IMPaCT Intelligence

Workplace Violence (WPV) Prediction Implementation Manual

Prepared by: Christopher Duym
Version: 1.0
Date: 04/26/2024
Sponsors: James Corbett (IMPaCT Intelligence), Arend van der Veen (WBAT Safety)

1 Introduction

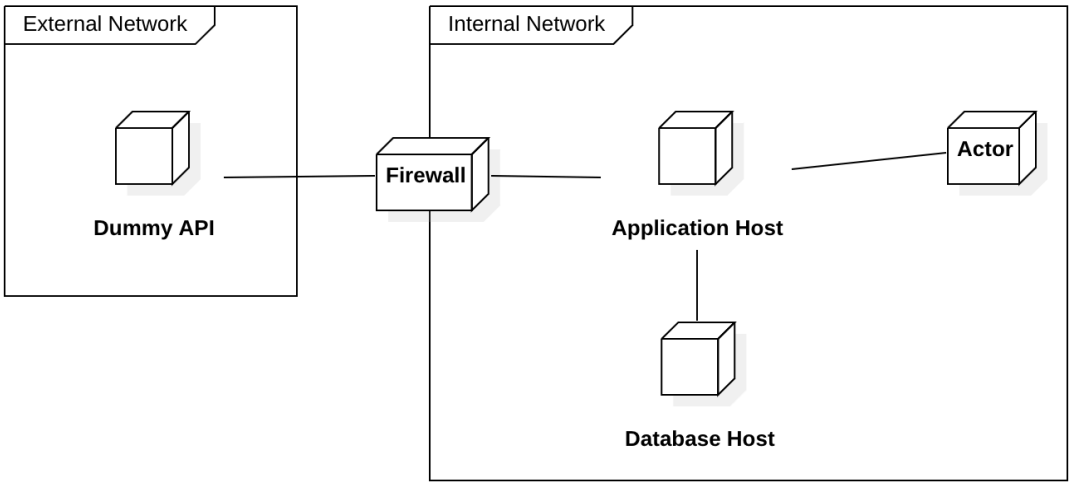
1.1 Purpose

Workplace violence in the healthcare sector is a persistent problem without obvious solutions. Our mission is to aid hospital administration in mitigating the factors that lead to workplace violence incidents before such an incident occurs. In order to do so, we developed a system that monitors various conditions in a hospital system and then utilizes a machine learning model to identify patterns in the conditions that result in workplace violence incidents. Our system gives hospital staff tools to identify conditions for workplace violence before such incidents actually occur.

