

**Capstone Project Phase B**

**NODUS - Childhood ADHD Symptom Detection with Possible Solutions**

**Maintenance Guide**

safwanhalabi60@gmail.com

bish.150b@gmail.com

# Table of Contents

[**Table of Contents 2**](#_3fzqgyy8jege)

[**Introduction 4**](#_mmbi8e2o5wpc)

[**1. System Architecture Overview 4**](#_xkumwob5ktom)

[1.1 System Components 4](#_eycyjyag6d6x)

[1.2 System Flow 6](#_tsglvd3oiwxr)

[**2. Software Dependencies 8**](#_bas5t09m21sk)

[2.1 Backend Dependencies 8](#_qw51vd2cftxk)

[2.2 Environment Management 9](#_hhwxkit3jq2b)

[**3. Code Review 10**](#_h5frjcpjn3ev)

[3.1 manage.py file 10](#_ehiljv71rdoa)

[3.2 db\_connection.py file 10](#_pbtvcrdr0o7i)

[3.3 FinalProject folder 10](#_sh6v1jysrw5n)

[3.3.1 settings.py file 11](#_8ojkmiooeksw)

[3.3.2 \_\_init\_\_.py file 12](#_wdokys16digm)

[3.3.3 wsgi.py file 12](#_wgota9rgsjej)

[3.3.4 asgi.py file 13](#_ojenso24ardl)

[3.3.5 urls.py file 13](#_33p7bp1flcbc)

[3.4 ADHD folder 13](#_ui0aft8c2pa0)

[3.4.1 \_\_pycache\_\_ folder 14](#_4pkjdhcod5er)

[3.4.2 migrations folder 14](#_7pa2zgsm8u8t)

[3.4.3 signatures folder 14](#_8usqdt41hpvr)

[3.4.4 static folder 14](#_vtw1ux9vlzzo)

[3.4.5 templates folder 15](#_jy9l0ergnmen)

[3.4.6 temporary\_files folder 15](#_w14n0zfwplhy)

[3.4.7 \_\_init\_\_.py file 15](#_fq20el58ql8h)

[3.4.8 admin.py file 15](#_5lx2bxdrp11j)

[3.4.9 apps.py file 15](#_lltfyywvckmx)

[3.4.10 models.py file 16](#_yk32lolnpwcd)

[3.4.11 tests.py file 16](#_1yly6y1vdagq)

[3.4.12 urls.py file 16](#_mpk5qx2brrr6)

[3.4.13 views.py file 17](#_hbkd70ni3iej)

[**4. Back-end Code 18**](#_lgey1q9pzoo7)

[4.1 Deep Learning model 18](#_lli3x5c8a8iz)

[4.2 Render Functions 19](#_jd2jyu8bhchf)

[4.3 Data Processing Functions 20](#_xh0mky7ykaj2)

[4.4 Helper Functions 22](#_iabsepsw3ss2)

[4.5 Database Functions 24](#_3mgqp1b87j3p)

[**5. Deployment 25**](#_d6orv8xgop2l)

[5.1 Overview 25](#_3ixp9ekewhq4)

[5.2 Render dashboard 25](#_oijekidn3zlw)

[5.3 The control panel 26](#_rrsca1ewdnpw)

[5.3.1 Events 26](#_vvjgtnmmn8v5)

[5.3.2 Logs 26](#_joma5dza3qa5)

[5.3.3 Environment 27](#_vhr7l17seqs9)

[5.3.4 Shell 28](#_hp8ltdy5vzjq)

[5.3.5 Metrics 29](#_bdr4z6l4icld)

[5.3.6 Settings 29](#_kg22fpvh521f)

[**External Links and Documentation 32**](#_kwp7hnum2sfs)

# 

# 

# 

# 

# 

# 

# 

# Introduction

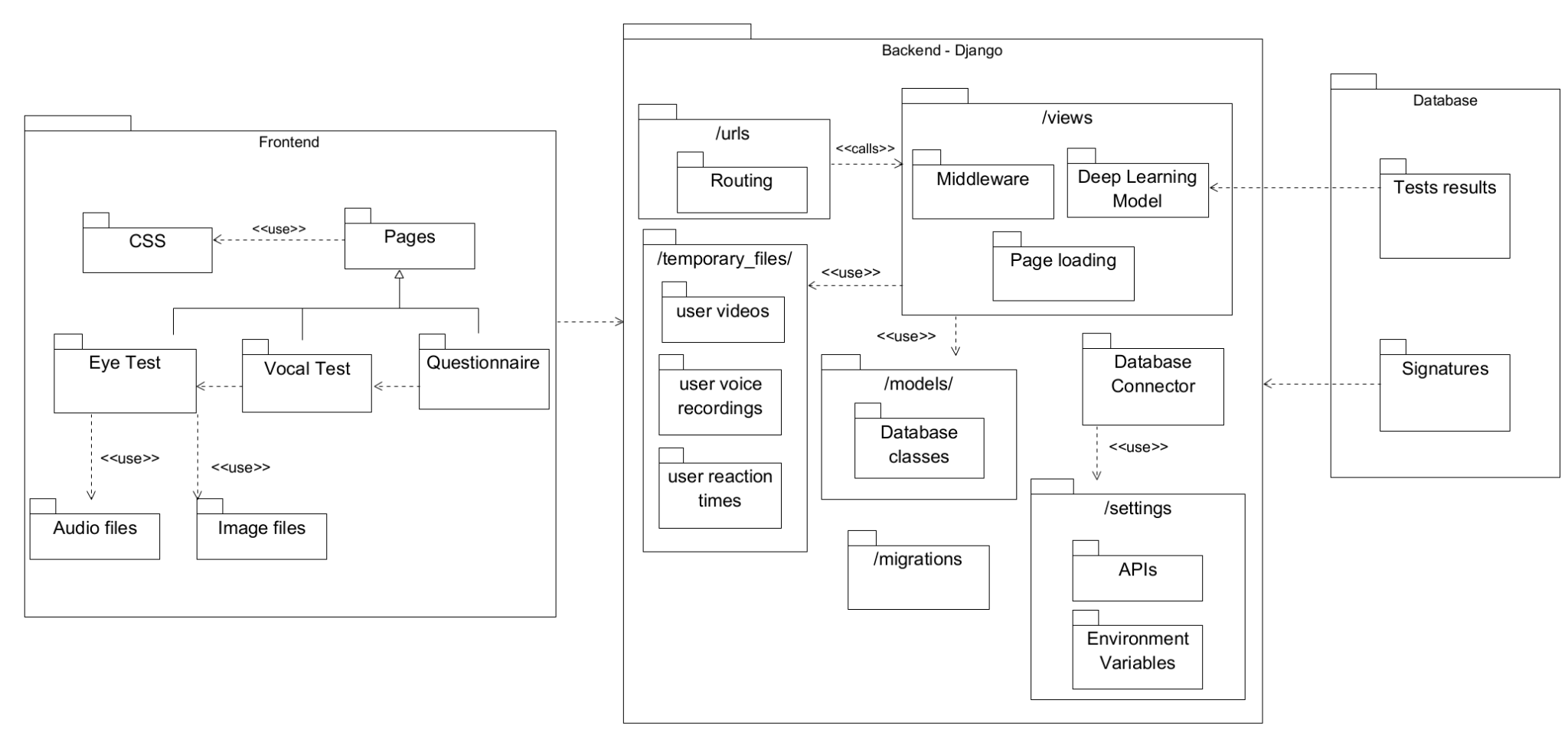
This maintenance guide is intended for engineers responsible for the ongoing support, maintenance, and improvement of the Nodus system. It provides detailed instructions on maintaining system integrity, managing software dependencies, ensuring data security, and troubleshooting common issues. The guide is divided into several sections, each addressing a key aspect of the system’s maintenance requirements.

# 1. System Architecture Overview

In this section we will breakdown the overall structure of the system, explaining how different parts of the system communicate and function, we will delve more into detail when necessary.

## 1.1 System Components

Our system employs a client-server architecture and thus has several independent components, here is a package diagram detailing the system structure:

We can split the system into 4 core components:

* **Frontend:** Built with HTML, CSS, and JavaScript, the frontend handles user interactions and displays results as well as several static files such as audio files and image files which are used by the various tests we conduct, because we employed django the HTML pages are saved in the server and rendered when the user sends the corresponding url route of said page to the back-end, the HTML pages call variables that are sent from the back-end to provide a dynamic front-end experience with the use of Jinja - a python library.
* **Backend:** A Django-based server that processes data, manages user sessions, and interacts with the database and pre-processes the test results before feeding them into the Deep Learning model, this is the central part of the system and this is what we deploy to the cloud - it handles everything from the front-end experience to the Deep Learning model functionality. The back-end houses many folders and files which dictate the system’s functionality and performance and thus we will dedicate an entire section to it later on.
* **Database:** MongoDB is used to store user data, test results, the API key for the database will be found in the environmental variables of the system, alternatively they can be handed over from the original engineering team, the data is saved as documents into one of two collections: **traindata** and **signatures**. The “traindata” documents have 2 attributes:

1. **\_id** - which is the default identifier that MongoDB provides.
2. **Label** - which is a long string with the processed test results and the label given by the system, it should be noted that this attributes name changes according to the label given by the system, I.E in one document it can be “No ADHD” and in another it could be “Combined”.

