IMPaCT Intelligence

Workplace Violence Prediction API

Design & Architecture Document

Prepared by: Avery Bobbitt
Version: 1.0
Date: 3/19/24
Sponsors: James Corbett (IMPaCT Intelligence), Arend van der Veen (WBAT Safety)

Introduction

Purpose

A design and architecture document serves as a foundational resource for a development team, offering a detailed roadmap for building a software system. It provides insights into the overall structure of the system, including its components, modules, interfaces, and interactions. By documenting design decisions, patterns, and trade-offs, it helps developers understand the rationale behind certain architectural choices and empowers them to make informed decisions during implementation. Moreover, the document facilitates collaboration among team members by establishing common terminology, design principles, and coding standards. This document will go over UML diagrams representing the design and architecture that is currently present in the **Workplace Violence Prediction API (WPV API)** application. This includes a description of the contents, its purpose, and why/to whom it is necessary.

Scope

The main goal of the **WPV API** application is to provide hospital staff and administration with a real-time monitoring service that notifies relevant personnel when the application detects an increased risk of workplace violence. The intended end-user of this application is the hospital staff, including: administration, IT technicians, and hospital security.

System Environment

The application requires a minimum version of **3.9** for Python. All later versions are supported. Currently, all other technologies used in this application do not depend on a specific version. The application can be deployed on any server running any operating system that supports Python. The development team also plans to implement containerization if necessary.

Architectural Views

Logical View

A logical view provides a high-level abstraction of the system's functionality, focusing on the structure and behavior of its components without delving into implementation details. It serves as a blueprint for understanding the system's conceptual organization, interactions, and data flow. This view is crucial as it facilitates communication between the development team and management by presenting a clear and coherent representation of the system's functionality, aiding in decision-making processes and ensuring alignment with business objectives.

While simple, the logical view for this application provides a comprehensive, high-level overview of the system functionality. There are 2 main components: the application layer and the data source layer. The application layer contains the Django application itself, which includes the API endpoints (correlating to various Django views), the machine learning model, the Python representation of the data model, and serializers for the model. The endpoints are accessible within the internal network by hospital staff. The machine learning model is only accessible by the application itself, thus it is depicted "sandwiched" between other components. The serializers for the Pythonic data models communicate directly with the database when creating or updating data. The data source layer consists of the data access logic and the database itself. Only the serializers write to the database directly, as the data must be serialized to ensure it is in the correct format.