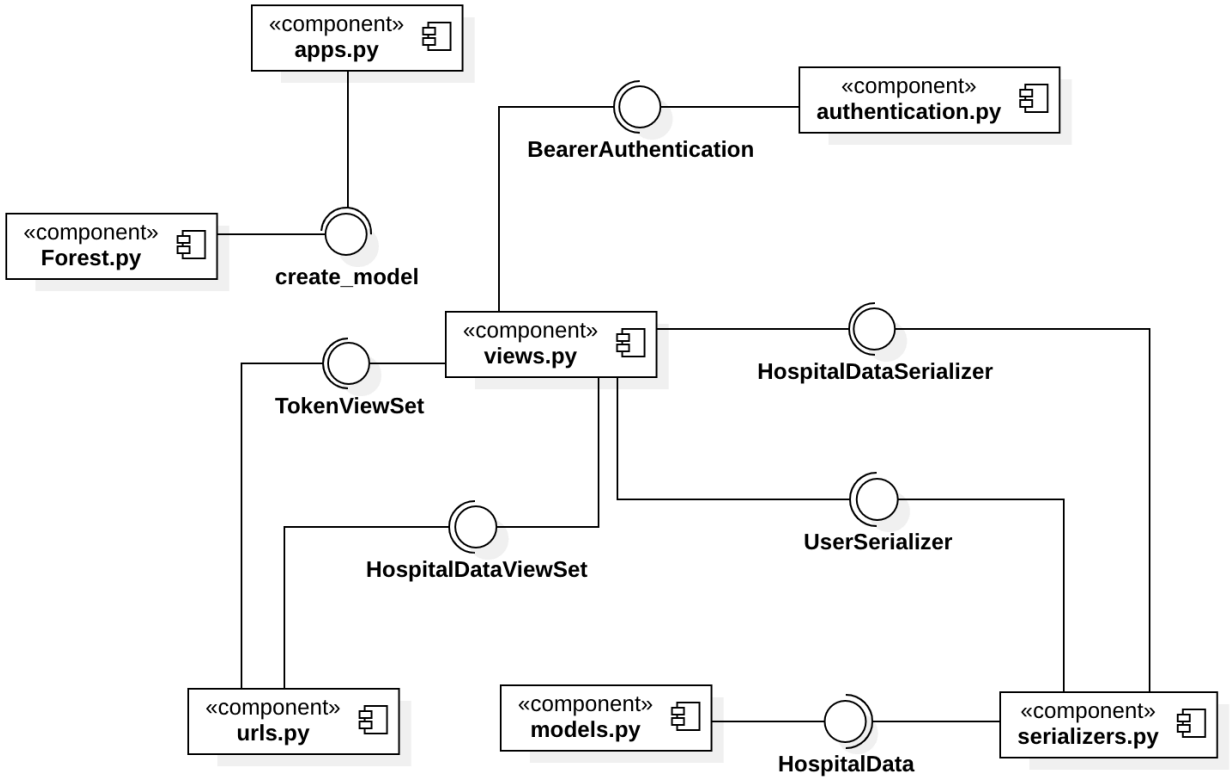
1.2 System Description

The WPV system is based in the Django Rest Framework, a Python based framework for developing web applications. To store our data, our system incorporates a MariaDB database that is hosted remotely. MariaDB serves as an open-source alternative to MySQL and provides better query speeds for our data retrieval. The machine learning model which makes predictions was implemented as a Random Forest classification model using the open source Python library Scikit-Learn. Due to the inability to obtain access to actual hospital systems, our team developed a separate dummy-API with Django. This API returns randomly generated hospital data when our main project sends a request to it. This allows us to simulate the polling of real hospital systems. Our front-end user interface was implemented using vanilla HTML, CSS, and JavaScript, and is hosted within our host application through Django Rest's view handling protocols.

A high-level overview of the system's structure is illustrated in the following diagram.



The following component diagram offers a more comprehensive overview of the system.



It is likely better to conceptualize each component as an independent file rather than a class due to the nature of the project. Django provides a level of abstraction that does not require the code to follow a strict class/object relationship. Views.py acts as a central communication hub for the entire project. Much of the functionality is defined within various views in this file, and the other files either refer directly to views or make API calls to it. Views.py also handles the API endpoints, as items such as the GUI make API calls to views within this file to be able to access functionality defined elsewhere in the project.

1.3 Assumptions and Constraints

The following assumptions were made when determining the implementation procedures for the system.

- The Client in charge of implementation must be knowledgeable about deploying projects built with the Django-Rest Framework using Docker.
- The Client must be knowledgeable on how to alter and configure a MariaDB database.
- The Client must have access to the hospital systems that they are integrating this system into.

- The Client must know the intended data points for the system to collect from the hospital systems, as well as the format that the data will be in.

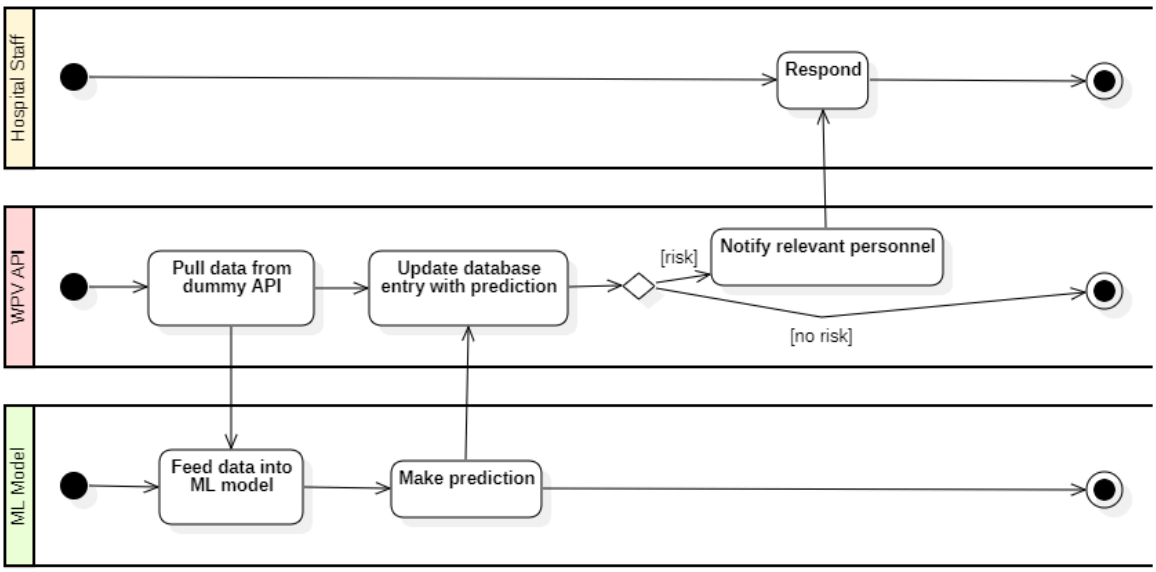
The following are potential constraints to the implementation of the system within a hospital environment

