key will only need Messaging permissions
- IF THIS VALUE IS COMPROMISED OR LOST, DELETE THE OLD KEY FROM THE SAME PAGE DESCRIBED ABOVE AND GENERATE A NEW ONE FOLLOWING THE SAME STEPS

24. SIGNALWIRE_SPACE_URL

- The URL to your SignalWire space
- Can be found by logging into your SignalWire account and entering your project's dashboard

25. SIGNALWIRE_PHONE_NUMBER

- A phone number you have purchased and added to your SignalWire account
- Can be found by logging into your SignalWire account and entering your project's Phone Numbers page
    - If you do not have a phone number added to your account, click "+ New"

    - You will only need a Local Number and at least $5 in added credit to your account in order to activate these capabilities

26. GOOGLE_MAPS_API_KEY ***IMPORTANT***

- Key for using Google Maps Geocoding and Places API
- To generate a key, follow the steps in this link
- IF THIS VALUE IS COMPROMISED OR LOST, DELETE THE OLD KEY AND GENERATE A NEW ONE BY FOLLOWING THE PREVIOUS STEP

27. GOOGLE_MAPS_EMBED_API_KEY ***IMPORTANT***

- Key for using Google Maps Embed API
- All steps from GOOGLE_MAPS_API_KEY apply

Loading Environment Variables to Azure

Using the Azure Portal

1. Log in to the Azure Portal
2. In your App Service, select “Configuration” in the left menu
3. Add each environment variable’s key and value one-by-one by clicking “+ New application setting”
4. Click “Save” when done

Using the Azure CLI

1. Log in to the Azure CLI with “az login”
2. Run “az webapp config appsettings set -n <app-name> -g <resource-group> “@appSettings.json”
    a. The appSettings.json file is automatically generated in your local project directory. **ONLY IN DEV (WHEN DJANGO_DEVELOPMENT IS TRUE PLEASE MAKE SURE IT IS NEVER TRUE IN PRODUCTION OR YOU WILL HAVE TO DELETE AND RE-GENERATE ALL ENVIRONMENT VARIABLES).**
    b. **Please review the code which writes the file before changing any environment variables manually (health.settings line 223)**

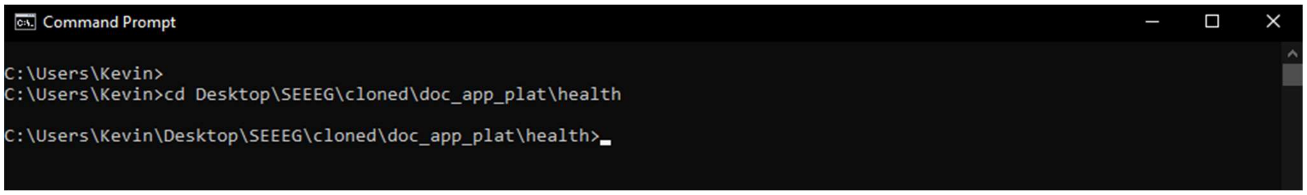
Wrap-Up

If you have followed all these steps, your app should now be properly configured and ready for you to browse. Try going to the app’s URL or clicking “Browse” on the Azure App Service page.

Running the App on a Local Machine

In order to get this app up and running locally on your machine, please take the following steps:

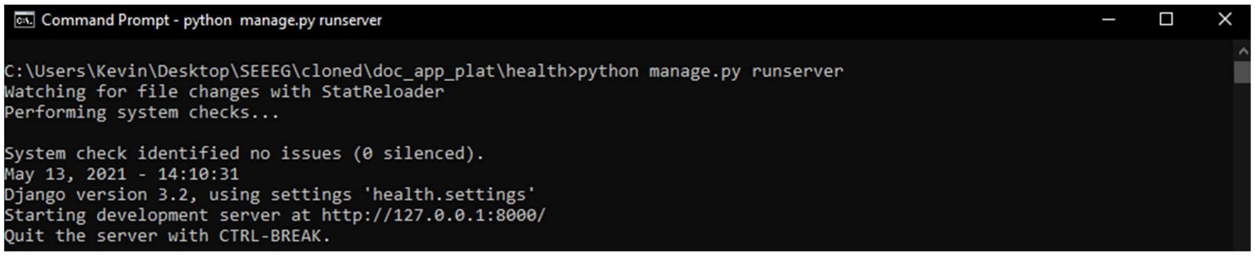
1. Ensure you are running at least Python version 3.8
2. Download all source files from the repository to a virtual environment directory (recommended) or any local directory
    a. ex. 'C:\Users\Name\Documents\Projects\This_Repo'
3. Install the required Python modules if they are not yet installed on your machine.
    a. If on Linux, run sudo apt-get install --reinstall libpq-dev
    b. Note: this can be done by running pip install -r requirements.txt
4. Set all the required environment variables in a .env file, located in the 'health' module, the same folder as settings.py.
5. Open a command line interface (i.e. cmd, powershell, etc) and navigate to the outer 'health' directory found in the cloned repository's folder



```
C:\Users\Kevin>
C:\Users\Kevin>cd Desktop\SEEEG\cloned\doc_app_plat\health

C:\Users\Kevin\Desktop\SEEEG\cloned\doc_app_plat\health>_
```