The “signatures” documents have 3 attributes:

1. **\_id** - which is the default identifier that MongoDB provides.
2. **full\_name** - which is a string with the name provided by the user when they fill out the consent form as well as an underscore (this character \_ ) and afterwards the session token of said user’s browser to help us identify them.
3. **signature** - which is a Base64 conversion of a png of the user’s signature provided by the user in the consent form.

* **Deep Learning Model:** A trained model that performs ADHD classification based on the input data, the model is built on top of the tensorflow platform and loads a custom-made/trained set of weights, the file that we load is found in a folder called “/static/weights/” in the back-end, this model was trained on data we extracted by running an experiment on multiple users, said data was saved on our MongoDB instance under the “Nodus” connection, “adhd” database and “traindata” collection.

## 1.2 System Flow

The Nodus system operates through a structured sequence of steps designed to guide the user from initial consent through testing, data collection, and finally to the presentation of results. The following outlines the typical flow of a successful run within the system:

1. **Consent Form Completion:**Upon accessing the system, the user is first presented with an informed consent form. The form outlines the purpose of the study, procedures involved, potential risks, and the user’s rights. To proceed, the user must carefully read the form, input their full name, and provide a digital signature at the end. This ensures that the user has acknowledged and agreed to the terms of participation, this signature is then saved to our MongoDB instance with the help of the function “save\_signature\_toDB(full\_name, signature)”, the full-name is saved as a string and the signature is saved as a binary file.
2. **Instructions Review:**Following the consent form, the user is directed to the instructions page. Here, the system provides detailed guidance on how to perform the tasks that constitute the test. The user is required to read these instructions thoroughly before proceeding to the first test to ensure proper understanding and execution.
3. **Eye-Test Initialization and Calibration:**The first test is the Eye-Test, which begins with a calibration process. The user is asked to focus on a red circle, and a short video recording is made. This video, named "initial-video.webm," is sent to the backend and temporarily stored in the "/temporary\_files/" directory. The video is renamed to "{session\_key}\_initial\_video.webm" to ensure it is uniquely identifiable by the session key. The "initial\_video\_check()" function is then invoked to process this initial video, ensuring the system is calibrated correctly for the user's specific conditions.
4. **Main Eye-Test Execution:**Upon successful calibration, the system signals the user to begin the main Eye-Test. During this test, two key files are generated: "recorded-video.webm," which captures the user's eye movements, and "reaction-time-arrays.txt," which records the user's reaction times. Both files are sent to the backend and saved in the "/temporary\_files/" folder under the names "{session\_key}\_{filename.filetype}," where {filename.filetype} corresponds to the file’s original name and type. These files are handled by the "upload\_video()" function, which is responsible for routing them to the appropriate processing workflows.
5. **Vocal-Test Execution:**The next stage involves the Vocal-Test. The user is instructed to read aloud a series of sentences, resulting in the creation of an "audio-recording.mp3" file. This file is sent to the backend, saved to the "/temporary\_files/" folder under the name "{session\_key}\_{filename.filetype}," and immediately processed using the "librosa" Python library. The processing extracts two key variables: "y," representing the audio time series, and "sr," the sample rate of "y." These variables are then used by the "soundfile" library to re-save the file under the same name. This step is critical, as directly saving the initial file may result in a faulty .mp3 format, whereas the re-saved version ensures compatibility for subsequent processing.
6. **Questionnaire Submission:**After completing the Vocal-Test, the user proceeds to the Questionnaire section. The responses are compiled into a file named "questionnaire.txt," where each answer is separated by a comma. This file is sent to the backend and stored in the "/temporary\_files/" folder, again under the name "{session\_key}\_{filename.filetype}," for later processing.
7. **Data Processing and Result Generation:**Upon completion of all tasks, the user is directed to a processing page, during which the backend initiates the "results()" function. This function systematically accesses all the files saved in the "/temporary\_files/" directory and begins the analysis and preprocessing of the data using several helper functions. These functions extract and prepare the data, which is then fed into the Deep Learning model via the function "model(eye\_analysis, base, dist, voice\_analysis, questions)." The model returns an array of floats with a length of four, representing the percentage likelihoods of each ADHD subtype. Finally, the system renders the "test-results.html" page, presenting the user with their results, including the context variables 'percentages,' 'eye\_analysis,' 'vocal\_analysis,' 'questionnaire,' and 'reaction\_time.'

It should be noted that the processed test data can be saved to our MongoDB instance if the user answers “Yes” to the “Save data to train our model?” question at the end of the questionnaire and saving the data happens with the help of the function “save\_test\_results\_toDB(label, data)”, the data is saved as a continuous string with the format: (label : data).

This systematic flow ensures that each step of the process is meticulously handled, from initial data collection to final result presentation, thereby providing an accurate and reliable assessment of the user's ADHD profile.

# 2. Software Dependencies

The Nodus system relies on a specific set of software dependencies to function effectively. This section provides an overview of the key libraries and frameworks used in both the backend and data processing components of the system. Proper management and regular updates of these dependencies are essential to maintaining system stability and performance.

## 2.1 Backend Dependencies

The backend of the Nodus system is built using Django, along with several other libraries that handle data processing, machine learning, and audio/video manipulation. The following are the key dependencies:

* **Python:**Needless to say our project is a python based WEB application, we used python version 3.11.9 - said version was the most compatible with all our libraries.
* **Django:**Django is the primary web framework used to manage the backend of the Nodus system. It handles HTTP requests, manages user sessions, and facilitates interaction with the database. Django’s built-in security features ensure that the system is protected against common web vulnerabilities.
* **OpenCV (cv2):**OpenCV is used for real-time computer vision tasks in the Eye-Test component. It processes video data to detect the user's face, track eye movements, and analyze the video frames, which are crucial for assessing the user's focus and reaction times.
* **Librosa and Parselmouth:**These libraries are employed for audio analysis during the Vocal-Test. Librosa provides tools for extracting features from audio files, such as waveform analysis and spectral representations. Parselmouth, with its Praat integration, enables detailed phonetic analysis of the user’s speech, contributing to the overall assessment.
* **Soundfile (sf):**Soundfile is used to read and write sound files, ensuring that the audio data collected during the Vocal-Test is stored and processed in a format compatible with the system's requirements. It plays a critical role in handling the .mp3 files after they are processed by Librosa.
* **TensorFlow:**TensorFlow is the deep learning framework used in the Nodus system for ADHD classification. The model, built using TensorFlow’s Keras API, processes input data from the tests and generates predictions about ADHD subtypes. TensorFlow’s robust features support the complex computations required for this analysis.
* **MongoEngine:**MongoEngine serves as the interface between Django and MongoDB, allowing seamless interaction with the database. It simplifies data manipulation by providing an Object-Document Mapper (ODM) that integrates with Django’s model system.
* **Other Essential Libraries:**
  + **NumPy (np):** Used for numerical operations, particularly in handling array data.
  + **OS:** Facilitates interaction with the operating system, particularly in file handling.
  + **Pathlib (Path):** Provides an object-oriented interface to interact with the file system.
  + **bson (Binary):** Handles the storage of binary data within MongoDB.

## 2.2 Environment Management

The Nodus system operates within a virtual environment to manage its dependencies. This ensures that all necessary packages are isolated from the system-wide Python installation, preventing conflicts and ensuring consistency across different development and deployment environments.

The project was built on a conda virtual environment as it helps isolate the system from and create a modular safe-to-use version of the system which can be installed on any device.

The environment and all its dependencies can be installed using pip using the “pip install -r requirements.txt” using the requirements.txt file or alternatively can be installed using conda with the “conda env create -f environment.yaml” using the environment.yml file which we provided.

# 3. Code Review

This section of the guide provides a comprehensive analysis of the backend codebase. We will delve into the functionality of each component, explaining the purpose and implementation of the code, as well as its role within the overall system architecture. Additionally, we will discuss best practices for maintaining the code, ensuring its scalability, and facilitating future enhancements. This review aims to equip developers with a deep understanding of the backend’s inner workings, enabling them to effectively manage and extend the system as needed.

## 3.1 manage.py file

This file is responsible for running the server and is provided by default by the Django framework when creating a new project.

To run the server you have to navigate to the folder which has this file and run the following command: “py manage.py runserver”, alternatively you can run the command “python manage.py runserver”.

This file serves no other purpose aside from activating and running the back-end automatically, the default setting of which is the localhost running on port 8000, we won’t be going over the alternatives to running on localhost for we will dedicate an entire section to the deployment.