- There may be compatibility issues between the Django-Rest framework and the APIs available to connect to the hospital's systems.
- There may be issues formatting the data to match the criteria required by the currently implemented machine learning model.

1.4 System Organization

The Django application has a very linear workflow. On a set interval, the application makes a request for sample data to the dummy-api. The Django application stores that data into our database in the hospital_data table. It then takes that data from the database and sends it to the machine learning model, which then makes a prediction of workplace violence levels and then stores it in the risk_data portion of our database. The Django project then repeats this process until the application is closed.

This workflow can be seen illustrated in the following figure.

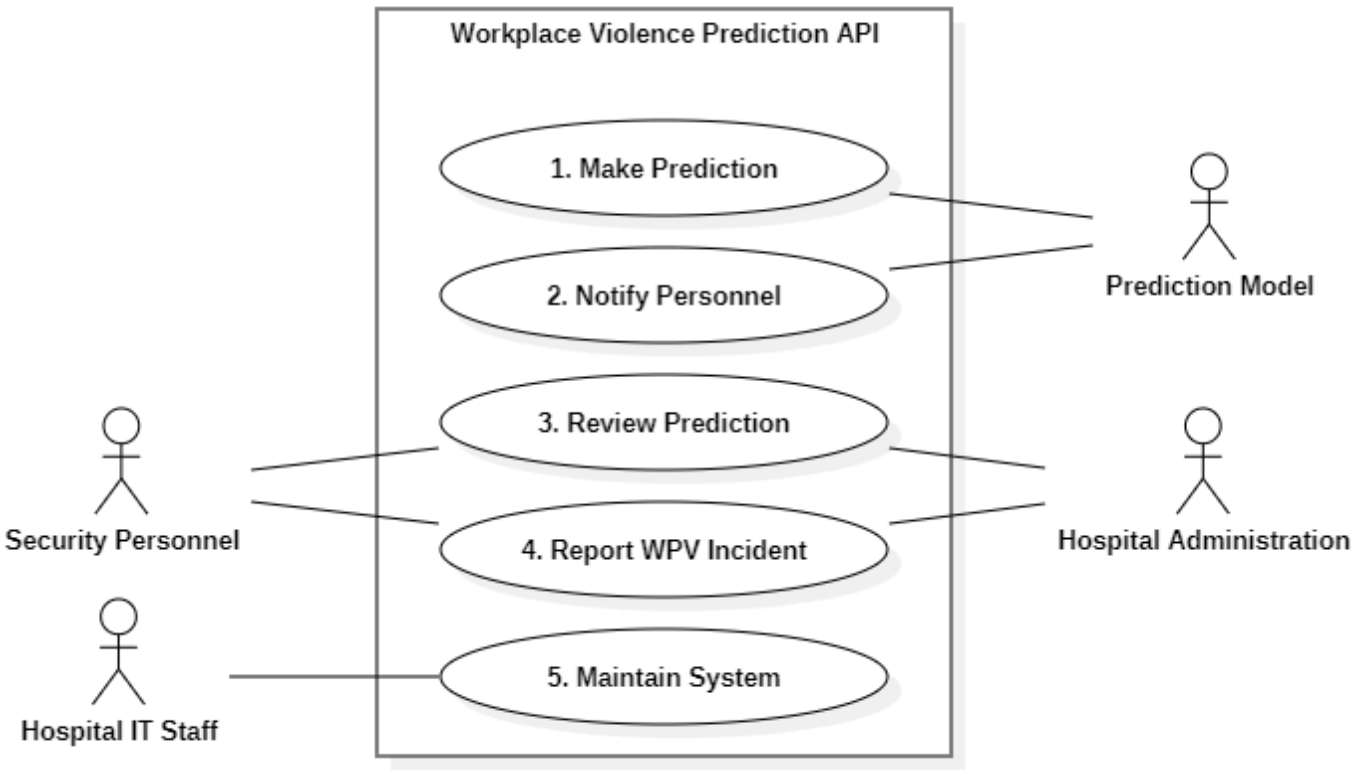


The database that has been configured for this application has several tables described in the following table:

Table	Description
training_data	Contains a large set of records which are pre-labeled as either WPV or non-WPV. This data is used to train the ML model on startup.
hospital_data	Stores data retrieved from the dummy-api. This data is similar to the data in training_data except that it is not labeled with risk values until it has been analyzed by our machine learning model.
risk_data	Stores prediction data from the machine learning model for a specific data point. Every risk_data entry is linked to one specific hospital_data point.

Our graphical user interface provides a few points of interaction between our system and the user. Our system is largely autonomous, as it collects data and makes predictions without being prompted by a user. Our GUI allows for the user to view the current risk level, manually log WPV incidents that actually occurred, and manage the emails on the emergency notification list.

The general use case for the system is depicted below:



2 Implementation Methodology

This section describes the necessary information before beginning the implementation of our system.

2.1 Implementation Environment

The system was developed within the Django-Rest Framework and then Dockerized. By Dockerizing the system, any operating-system based dependencies have been removed that may have been caused by development on our local machines.

2.2 Dependencies

This section describes the dependencies that the system relies on to operate as intended.

User Authentication

The system relies on Bearer Token Authentication to ensure only qualified users are attempting to access the system. The System Administrator must ensure that they create credentials within the system for their Software Engineers, and that the Software Engineers have their own unique token.

Installation Dependencies