6. Once in the 'health' folder, run the following commands in the command line:
    a. python manage.py migrate
    b. python manage.py runserver
    c. If successful, the following message should appear:


Command Prompt - python manage.py runserver

```
C:\Users\Kevin\Desktop\SEEEG\cloned\doc_app_plat\health>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
May 13, 2021 - 14:10:31
Django version 3.2, using settings 'health.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

7. In a browser, open the following link: http://127.0.0.1:8000/

8. You're all set! To stop the server, press Ctrl + C

App Maintenance

Troubleshooting

When troubleshooting, always make sure DJANGO_DEVELOPMENT is set to “True” so that error logs will appear to give more detail on the nature and cause of the error. If the error is not showing up in the dev/test server, it is suggested to run the action that is causing the error both when the DJANGO_EXT_DB environment variable is set to “False” **and** when it is set to “True”. This is because it is possible for the local and production databases to not be synced, leading to some errors only appearing when one database is active and not the other. This happens when migrations are made for one database but not the other. To prevent this, always run “python manage.py makemigrations” (this only has to be run once), then “python manage.py migrate” once when DJANGO_EXT_DB is “False” and another time when DJANGO_EXT_DB is “True”.

Error 400 when loading any page

An HTTP Error Code 400 is known as a Bad Request error. This usually represents an error in the entered URL, and in most cases, the URL you are trying to access is not one of the allowed hosts of the app.

To fix this error, check if all domains used for your app have been specified in the ALLOWED_HOSTS variable. There is an ALLOWED_HOSTS variable in both the production.py file and the development.py file, so change the variable in the settings for whichever environment the error shows up in (dev/test or production). These are in the same folder that holds the settings.py file.

Error 403 when loading app on production

An HTTP Error Code 403 is known as a Forbidden error. It is outputted when the server receives a request, but then denies it. If this is happening when trying to load any page of the app in production, then the error may be directly linked to your application's resources being unavailable. The most common cause for this is exceeding your cloud compute limits in your App Service Plan and it can be remedied by simply scaling up your application to allocate more resources to its running. However, there are times when the error is not your fault, and the servers are simply unable to host your application at that time, although this usually only happens when using the F1 Free Tier Azure App Service Plan.

Error 403 when submitting a form

When submitting a form on the web, it is necessary for sites to include a CSRF (Cross-Site Request Forgery) encryption cookie along with the data to protect the information being sent. This prevents the data from being intercepted and the request from being hijacked by potential attackers. Without a valid CSRF token, it is possible for attackers to gain access to another user's authorization and permissions, allowing them to perform unauthorized actions using their data without them knowing. Now that you have the basics of CSRF, you can understand why the Error Code 403 may be showing up. Here are the main two reasons:

1. There is an invalid or no CSRF cookie in a form being submitted
2. There is a valid CSRF cookie, but the user does not have cookies enabled in their browser

Error 404 when trying to test Microsoft Graph functionality

An Error Code 404 is a Page Not Found error. If it appears when attempting to sign in with Microsoft Graph, we can narrow it down to one common cause, being the Azure App Registration Redirect URIs. As described in Setting up Microsoft Graph API through Azure Active Directory, an error code 404 will only appear if the callback URI you are trying to access matches any of the Redirect URIs specified in the Azure App Registration. If this is happening in dev/test, then make sure http://localhost:8000/graph/callback is added as a Redirect URI and you are running the server with the command “python manage.py runserver --localhost”. If it is occurring in production, then simply make sure the correct URI with your production domain is specified as a Redirect URI.

Error 404 when trying to load another page

Receiving an Error Code 404 otherwise is most likely a sign that you may have entered the wrong URL, you have not yet included the URL in one of your urls.py modules, or the URL you have entered has not been configured with the proper acceptable parameters.

Error 500 when trying to load a page

An Error Code 500 is an Internal Server error. This represents there being some error in the code that was pushed to the server for running and prevents the app from running at all.