## 3.2 db\_connection.py file

This file is responsible for connecting the back-end to our MongoDB instance, the information needed for said connection are fetched from the operating system’s environmental variables and are labeled “DATABASE\_USERNAME” and “DATABASE\_PWD” for the username and password respectively.

The connection is then handled by the library PyMongo and can be then accessed by any file in the back-end using the “db” variable declared in this file by using “from db\_connection import db”.

## 3.3 FinalProject folder

This is the main folder in our back-end architecture and is provided by default by Django when creating a new project, typically referred to as the "project folder" or "project directory." This folder is crucial as it serves several important functions:

### 3.3.1 settings.py file

This file is a central component of the Django project, responsible for defining the configuration settings that dictate the operational behaviour of the application. This file includes a comprehensive array of settings that govern various aspects of the project, including:

* **Basic Configuration:** The **BASE\_DIR** setting establishes the base directory for the project, ensuring that all file paths within the project are correctly resolved relative to this directory.
* **Security Settings:** The **SECRET\_KEY** is a critical security measure used for cryptographic signing, essential for securing sessions, cookies, and other sensitive data. In production environments, the DEBUG setting must be disabled (False) to prevent the exposure of sensitive error details. The **ALLOWED\_HOSTS** setting specifies which domains are permitted to serve the Django site, protecting against HTTP Host header attacks.
* **Installed Applications:** The **INSTALLED\_APPS** section lists all the Django applications that are active within the project. This includes both the built-in Django apps (such as authentication and admin interfaces) and any custom applications developed for the project.
* **Middleware:** Middleware components listed in the **MIDDLEWARE** setting are responsible for processing requests and responses as they pass through the application. Each middleware component performs a specific function, such as managing sessions, enforcing security measures, or transforming HTTP requests and responses. The order of middleware execution is critical to ensure the correct functioning of these components.
* **URL Configuration:** The **ROOT\_URLCONF** setting points to the module that contains the URL declarations for the project. This setting is crucial for defining the routing patterns that direct incoming HTTP requests to the appropriate views within the application.
* **Template Configuration:** The **TEMPLATES** setting manages how Django processes and renders templates. It includes settings for the template engine, directories where templates are stored, and context processors that make certain variables globally available to all templates.
* **WSGI Configuration:** The **WSGI\_APPLICATION** setting specifies the WSGI application callable that Django uses to serve the project. WSGI (Web Server Gateway Interface) is a standard interface between web servers and Python web applications.
* **Database Configuration:** The **DATABASES** setting defines the database connection details for the project. It specifies the database engine, database name, and other connection parameters necessary for Django to interact with the database.
* **Password Validation:** The **AUTH\_PASSWORD\_VALIDATORS** setting contains a list of password validation rules that enhance security by enforcing strong, complex passwords. These rules help protect user accounts from unauthorized access.
* **Internationalization and Localization:** The **LANGUAGE\_CODE** and **TIME\_ZONE** settings control the language and time zone settings for the project. These configurations determine how dates, times, and language-specific content are displayed.
* **Static Files:** The **STATIC\_URL** and **STATICFILES\_DIRS** settings manage the handling of static files, such as CSS, JavaScript, and images. These settings are essential for serving static content efficiently, both in development and production environments.
* **Default Primary Key Field Type:** The **DEFAULT\_AUTO\_FIELD** setting defines the type of primary key field that Django will use by default for models. This setting helps streamline database management by ensuring that primary keys are automatically generated and managed.

### 3.3.2 \_\_init\_\_.py file

This file is crucial for the Django framework to function, although it’s an empty file removing it would render the system unusable as it wouldn’t be able to run the server.

### 3.3.3 wsgi.py file

The wsgi.py file is a critical component in the Django project architecture, serving as the entry point for WSGI-compatible web servers to interact with the Django application. WSGI, which stands for Web Server Gateway Interface, is the standard Python interface between web servers and Python web applications. This file is particularly important in production environments, where it facilitates the deployment of the Django project.

The primary function of wsgi.py is to expose the WSGI callable as a module-level variable named application. This variable allows WSGI servers, such as Gunicorn or uWSGI, to communicate with the Django application, handling requests and responses in a standardized way. The wsgi.py file ensures that the application can serve web pages, process data, and perform other web-related tasks efficiently and securely in a production environment.

### 3.3.4 asgi.py file

The asgi.py file plays a similar role to wsgi.py, but it is designed for use with ASGI (Asynchronous Server Gateway Interface) servers. ASGI is a newer standard than WSGI and is designed to handle asynchronous communication, making it more suitable for modern web applications that require real-time features such as WebSockets, chat applications, or live notifications.

The asgi.py file serves as the entry point for ASGI-compatible web servers to interface with the Django application. Like wsgi.py, it exposes an ASGI callable as a module-level variable named application. This allows ASGI servers, such as Daphne or Uvicorn, to manage asynchronous requests and responses, enabling the Django application to handle multiple connections simultaneously without blocking.

ASGI is particularly important for applications that need to manage high levels of concurrency or require long-lived connections, such as streaming data or interactive features. The asgi.py file ensures that the Django project is capable of supporting these advanced functionalities, providing the flexibility to deploy the application in environments that demand asynchronous processing.

### 3.3.5 urls.py file

This file in the FinalProject folder is responsible for routing the urls that arrive at the back-end, if any other routes need be added they could be added to this file or alternatively they could be added to another file and afterwards we can add a route that goes to the new file from this current “urls.py” file.

In our project we chose to do the latter as we primarily worked in a different app folder called ADHD to create an isolated environment, we then added a route using this command: path('', include('ADHD.urls')) - which states that all routes that arrive at this current “urls.py” file should also check the routes in the “urls.py” file in the ADHD app.

## 3.4 ADHD folder

This folder represents the app we created, the Django framework allows developers to create sub apps in the system, each sub app can work independently and are all connected to the central project directory (in our case FinalProject folder).

We decided to create an app called ADHD which has the HTML pages, the JS scripts used in the HTML pages, the server-side logic, the app routes, the database collections, the image, vocal and Deep Learning model weights files and a folder where temporary files are stored as they await processing.

We will go over each file and folder in the ADHD folder and explain their usage in great detail.

### 3.4.1 \_\_pycache\_\_ folder

This folder has compiled versions of the .py files, usually saved with the .pyc file type, this folder basically works as a local version history provided by the Django framework.

**This folder works automatically and should not be changed manually**

### 3.4.2 migrations folder

This folder serves as a version history for the database declared in the settings and the models created in the “models.py” file, the files found in this folder represent changes made to the models found in “models.py” file - thus this folder saves changes made to said models to later merge them with the database instances, allowing for a seamless transition between new and old database.

**This folder works automatically and should not be changed manually**

### 3.4.3 signatures folder

This folder has all the user signatures submitted from the consent form and sent to the back-end, the purpose of this folder is merely to save the signatures locally if the developers choose to work on the localhost instead of the cloud version of the project and need quick access to the signature files as images instead of Binary files.

**This folder is not used by the cloud version and will not see any changes on its own, it exists purely for testing and verification purposes.**

### 3.4.4 static folder

This folder has all the non-HTML static files, I.E images, audio files and Deep Learning model weights, the files in the static folder are organized into 4 main categories: Images, Audio, Scripts and Weights - all of which are subfolders that have several files, it should be mentioned that the scripts folder is not in use anymore as each HTML page has it’s own scripts embedded in the page.

Any change/additional functionality to any of the HTML scripts should be resolved in the HTML page itself, as inserting new scripts into this folder will not trigger them as we ran into several issues with Django static file calling the script and having 2 way communication between the script files and the HTML scripts.

The Deep Learning model weights should be put in the “weights” subfolder in the “static” folder under the same name as the old weights file in order for the model to load the weights successfully, alternatively once can change the name of the weights file loaded into the model, the code can be found directly under the imports in the “views.py” file.

### 3.4.5 templates folder

This folder has all the HTML pages, we preferred putting the HTML pages in a separate file and not in the static folder.

This folder is used when the back-end renders an HTML page.

### 3.4.6 temporary\_files folder

This folder serves as a local storage in the back-end, all the user test result files are stored in this folder temporarily until the pre-processing part is done and the model does its calculations and gives the final results.

The contents of this folder change constantly all on their own, a user’s files are saved with his session\_key in their name to help us clean the appropriate files when the tests conclude and the results are shown.

### 3.4.7 \_\_init\_\_.py file

Just as we mentioned before, this file is crucial for the Django framework to function, although it’s an empty file, removing it would render the system unusable as it wouldn’t be able to run the server.

### 3.4.8 admin.py file

This file in the Django app directory is responsible for registering models to the Django admin interface. This file allows the models defined in the app to be managed through Django's built-in admin panel (a WEB page). To make a model accessible in the admin interface, it must be imported and registered within admin.py using admin.site.register(ModelName).

In our project, we did not use the admin.py functionality, we didn’t find it necessary for our system’s development.