All dependencies are listed in the requirements.txt file within the Django project.
In order to use the WPV API, you must run the following command:

```
pip install -r requirements.txt
```

within your PowerShell terminal, as this will install all of the libraries that were utilized in the development of our system.

Hospital Systems

Our system was developed without access to hospital systems which are currently in production. Currently, our system polls an outside API that we created with specific specifications and data formats in mind. The polling system for the project will have to be modified to format and store the data it gathers from an actual hospital system into a format usable by the machine learning model.

2.3 Configuration Requirements

The following configurations within the Django project must be set before attempting to use the system.

config.toml

This file sets up the user authentication for use with the system. The various fields are described in the table below:

Group	Description
[database]	All four of these fields must be set to establish a connection to the database. - Username of the system administrator. - Password of the system administrator - Database's name - Database's host IP
[auth]	Stores the bearer token for authentication

Each software engineer should have their own version of this file including their specific credentials. This file should be omitted from version control systems that are hosted on public webpages to prevent web-scrapers and unauthorized users from gaining access to API login credentials.

2.4 Implementation Procedures

This section describes the steps and procedures for the proper implementation of the system.

Creation of the Database

The development database can be used as a template for the creation of the production database, however the fields within the hospital_data and training_data table spaces must be altered. After determining what factors will be tracked during your utilization of the system, the previously mentioned tables should be set up as follows.

- The ID field should be the primary key, and should be an automatically generated, auto-incrementing integer value.
- Each field after the primary key will represent one of the data-points being collected by the system that will be used to make a workplace violence prediction.

- In the training_data table only, there should be a final field named wpvRisk; this field will hold the pre-classified workplace violence value, as this data will be used to train the machine learning model.

Incorporating the Database in Django

Within the Django project is a file named *models.py*. This file contains the python representation of the tables held in the database. Now that the database has been configured to work with the data that is intended to be collected by the system, we have to configure the model of the database to match it.