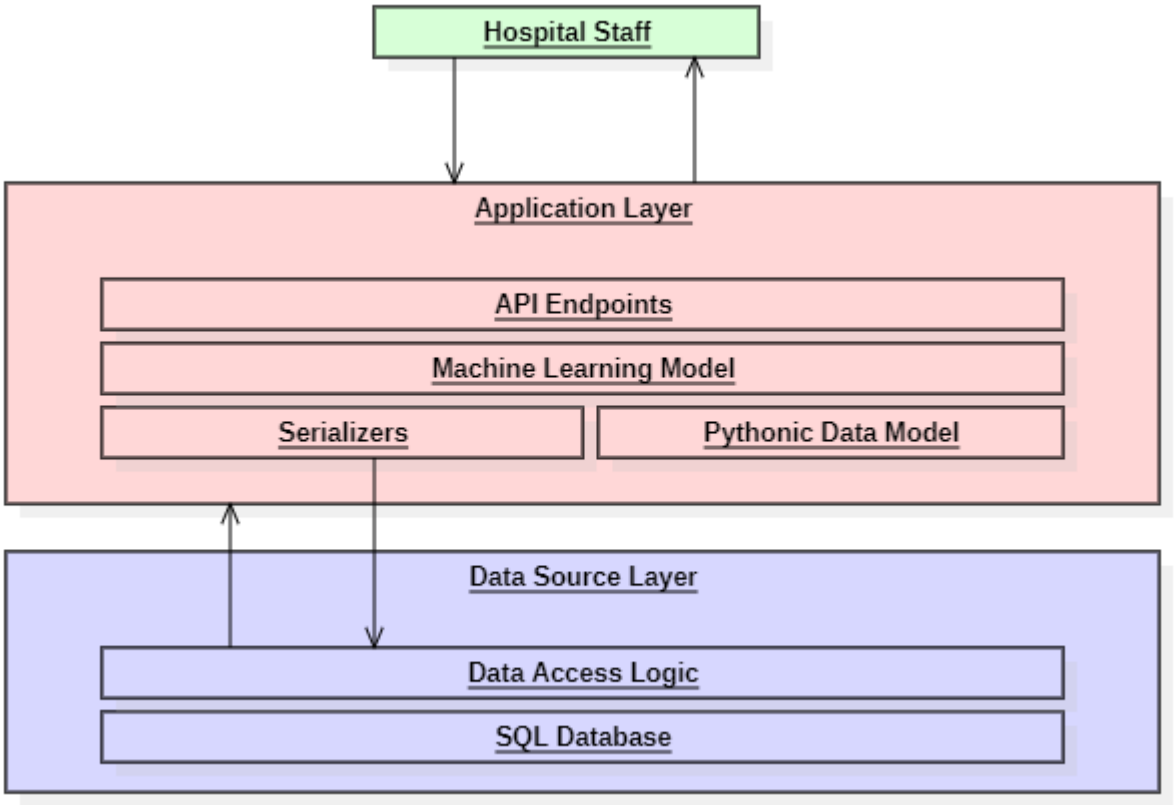

Figure 1.1: Logical view diagram depicting abstracted layers of the application

Physical View

The physical view of the system architecture shows the hardware involved in the system and how software components are distributed across processors. Used by system engineers and developers, the physical view depicts physical constraints and requirements of the system, such as hardware dependencies, network topology, and scalability considerations. By visualizing the physical layout of components and their interactions, stakeholders can assess factors like performance, reliability, and resource utilization. The physical view also aids in system deployment and maintenance: it provides guidance on how to install, configure, and manage the system's hardware and software components in real-world environments, and enables staff to make informed decisions, address technical challenges, and ensure the system meets its functional and non-functional requirements in real-world scenarios.

The physical view for this application consists mainly of 3 servers: the dummy API, the application host, and the database host. The database can optionally be deployed on the same host, if necessary. The application host is the center of the system; it contains the API process and the machine learning model. It connects to the database host, which hosts the database, to retrieve and send data, and it connects to the external dummy API. The dummy API, hosted on a personal remote server, is meant to simulate pulling data from a real source. The application makes a HTTP request, and the dummy API returns simulated, statistically accurate data.