### 3.4.9 apps.py file

The apps.py file in the ADHD app is used to configure the app's settings within the Django project. It defines the AdhdConfig class, which inherits from AppConfig. This class specifies the configuration for the ADHD app, including the default type of primary key field used for models (BigAutoField) and the name of the app ('ADHD'). This configuration is automatically referenced by Django when the app is included in the INSTALLED\_APPS setting of the project's settings.py file.

### 3.4.10 models.py file

The models.py file in the ADHD app is responsible for defining the data models used within the application. In Django, models are the source of truth for the structure of your database tables. This file contains an unused model which is the Patient model. which defines the schema for storing patient data, including fields like Subtype and Percentage, both of which are character fields with a maximum length of 255 characters - said model can be removed, it is not necessary for the system’s functionality.

Additionally, this file sets up connections to MongoDB collections using the db\_connection module. The signatures\_collection and train\_data\_collection variables represent the MongoDB collections named 'signatures' and 'traindata', respectively. These collections are accessed directly in the application for storing and retrieving patient-related data outside of Django's ORM.

### 3.4.11 tests.py file

The tests.py file in the ADHD app is typically used for writing automated tests to validate the functionality of the application. It is provided by Django as a default location for unit tests using Django's TestCase class. However, in our project, this file was not utilized, and no tests were implemented within it. If testing is required in the future, this file can be used to create and organize test cases to ensure the reliability and correctness of the app’s functionality.

### 3.4.12 urls.py file

This file in the ADHD app is responsible for defining the URL routing patterns for the application. It maps URL paths to specific views, determining how incoming requests are handled and which part of the application responds. This file includes routes for all major components of the system, such as the eye test, vocal test, questionnaire, and results page. Each URL pattern is associated with a corresponding view function in the views.py file, allowing for organized and efficient handling of user requests.

Key routes include:

* **ADHD**: Routes to the eye test page.
* **Landing/Home**: Routes to the landing page of the application.
* **FAQ**: Routes to the Frequently Asked Questions page.
* **Instructions**: Routes to the instructions page before tests begin.
* **Video Configuration**: Handles video upload and processing for the eye test.
* **Vocal Test**: Routes to the vocal test page.
* **Questionnaire**: Routes to the questionnaire page.
* **Audio Configuration**: Handles audio upload and processing for the vocal test.
* **Results**: Displays the test results.
* **Processing**: Indicates that the results are being processed.
* **Done**: Cleans up and finalizes the session.
* **Consent**: Routes to the consent form.
* **Signature**: Handles the saving of user signatures.

This file is central to the operation of the ADHD app, ensuring that all user interactions are correctly routed to the appropriate functionality within the application.

### 3.4.13 views.py file

This file has all the main functionality of the back-end, it’s made up of various functions which are connected to the routes as well as the declaration and usage functionality of the Deep Learning model.

We will dedicate the next section to this file specifically as most of the work was done in it.

# 4. Back-end Code

This section serves as the expanded and practical review of the “views.py” file specifically, in this section we will explain each function making up the file, including their purpose, their method of achieving said purpose, inputs and outputs.

## 4.1 Deep Learning model

The declaration starts straight after the imports (line 25), its a multi-layered model which receives the a NumPy array which represents the various inputs:

“” **eye\_tracking\_input\_shape = (3,)** “”

“” **reaction\_time\_input\_shape\_1 = (3,)** “”

“” **reaction\_time\_input\_shape\_2 = (3,)** “”

“” **audio\_input\_shape = (5,)** “”

“” **questionnaire\_input\_shape = (20,)** “”

In total the model receives 34 numbers as inputs, it processes the inputs in a concatenated fashion, we see the concatenation in the following line:

“” **concatenated = Concatenate()([x\_eye, x\_reaction\_time\_1, x\_reaction\_time\_2, x\_audio, x\_questionnaire])** “”

The model then loads the weights found in the static folder, see the following lines:

“” **weight\_path = Path('ADHD/static/weights/modelweights.weights.h5)** “”

“” **model\_pred.load\_weights(weight\_path)** “”

The model is also used the following function:

“” **def model(eye, react1, react2, vocal, questions)** “”

Which receives the input arrays as strings and turns them to NumPy arrays in the shape mentioned above and feeds that data to the model, which then returns a NumPy array of floats in the shape (4,) which represent percentages summing up to 1.

## 4.2 Render Functions

We will now go over all the functions that merely render an HTML page.

Said functions include:

* **def consentForm(request)**
* **def instructions(request)**
* **def eyeTest(request)**
* **def vocalTest(request)**
* **def questionnaire(request)**
* **def landingPage(request)**
* **def FAQ(request)**
* **def processing(request)**

All the functions mentioned above render their HTML respectively and do nothing more, they receive a Django request as input and return the rendered HTML page.

We see the following lines of code that render pages at the end of each one of said functions:

“” **template = loader.get\_template('{HTML\_PAGE}')** “”

“” **return HttpResponse(template.render())** “”

Aside from that each one of those functions has access to the session\_key variable which is an identifier found in the user request, the functions save that session\_key to a local variable should they need it.

## 4.3 Data Processing Functions

This section will handle the functions which process data, both the ones that pre-process them before feeding them to the back-end and the ones that deal with the data when it first arrives at the back-end.

We will list the functions based on the order in which they appear in the “views.py” file:

* **def saveSignature(request)** - receives the user’s signature and full-name and saves the signature as a Binary file into the MongoDB instance. This function is labeled with csrf\_exempt because it saves a file locally in the back-end.
* **def clean(request)** - receives a request from a user, extracts the session\_key of the user and removes all the user’s files saved locally in the back-end.
* **def results(request)** - receives a request from a user, extracts the session\_key and loads all the files saved in the back-end of said user based on the session\_key, processes them using other helper functions and concatenates all the results to 1 string and splits that string into multiple tuples and send those tuples to the “model” function - the “results” function returns a rendered “test-results.html” page with the ADHD subtype percentages along with tips for how to deal with the most prominent subtype.
* **def upload\_video(request)** - receives a request from a user with a video in the .webm format, this function saves that video locally in the /temporary\_files/ folder under the name {session\_key}\_{video\_name}. If the video received is the first video or “initial\_video.webm” this function then calls the next function and returns its result to the user.
* **def initial\_video\_check(url)** - This function receives the file path of a video as input and processes it to determine whether the user’s face and eyes are detected consistently throughout the video. The function uses OpenCV’s pre-trained Haar cascades for face and eye detection. It reads the video frame by frame, counting the number of frames where faces and eyes are successfully detected. After processing all frames, it returns a boolean value based on whether the number of frames with successful detections is significantly greater than those without. Specifically, the function returns True if the number of detected frames is more than 1.5 times the undetected frames and there are more than 40 detected frames in total.
* **def eye\_test\_analysis(url)** - This function processes a video file to track and analyze the user’s eye movements during an eye-tracking test, utilizing OpenCV’s Haar cascades for face and eye detection. As the video is read frame by frame, the function identifies the user's face and then detects the eyes within the region of interest. It records the initial coordinates of the left and right eyes and tracks their movements throughout the video, storing these movements in arrays. The function then calculates the differences in eye positions over time and computes the mean, median, and standard deviation of these movements for both eyes. The results are returned as a summary of the user's eye-tracking data, which can be used to assess their focus and reaction during the test.
* **def upload\_voice(request)** - This function handles the upload and processing of an audio file submitted by the user. It begins by ensuring that the user has an active session, creating one if necessary, and retrieves the session key to uniquely identify the file. When a POST request containing an audio file (specifically named 'audio-recording') is received, the function saves the file locally in the /ADHD/temporary\_files/ directory, appending the session key to the file name for identification. The audio file is then processed using the librosa library to load the audio data, after which it is re-saved using the soundfile library (sf.write) to ensure the file is in a compatible format for further processing.
* **def analyze\_voice(url)** - This function processes an audio file from the vocal test to analyze various vocal characteristics of the user. The function begins by extracting the Fundamental Frequency (F0) of the audio, which is a key measure of pitch. It then calculates additional vocal metrics such as jitter, shimmer, and Harmonics-to-Noise Ratio (HNR), which provide insights into the stability and quality of the user's voice. Finally, the function computes a composite metric called the Voice Quality Index (VQI), which combines these individual metrics into a single value that offers a general assessment of vocal quality. The function returns these five values—F0, jitter, shimmer, HNR, and VQI—which can be used for further analysis or evaluation of the user's vocal characteristics.
* **def upload\_answers(request)** - This function handles the upload and saving of the user's responses to the questionnaire. It begins by ensuring the user has an active session, creating one if necessary, and retrieves the session key to uniquely identify the file. When a POST request containing the questionnaire answers is received, the function saves these answers locally in the /ADHD/temporary\_files/ directory, naming the file with the session key followed by the filename to ensure it is uniquely associated with the user's session.