The hospital_data model should include a variable for each field in the database. Each variable should be initialized with:

*model.**DATATYPE**(db_column="**FIELDNAME**")*

The information in bold should be replaced by the correct information pertaining to that specific field.

The TrainingData model should be constructed the same way. These models will be almost identical, except the TrainingData model should have an extra field named *wpvRisk*, which will represent the classification of the data's workplace violence risk.

Database Model Serializers

After the creation of the database models, each model needs a serializer. The purpose of serializers is to ensure that the data being fed into a Django model matches the format of the data required by the specified model. Each model should have a respective serializer created in the *serializers.py* file.

Data Collection from the Hospital Systems

Data collection by the system is handled autonomously through the *updater.py* and *tasks.py* files within the Django project. The updater.py builds a scheduler object using the apscheduler python library, and then adds tasks with predefined time intervals to that scheduler object. The tasks.py file defines the behavior of the tasks that are added to the scheduler object.

The *get_data()* method within tasks.py defines the task that acquires data from the external system, which we then serialize and save into the database. Currently, the system sends a get request (using the python requests library) to the external system, and then stores a json representation of the data into a variable. The external system

was developed to return data in the same format required for the serializer, and thus the whole application.

Most likely, the hospital systems will not return data in the same format required by the system. Within *get_data()*, the client software engineering team will be responsible for making the request to the external system and then cleaning the data into the appropriate format before attempting to serialize and save that data. The format the data will have to be cleaned into will depend on the structure of the hospital_data data table within the database that was created earlier. To match the serializer, the data points should be in the same order as the fields in the table, and the final data should be in JSON format. A serializer object will then be created, where JSON represents the name of the variable that the data was stored in.

serializer = HospitalDataSerializer(data=**JSON**, many=False)

Adapting the Prediction Model

Once the data points that are intended to be collected by the system have been set, the machine learning model must be altered to work with those data points. Within *forest.py*, the function *queryset_to_dataframe(self)* loops through every entry in the training_data table and creates a Pandas Dataframe that is usable by the model. To do so, the function creates a python dictionary of JSON objects, named *formatted_data*, where the members of each entry consist of the name of the field and the data pertaining to that field for a specific entry.

The contents of the function *formatted_data.append()* must be altered to match the fields within the database created by the software engineers. Each field should have its own entry in the JSON object, and should be followed by the data obtained by the query set. One field within the append function would look like this:

‘id’ = q[“id”]

Making a Prediction on the Collected Data

Within the *tasks.py* file, the *predict()* function takes the latest entry from the database and sends a request to the prediction model to create a prediction from that data. With the prediction model configured to work with the chosen data points, there should be no necessary changes to the predict() function.

2.5 Entry and Exit Criteria

This section describes the steps that must be taken before the implementation of our system (Entry Criteria), and the steps that should be taken after implementation to ensure that it is working correctly (Exit Criteria).

Entry Criteria

Access to the desired hospital system must have been acquired and confirmed before the beginning of implementation. The hospital_data and training_data tables of the database both contain fields that are specifically formatted for the data it is intended to collect, and thus must be altered to fit the data our system is retrieving from the hospital system.

Proper authentication credentials must be created by the System Administrator to ensure that the Software Engineers can access the database as intended.

Exit Criteria

Once the database has been successfully implemented, the software engineers will follow the steps outlined in the Implementation Procedures section. Once implemented, upon navigating to the home page, the dial-based view should appear on the left hand side, and the right hand side should contain a view of the database information, which will automatically update at the set polling interval. The Software Engineer can ensure that all of the functionality is working correctly by using the buttons to navigate to the email management page and adding / removing an email address, and by attempting to log an incident on the logging page.

3 Roles and Responsibilities

The following table depicts the responsibilities of the client personnel maintaining the system.

Position	Responsibilities
Client Software Engineer	● Create the database according to the specified parameters. ● Ensure valid connection between the database and the system. ● Modify the system's code base to be able to utilize the API of the hospital system.
Client System Administrator	● Create valid user credentials in the database for the software engineering team. ● Ensure proper bearer token authentication is configured for each user.

4 Project Schedule

The Client and their Software Engineering team will implement our system within phase two of this project, allowing for the implementation of the system with real hospital data and real hospital systems.