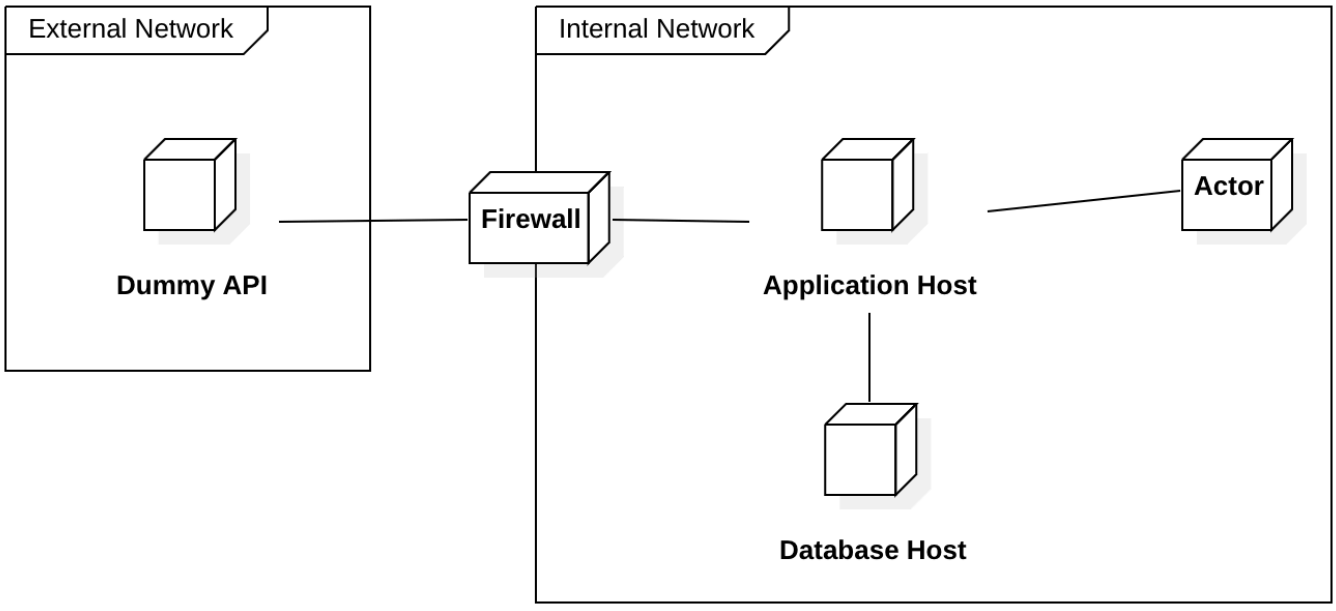

Figure 1.2: Physical view diagram showing a very high-level view of the physical systems

Development View

The development view diagram provides a high-level visualization of the architectural components and their interactions within a software system, offering a blueprint for developers to understand the system's structure. It serves as a roadmap for implementing and evolving the system by delineating modules, interfaces, and dependencies. It depicts a system from the standpoint of a programmer and is mainly concerned with software administration. This diagram aids developers in comprehending the system's complexity, facilitating efficient collaboration and informed decision-making throughout the development lifecycle. Its utility extends beyond developers to architects, project managers, and stakeholders, providing a shared understanding of the system's architecture and fostering alignment towards achieving project goals.

The development view for this application contains 9 elements. Due to the complexity and extensivity of the Django REST framework, this diagram is limited to only the elements relevant to this application and the parts within it. The URL router is at the core of the “controller” piece regarding the MVC architecture and uses the Django DefaultRouter. It depends on the different views and registers them with an API endpoint. The views inherit from Django’s general ViewSet class, which provides nothing on its own except very standard view functionality. ModelViewSet, also from Django, inherits from the base ViewSet class, providing default implementations for standard CRUD (create, retrieve, update, and delete) operations. TokenViewSet provides functionality for a user to retrieve an existing authentication token or create a new authentication token, while HospitalDataViewSet provides most of the functionality related to manipulating the hospital data entries. Both views require a serializer to serialize incoming data before adding it to the database. UserSerializer and HospitalDataSerializer inherit from ModelSerializer which, similar to the ModelViewSet class, provides default implementations of standard functionality. Much of the implementation has been omitted from the diagram for brevity, however the important functions are included. Finally, the Forest class represents the machine learning model. It provides all the methods and fields to create and retrieve information from the trained model. Similar to the ModelSerializer, much of the inheritance tree of the Forest class has been omitted for brevity - everything that the Forest class depends on comes directly from the SciKit-Learn Python package.

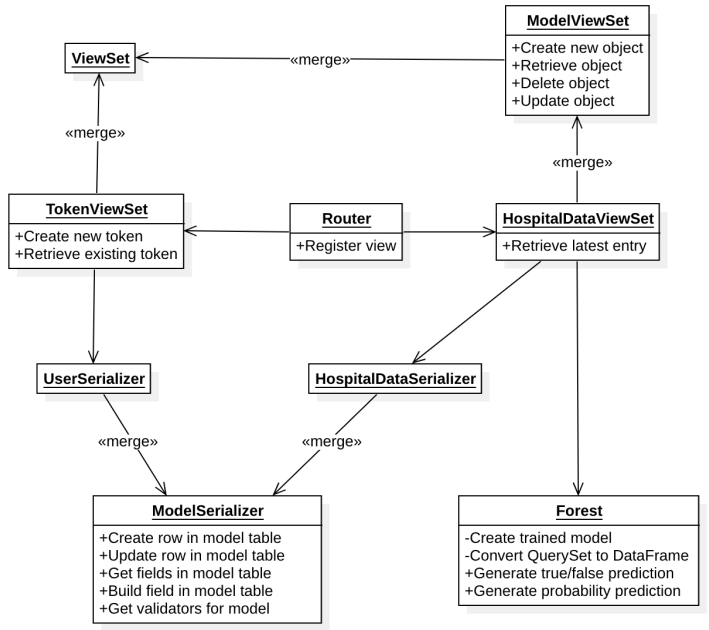

Figure 1.3: Development view diagram depicting a high-level view of the classes involved in the application

Process View

A process view provides a high-level overview of the steps involved in a particular process or system, focusing on the sequence of actions and interactions between components or stakeholders. It is essential in design and architecture documentation as it offers a clear understanding of how different elements come together to achieve a desired outcome, aiding in identifying bottlenecks, redundancies, and opportunities for optimization. This view is particularly valuable for project managers and stakeholders, enabling effective communication, decision-making, and alignment of objectives throughout the project lifecycle.