It should also be mentioned that some of these functions employ a progression system, this is what the progression system looks like in the code:

“” **if request.session.get('progress') != 'step3':** “”

“” **return redirect('/ADHD/')** “”

## 4.4 Helper Functions

This section will handle the helper functions, which are functions with generic functionality which are reused throughout the code, many of the functions in the last section rely on the functions covered in this section.

We will list the functions based on the order in which they appear in the “views.py” file:

* **def getLabel(questionnaire)** - This helper function determines the user's ADHD subtype based on their responses to the questionnaire. The function initializes two counters, inat and hyp, to track symptoms related to inattention and hyperactivity, respectively. It then iterates through the user's answers, converting each answer from a string to an integer. For the first 9 questions, responses of 2 or 3 increase the inat counter, indicating inattention symptoms. For the next 9 questions, similar responses increase the hyp counter, indicating hyperactivity symptoms. After evaluating the relevant answers, the function categorizes the user into one of four subtypes: "Combined" (if both inattention and hyperactivity symptoms are prominent), "Hyperactivity" (if only hyperactivity symptoms are prominent), "Inattentive" (if only inattention symptoms are prominent), or "No ADHD" (if neither set of symptoms is sufficiently present). This categorization is returned as the function's output.
* **def get\_tips(percentages)** - This helper function provides tailored tips based on the user's ADHD subtype, determined by the highest percentage value in the input array. The function defines four different sets of tips corresponding to the possible subtypes: "Combined," "Hyperactive," "Inattentive," and "No ADHD." Each set of tips offers specific advice to help manage symptoms associated with that subtype or to promote overall well-being. The function stores these tips in an array and uses np.argmax(percentages) to identify the index of the highest percentage, which corresponds to the user's most likely subtype. The function then returns the set of tips associated with that subtype, providing personalized guidance to the user based on their individual needs.
* **def calculate\_difference(eye\_position\_array)** - This helper function calculates the squared differences between consecutive eye positions in the provided array. The function iterates through the eye\_position\_array, which contains the coordinates of the eye positions at each step of the eye-tracking test. For each pair of consecutive positions, it computes the squared distance using the distance\_squared function and appends this value to a new list called difference. The function returns this list of squared differences, which can be used to analyze the variability and movement of the user's eyes over time.
* **def distance\_squared(point1, point2)** - This helper function calculates the squared Euclidean distance between two points, point1 and point2, in a 2D space. Each point is represented as a pair of coordinates (x, y). The function computes the difference between the x-coordinates and the y-coordinates of the two points, squares these differences, and then sums them to produce the squared distance. The result is returned as a single numerical value, which is used to measure the distance between the two points without taking the square root, thereby avoiding unnecessary computation in contexts where the squared distance is sufficient.
* **def calculate\_mean\_median\_std(eye\_movement\_distance\_array)** - This helper function computes three statistical measures—mean, median, and standard deviation—for a given array or list of numerical values, specifically distances related to eye movements. The function begins by iterating through the input eye\_movement\_distance\_array and appending each value to a new list called data. It then uses NumPy functions to calculate the mean, median, and standard deviation of the values in data. These three statistical measures are returned as a tuple, providing a summary of the distribution and variability of the eye movement distances.
* **def extract\_fundamental\_frequency(audio\_path)** - This helper function extracts the Fundamental Frequency (F0) from the user's voice, which is a key measure of pitch. The function uses the parselmouth library to process the audio file specified by audio\_path. It first creates a Sound object from the audio file and then generates a pitch object using the To Pitch function, specifying a range of expected pitch frequencies (from 75 to 600 Hz). The function then calculates the mean fundamental frequency across the entire audio file in Hertz using the Get mean function. The calculated mean F0 is returned as a numerical value, representing the average pitch of the user's voice.
* **def extract\_jitter\_shimmer\_hnr(audio\_path)** - This helper function analyzes the user's voice to extract three important vocal characteristics: Jitter, Shimmer, and Harmonics-to-Noise Ratio (HNR). Using the parselmouth library, the function processes the audio file provided through audio\_path, first creating a Sound object and generating a PointProcess object to analyze the periodicity of the voice within a typical pitch range. It calculates Jitter, which measures frequency variation and reflects pitch stability, Shimmer, which measures amplitude variation and indicates the consistency of voice amplitude, and HNR, which assesses the ratio of harmonic sounds to noise, indicating vocal clarity. These metrics are then returned as a tuple, providing a comprehensive analysis of the user's vocal quality.

## 4.5 Database Functions

This is a short section, it delves into the functions in the “views.py” file that have access to the MongoDB instance.

There are only 3 functions that access the database, 1 of which is not used at the moment but it exists to help us extract information from the database by using the back-end instead of MongoDB Compass or Atlas.

We will list the functions based on the order in which they appear in the “views.py” file:

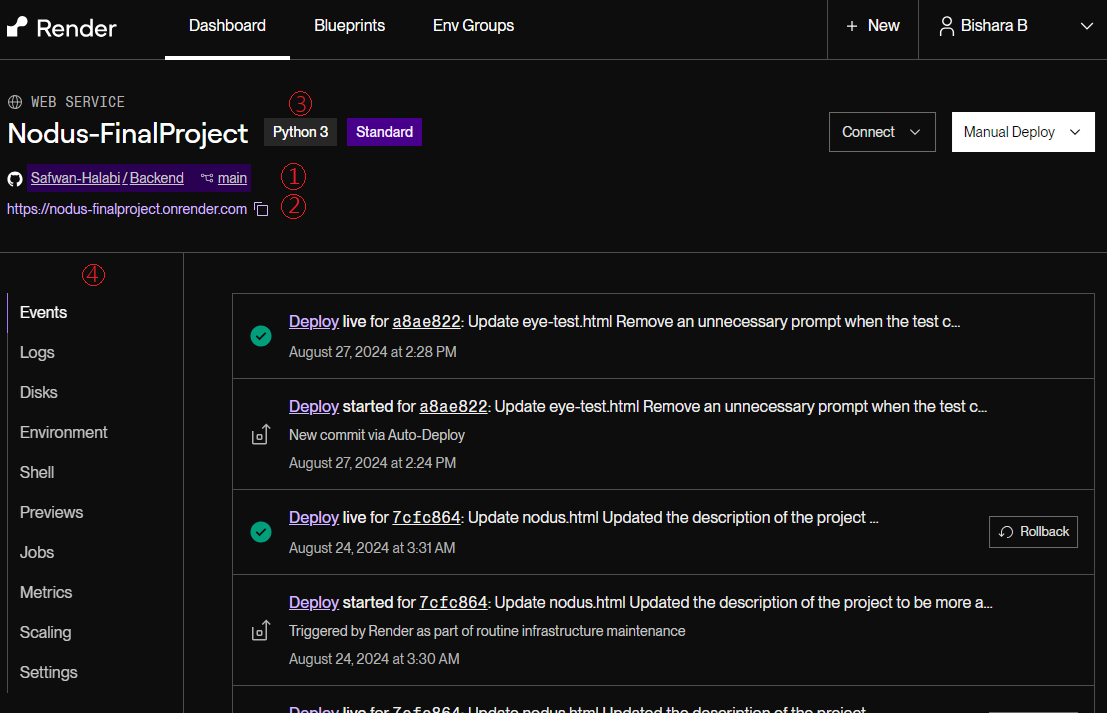
* **def save\_signature\_toDB(full\_name, signature)** - This function saves a user's digital signature to a MongoDB database. It first checks if the provided signature is an instance of InMemoryUploadedFile, ensuring that the uploaded file is valid. If valid, the function reads the file content and converts it into BSON Binary format, which is suitable for storing in MongoDB. It then creates a record containing the user's full name and the binary signature, which is inserted into the signatures\_collection in MongoDB. The function returns an HTTP response indicating the successful addition of a new record. If the file type is invalid, it returns an HTTP response with a status of 400, indicating the error.
* **def save\_test\_results\_toDB(label, data)** - This function is designed to create a record for storing test results in a database. It takes two parameters: label, which identifies the type or category of data being stored, and data, which contains the actual test results. The function creates a dictionary where the label serves as the key and data as the corresponding value, effectively organizing the test results under the specified label. However, the function as written does not currently save the record to a database or return any response, so additional implementation is needed to complete the storage process.
* **def get\_all\_signatures(request)** - This function retrieves all digital signatures stored in the MongoDB signatures\_collection. It queries the collection using the find() method, which returns a cursor containing all documents (signatures) in the collection. The function then returns this cursor, which can be used to iterate over and access the individual signature records. Currently, the function does not format or process the results for display, nor does it return an HTTP response, so further implementation may be needed depending on how the data will be used or presented.

# 5. Deployment

## 5.1 Overview

Our service was deployed on the Render platform. The platform’s environment is simple and easy to use, with minimal configuration required. The deployment is directly linked to the [GitHub repository](https://github.com/Safwan-Halabi/Backend/tree/main)’s main branch where the project resides. This integration allows for a streamlined update process: whenever new commits are pushed to the repository, Render automatically detects these changes and triggers a redeployment. This continuous deployment pipeline ensures that the website stays up-to-date with the latest code, reducing manual intervention. Render integrates a linux shell, allowing an easy way to access the cloned files from the github which the deployment is built upon, as well as modifying them, however, the modification is only done in the render environment and does not affect the github repository.