The process view diagram for this application consists of 3 components: the application, the hospital staff, and the machine learning model, each represented by their respective swim lanes. The application begins the process by requesting a new set of data from the dummy API (again, simulating pulling real data from a hospital system) and passing it to the machine learning model. The machine learning model receives the information and feeds it into itself, generating a prediction of whether the current conditions of the hospital can result in a risk of workplace violence. This prediction is sent back to the application, where the data entry in the database is updated with the prediction value. If the model predicts an increased risk, the proper staff are notified via email and are instructed to respond appropriately to the potential incident. Otherwise, the system does *not* notify any staff and continues regularly pulling new data.

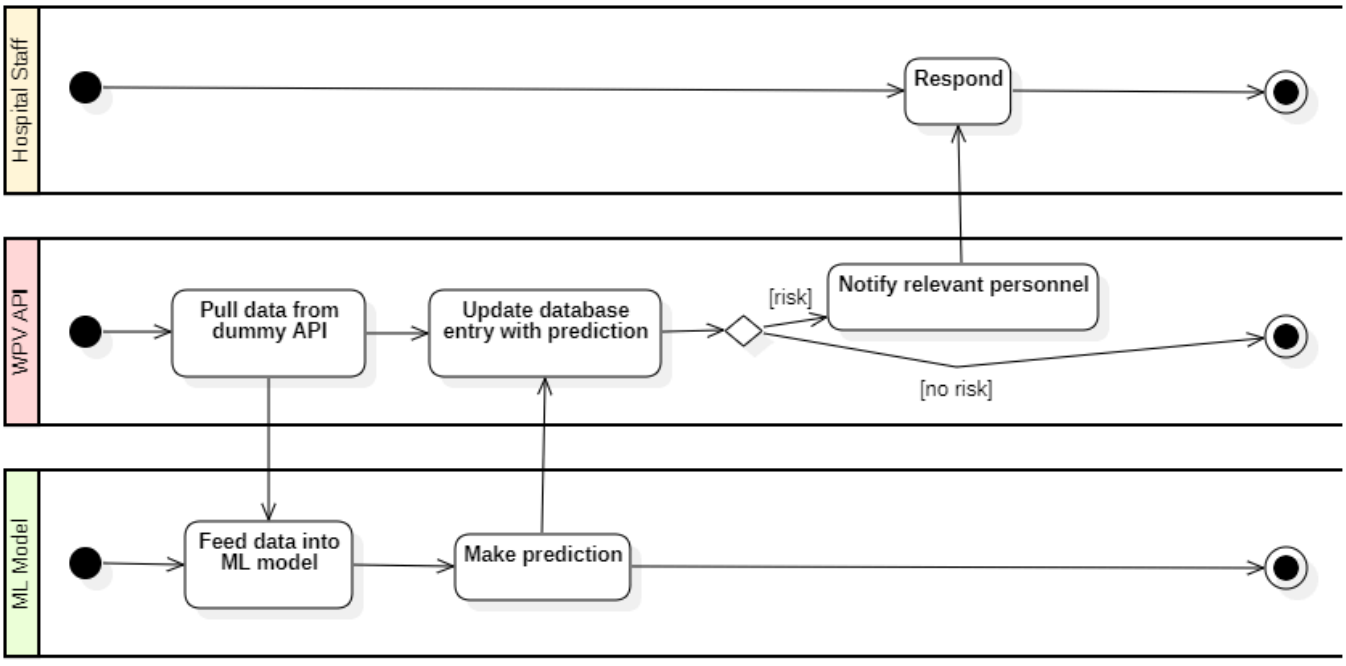

Figure 1.4: Process view diagram showing the sequence of events when the system retrieves a new data entry

Design Models

Structural Models

Class Diagram

A class diagram is a foundational element of object-oriented design, depicting the structure and relationships among classes in a system. It serves as a visual representation of the static view of a software application, outlining the various entities (classes), their attributes, methods, and the associations between them. This diagram is indispensable during the initial stages of system design as it aids in understanding the system's architecture, facilitating communication among stakeholders, including developers, architects, and project managers. Its utility extends throughout the software development lifecycle, providing a blueprint for implementation, aiding in code maintenance, and serving as a reference for system documentation. Ultimately, the class diagram acts as a vital tool for both technical and non-technical stakeholders, ensuring a shared understanding of the system's structure and behavior.

The class diagram for the WPV API is similar to the development view diagram, but with a few key differences. First, the class fields and their respective types are included in the diagram, along with return types for the methods. Also regarding methods, the actual method name is used instead of a short description of the method's purpose. The elements depicted in the class diagram are a more concrete and discrete overview of the classes in the application as opposed to the high-level, slightly more abstract interpretation of the development view.

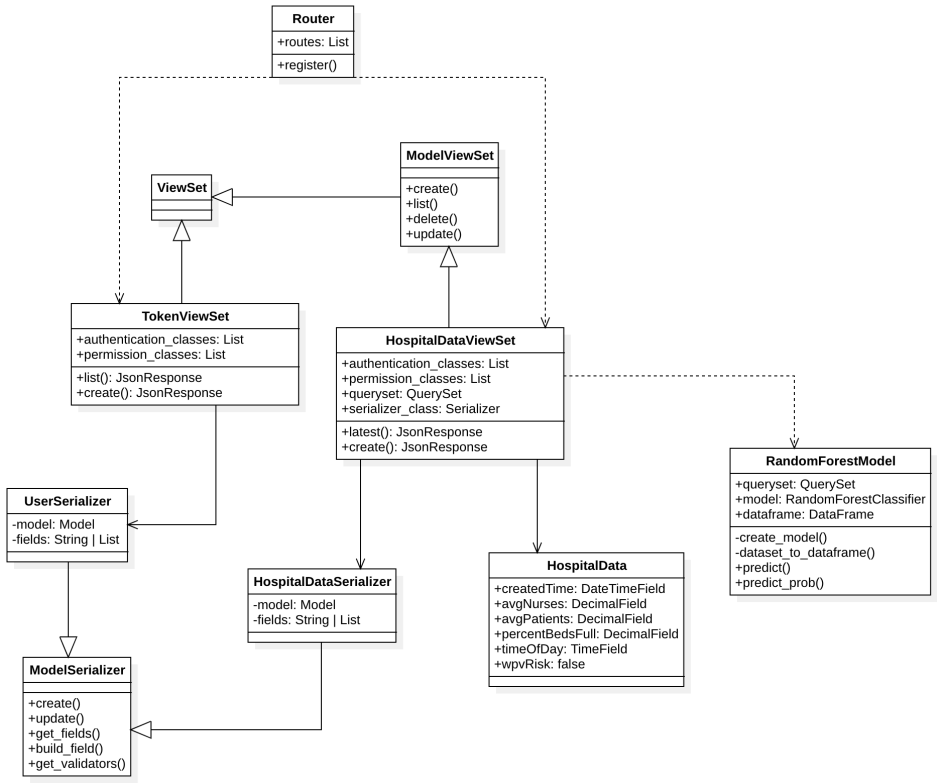

Figure 2.1.1: UML class diagram

Object Diagram

An object diagram is a visual representation within the Unified Modeling Language (UML) that illustrates the structure of a system at a specific point in time. It portrays the objects in the system and their relationships, showcasing how they interact to fulfill functionalities. This diagram serves a crucial purpose in software design and architecture by offering a tangible snapshot of the system's dynamics, aiding in the comprehension of complex systems, and facilitating communication among stakeholders, including developers, designers, and project managers. Object diagrams are particularly useful during the design phase, allowing teams to identify potential design flaws, validate system behavior, and ensure alignment with project requirements.

This object diagram illustrates the type of HospitalData that would typically be inserted into the database. Note that fields are represented in the JSON format and “02.24.00” represents “02:24:00” due to limitations in the UML diagram software.