## 5.2 Render dashboard



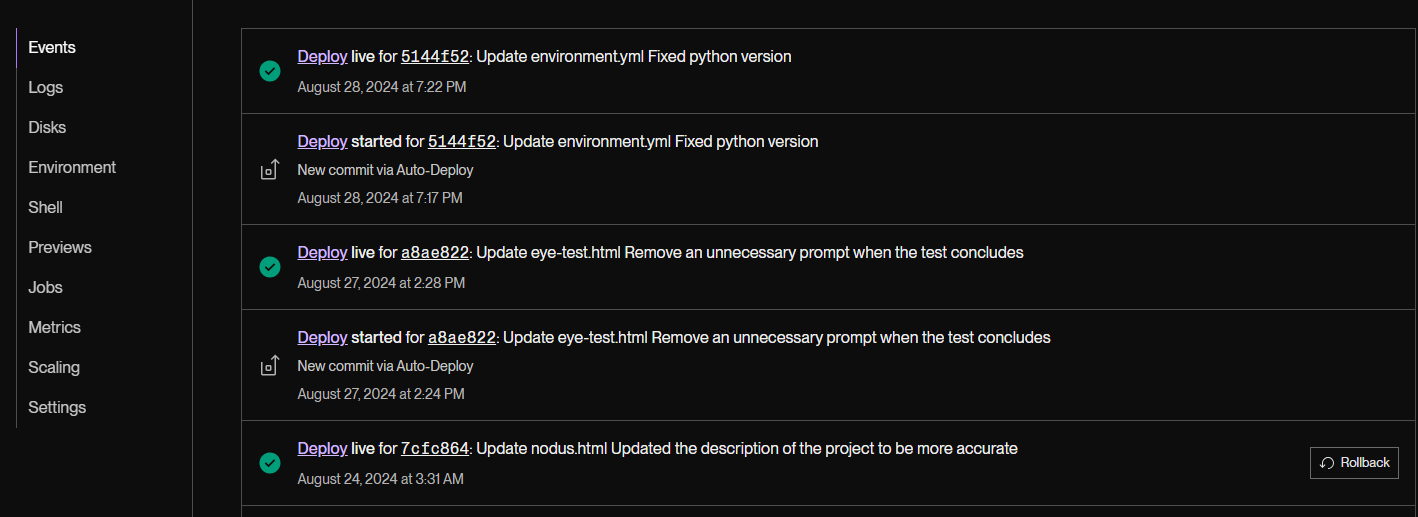
The above image represents the Render dashboard. Key components in the page:

1. The main repository of the *Backend* github repository.
2. The website’s URL provided by Render.
3. Python3: Our project was developed with 3.11.9.
4. The control panel for configuring the deployed service.

## 5.3 The control panel

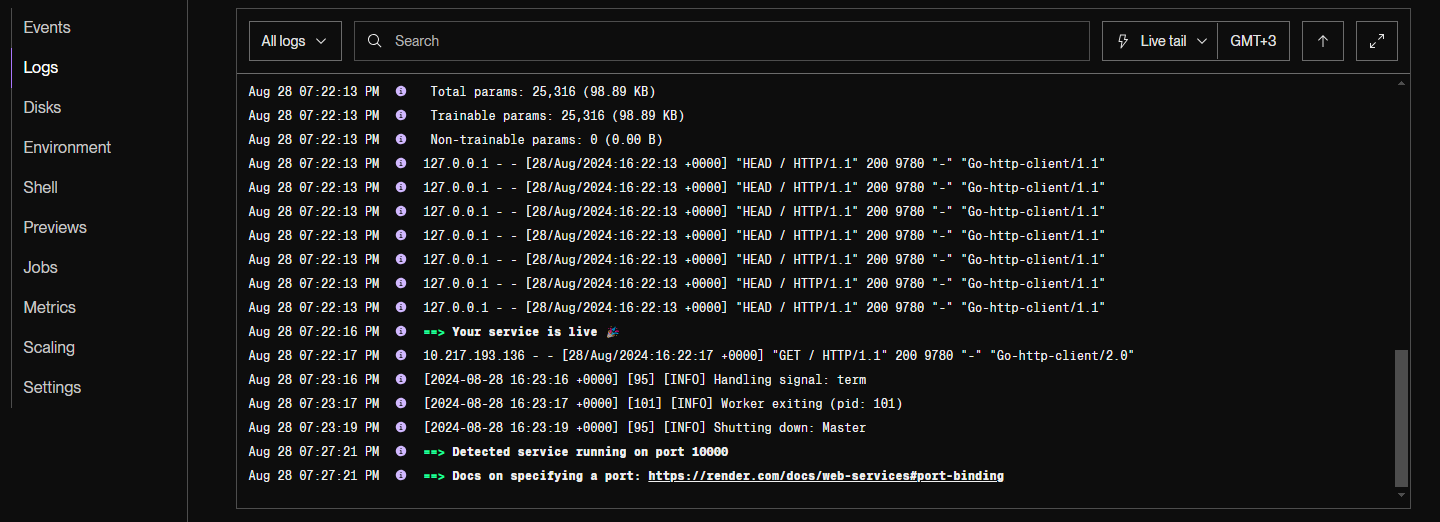
The control panel (Denoted (4) in the image from the previous section: Render’s dashboard) is the main tool for configuring the website. Includes: Events, Logs, Disks, Environment, Shell, Previews, Jobs, Metrics, Scaling and Settings. However not all of the said tools are necessary to use and modify. The next subsections will discuss the applications of the tools we have used to maintain our web service.

### 5.3.1 Events



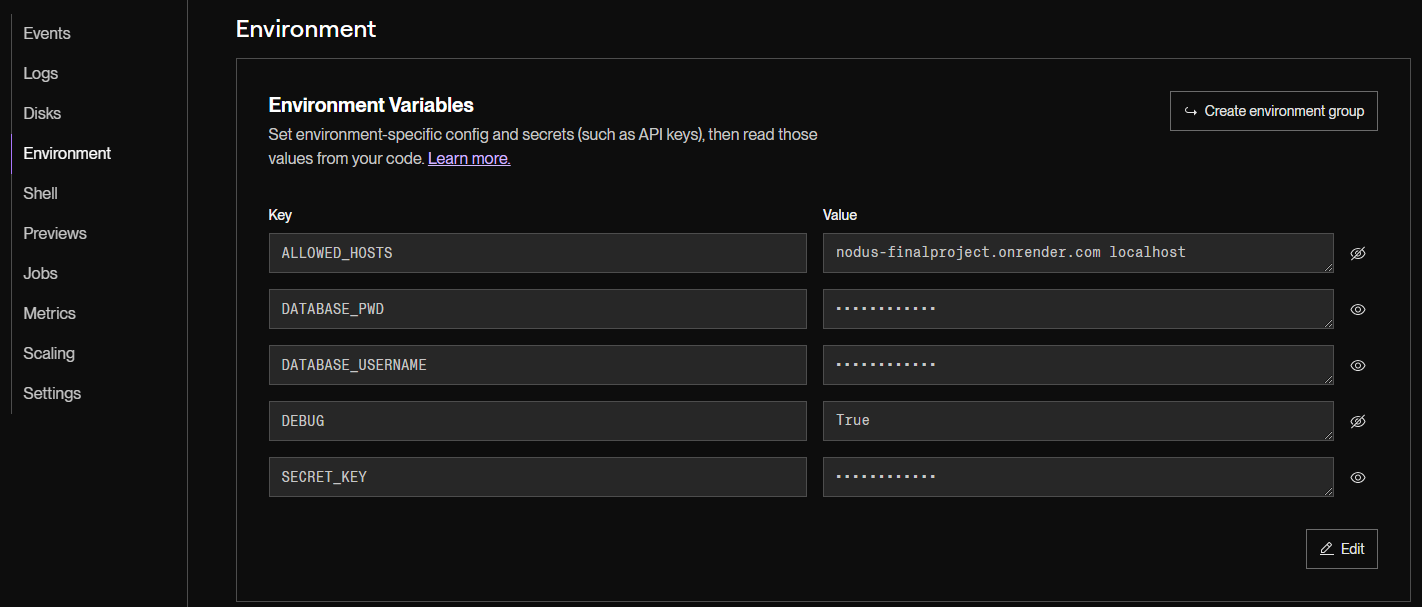
**Events** section provides a summary of entries representing the previously deployed commits to the Github repository, Each entry can be accessed, displaying a detailed log on the deployment process. Example log will be presented in the next subsection

### 5.3.2 Logs



**Logs** section displays the log that corresponds to the current deployed version of the service, providing a detailed look into the website’s deployment status, and visitors’ activities on the website. In the above image, a message saying: “Your service is live” conveying that the service has been successfully deployed and live. It also displays the “GET” and “POST” requests.