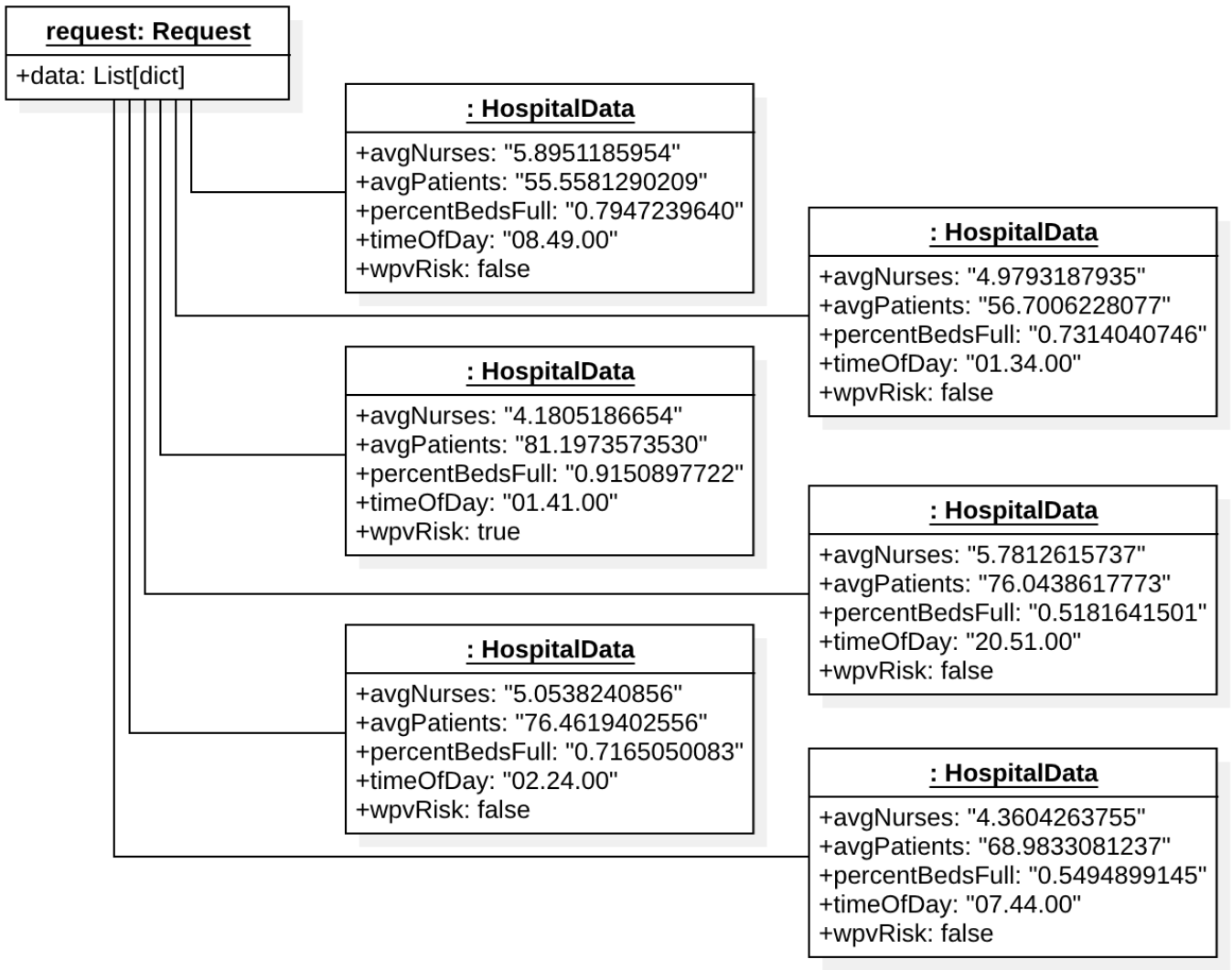

Figure 2.1.2: UML object diagram

Analysis/Robustness Diagram

A UML analysis or robustness diagram is a crucial component of the early design phase in software development, providing a high-level overview of the system's functionality and interactions. Its purpose is to depict the essential elements of the system, such as actors (external entities interacting with the system) and use cases (functions the system performs), in a simplified manner. By illustrating the system's behavior and relationships, the diagram helps stakeholders, including developers, designers, and project managers, to gain a clear understanding of the system's requirements and functionalities. This visual representation serves as a communication tool, facilitating discussions among team members and ensuring alignment between stakeholders regarding the system's design and functionality.