### 5.3.3 Environment

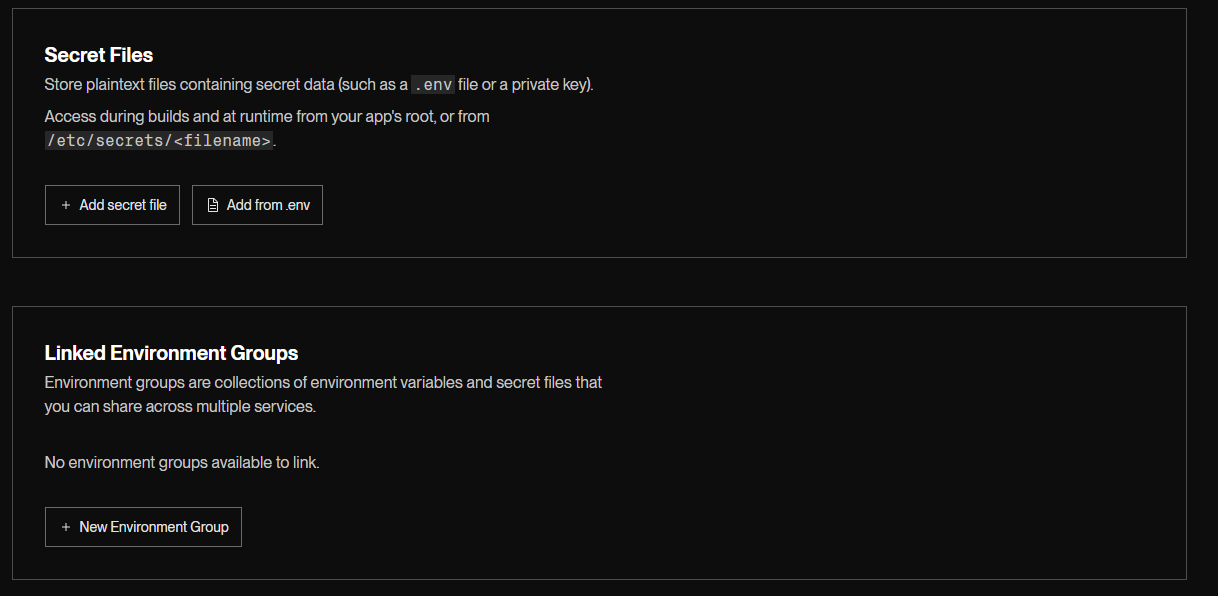


The **Environment** section stores the environment variables used for this project. These variables include sensitive information and configuration settings such as database credentials, API keys, and debug modes, which are essential for the application to run correctly in different environments (e.g., development, testing, production).

By storing these variables in Render, they are securely encrypted and kept separate from the codebase, reducing the risk of accidental exposure. Moreover, the environment variables can be easily updated without requiring changes to the code, allowing for flexible configuration and quicker deployment adjustments as needed.

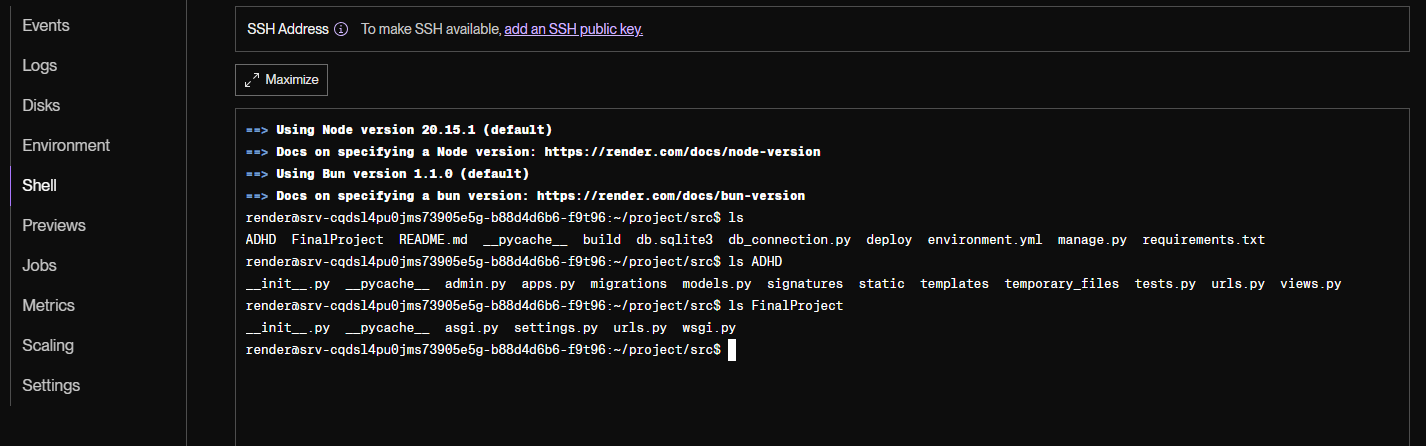
The following is the goal of each key:

* **ALLOWED\_HOSTS**: Specifies a list of host/domain names that the web server can serve. This prevents HTTP Host header attacks. In this case, it allows requests from the website’s URL and localhost.
* **DATABASE\_PWD**: Stores the password for accessing the project's database.
* **DATABASE\_USERNAME**: Contains the username used to connect to the project's database.
* **DEBUG**: A boolean setting (True or False) that determines whether the deployment is in debug mode. When set to True, detailed error pages will be shown; for production, this should be set to False.
* **SECRET\_KEY**: A unique key used for cryptographic signing, such as sessions and cookies in web applications. The key is generated by Render.



Environment section enables storing Secret Files, and Linked Environment Groups. The said settings are not used for this project.

### 5.3.4 Shell

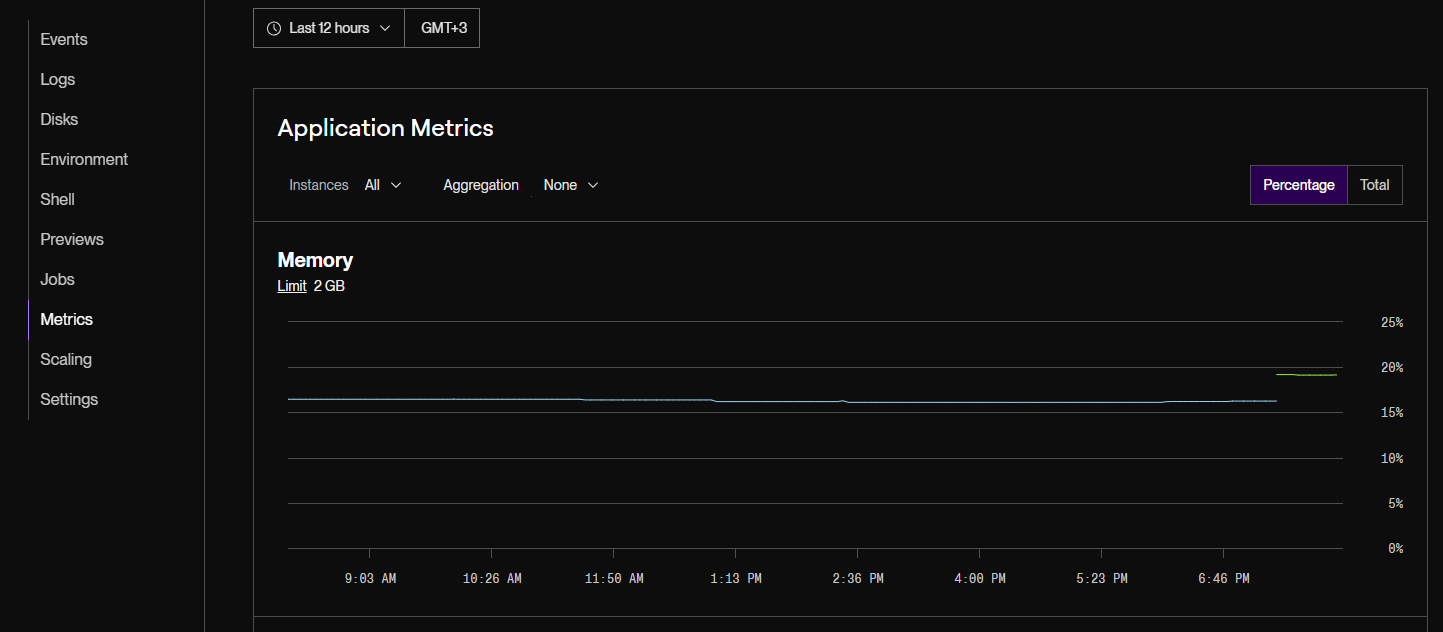


The **Shell** section in Render provides a linux command-line interface (CLI) to the server environment where the project is hosted. This feature allows developers to execute commands directly on the server, making it easier to perform various administrative tasks such as inspecting files, managing processes, running scripts, and debugging.