Bridging the gap between the Use Case diagram and the Sequence diagram, the robustness diagram for this application provides a high-level overview for how actors interact with the system and how the system interacts with itself. Each element is represented by one of three types: entity, control, and boundary. An entity is associated with the “model” part of the MVC architecture, as seen with the **database** and **machine learning model**. A boundary is something that the actors interact with directly, similar to MVC’s “view”. Due to the nature of the application, there are very few boundary elements. The **GUI** for the application is the only one depicted in this diagram, and although it is not yet implemented, it will serve as the tool the hospital staff uses to report an incident and review previous incidents. Finally, a control element, or “controller” in MVC, is the glue that binds everything else together. The three control elements in this diagram are the **router** and **ViewSets**. The ViewSets directly communicate with the database and machine learning model to manipulate HospitalData objects and predictions, while also being referenced by the router to create API endpoints.

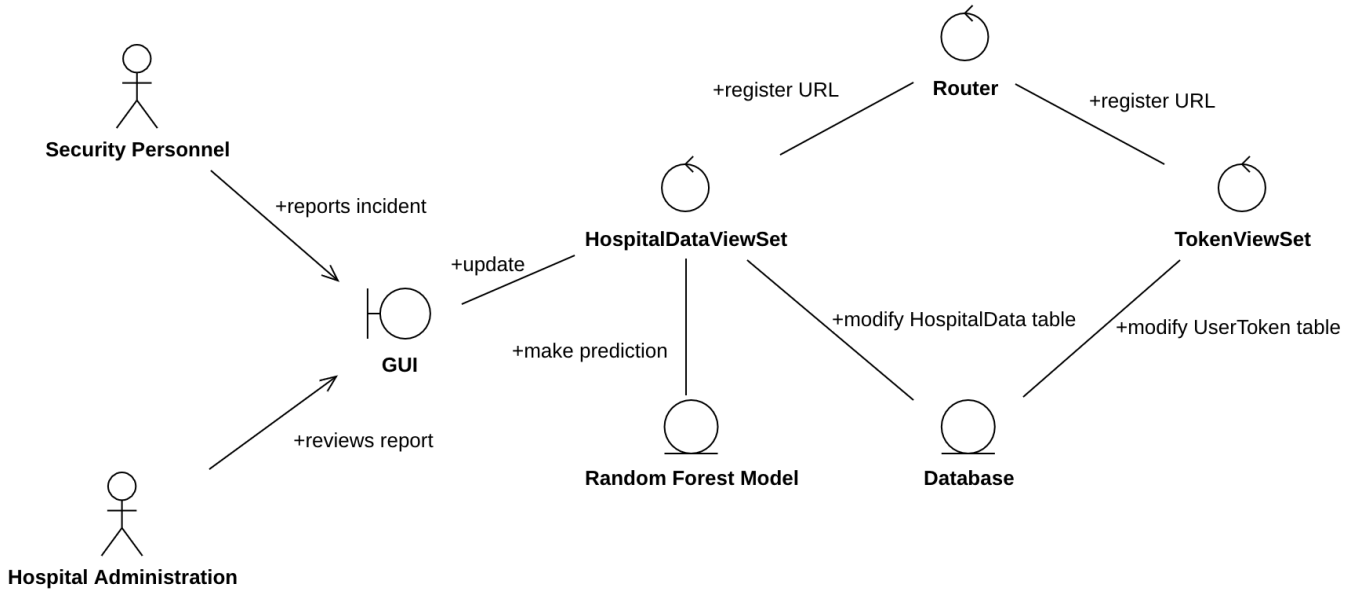

Figure 2.1.3: Non-UML-standard class analysis/robustness diagram

Behavioral Models

Use Case Diagram

Actors: Prediction Model, IT Technicians, Security Personnel, and Hospital Administration

The purpose of the use-case diagram in this scenario is to visually represent the interactions between different actors and the system, specifically the WPV API application. It outlines the functionalities available to each actor and their respective roles in the workplace violence prediction and response process.

The following use Case Diagram shows all the actions that the Actors above can perform while using the **WPV API** application. The prediction model, based on a random forest machine learning model, will read information from the database and make a prediction for whether there is an increased risk of workplace violence due to the current conditions in the hospital. If it detects an increased risk, the application will send a notification to other actors. Security personnel and hospital administration will then review this prediction, respond if necessary, and report whether an incident has occurred or not. Finally, if necessary, the IT staff will repair and do maintenance on the systems involved. It is also the responsibility of the IT staff to maintain the WPV API application itself.