In the screenshot, the shell output shows commands like *ls*, which lists the contents of the project's directory which has been cloned and deployed.

The Shell tool is particularly useful for quickly accessing the server to make adjustments or monitor the deployment's health, providing flexibility and control over the deployed application.

### 5.3.5 Metrics



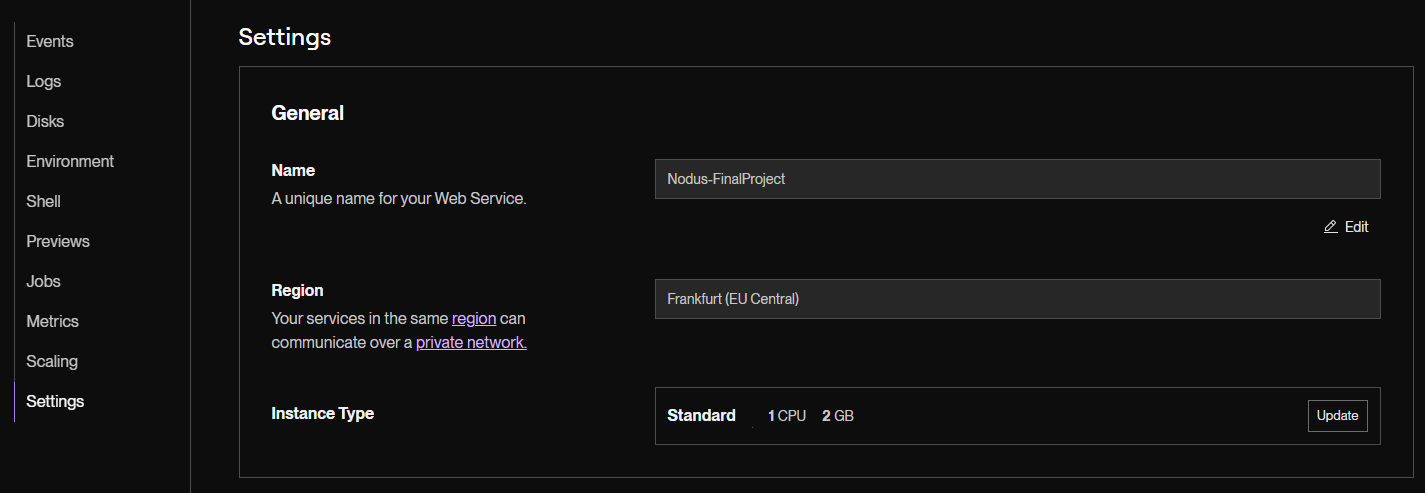
The **Metrics** section provides valuable insights into the application's performance, and resource usage, allowing for proactive management and optimization.

The screenshot shows the **Memory** usage graph, which displays the amount of memory being used by the application over time. The memory limit is set to 2 GB, and the graph shows the memory consumption in percentage or total usage.

Other metrics available in this section (though not shown in the screenshot) include: CPU Usage, Network metrics including total requests and response times, and Outbound bandwidth.

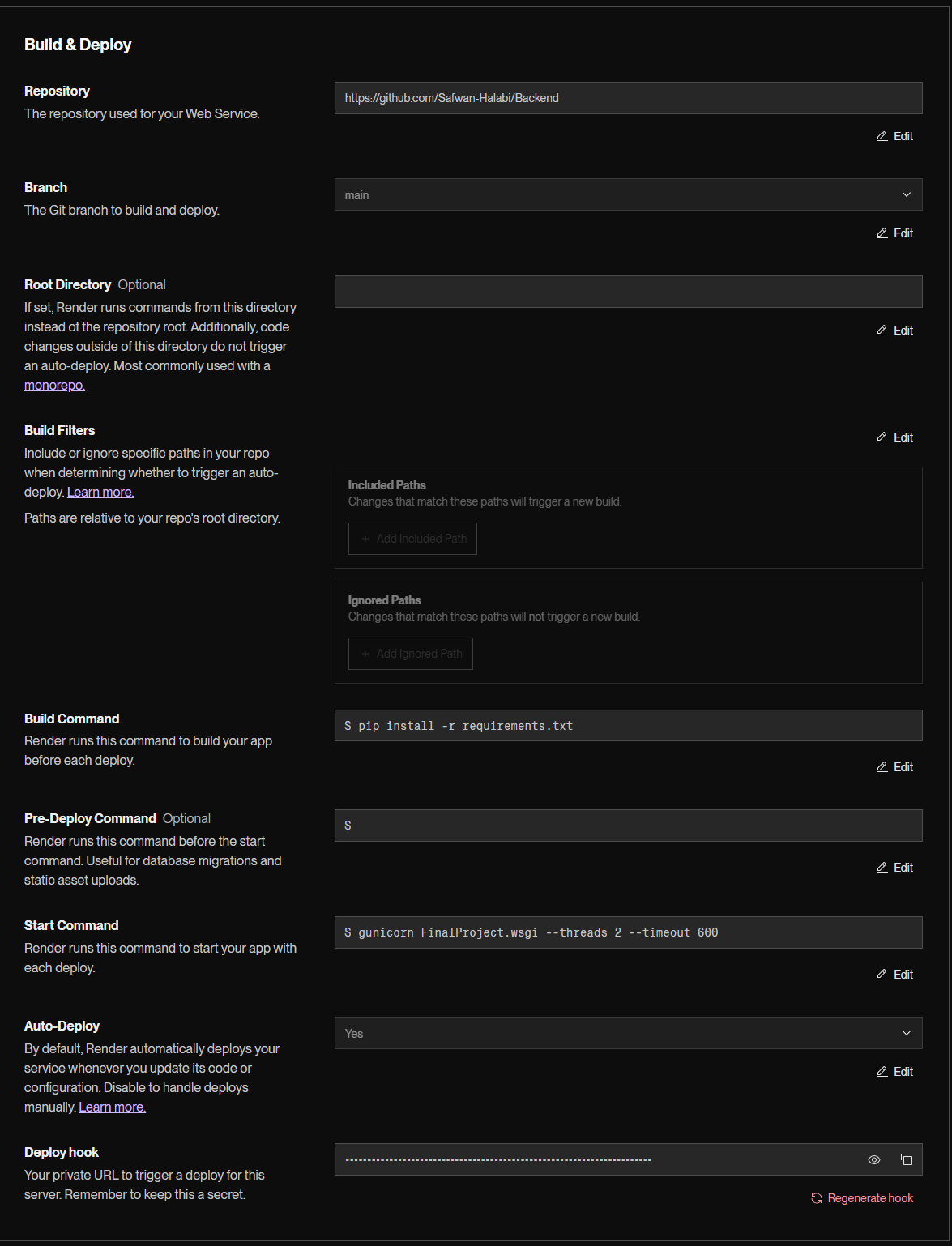
### 5.3.6 Settings

**General section:**



The **Settings** section is the main configuration point of the entire deployment. We will go over all the settings presented in the section. In the image above, General settings show the Name, Region and Instance type of the deployment. Currently, the instance type is set to standard, which can be upgraded to higher tiers, offering better resources for the website to run (higher CPU and RAM).

**Build & Deploy section:**



For the **Build & Deploy** section, the important parts are:

* **Repository:** The Github repository that contains the project
* **Branch:** The branch that will be used to deploy the project.
* **Root Directory:** Specifies the path to the source files of the entire project. For this project, it is not needed to specify, as the source files are placed at the base level of the directory.
* **Build command:** Command that runs before deploying the project. It is necessary to install the required libraries. In this project, libraries are detailed in the requirements.txt file.
* **Start Command:** The project uses *gunicorn* library, it’s best to run the command: “gunicorn FinalProject.wsgi --threads 2 --timeout 600”
* **Auto-Deploy:** Specifies whether to deploy the project automatically upon pushing changes to the branch used to deploy the project. Default set to Yes

For **Custom Domains**, **PR Previews**, **Notifications** and **Health Checks**, no changes have been made.

# External Links and Documentation

For further documentation and more information about the following subjects, visit the links provided below:

* The Django Framework - [Django Documentation](https://docs.djangoproject.com/en/5.1/)
* Python - [Python Documentation](https://docs.python.org/3.11/)
* OpenCV - [OpenCV Documentation](https://docs.opencv.org/4.x/index.html)
* Librosa - [Librosa Documentation](https://librosa.org/doc/latest/index.html)
* Parselmouth - [Parselmouth Documentation](https://parselmouth.readthedocs.io/en/stable/)
* Soundfile - [Soundfile Documentation](https://python-soundfile.readthedocs.io/en/0.11.0/)
* TensorFlow - [TensorFlow Documentation](https://www.tensorflow.org/api_docs)
* MongoEngine - [PyMongo Documentation](https://pymongo.readthedocs.io/en/stable/)