This diagram is necessary for several reasons. It provides clarity on the roles and responsibilities of each actor involved in the system; for instance, the prediction model's role is to analyze data and generate risk predictions, while security personnel and hospital administration are responsible for reviewing these predictions and taking appropriate action. Additionally, it helps in understanding the flow of information and actions within the system, ensuring smooth coordination and communication between actors. Serving as a reference for system developers, IT technicians, and other stakeholders involved in the design, implementation, and maintenance of the WPV API application, it aids in the identification of requirements and potential improvements.

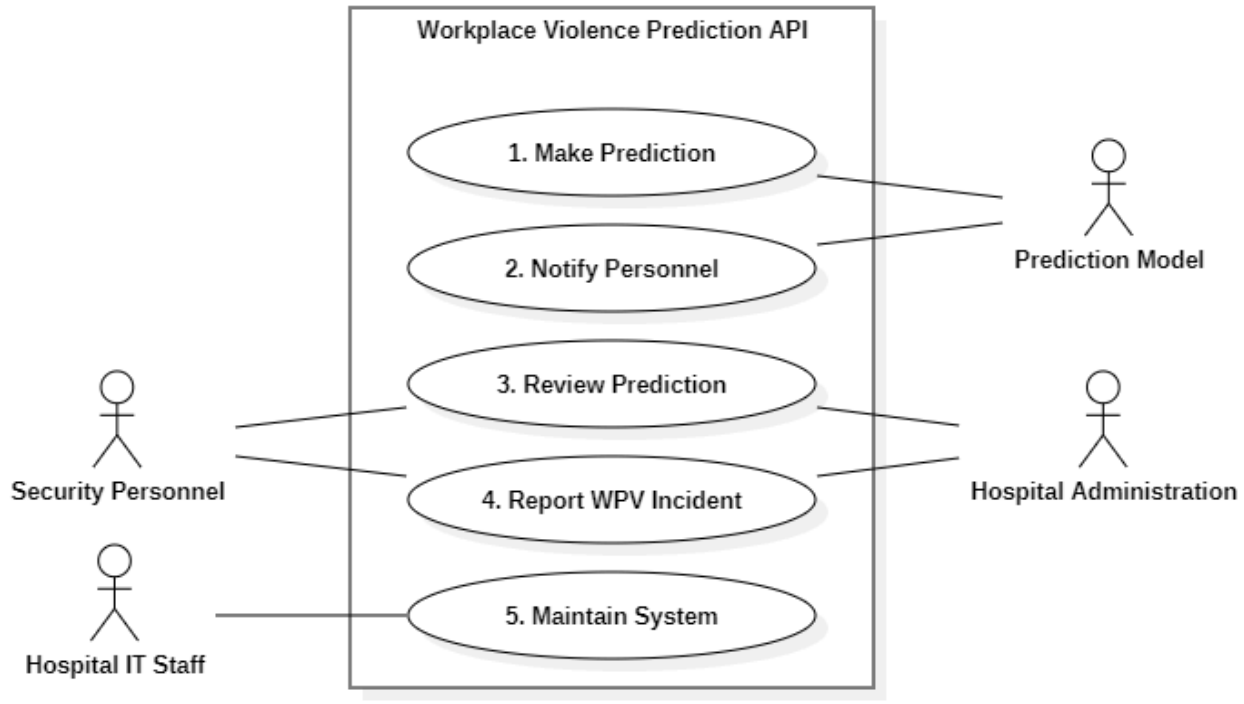

Figure 2.2.1: UML use case diagram

Sequence Diagram

The sequence diagram is a powerful tool within the realm of software engineering and system design. It serves as a dynamic representation of interactions between objects or components in a system over time, showcasing the flow of messages or events between them. Its purpose is manifold; firstly, it aids in understanding the behavior of a system by illustrating the sequence of actions and their temporal dependencies. This clarity is invaluable during the design phase, facilitating communication between developers, designers, and stakeholders. Additionally, sequence diagrams assist in identifying potential bottlenecks, optimizing system performance, and validating the correctness of the system's logic. They are particularly useful for software architects, developers, and quality assurance teams, providing a visual blueprint to guide the implementation process and ensure the alignment of the final product with the intended design.

The diagram below depicts the sequence of events for the main focus of this application: getting data and making a prediction on whether there is an increased risk of workplace violence. It begins by receiving data from the dummy API, then the view uses the serializer to serialize the data and save it to the database. The machine learning model then reads that entry from the database and makes a prediction. The serializer updates the database with the prediction, and if it is determined that a risk exists, the appropriate personnel are notified.

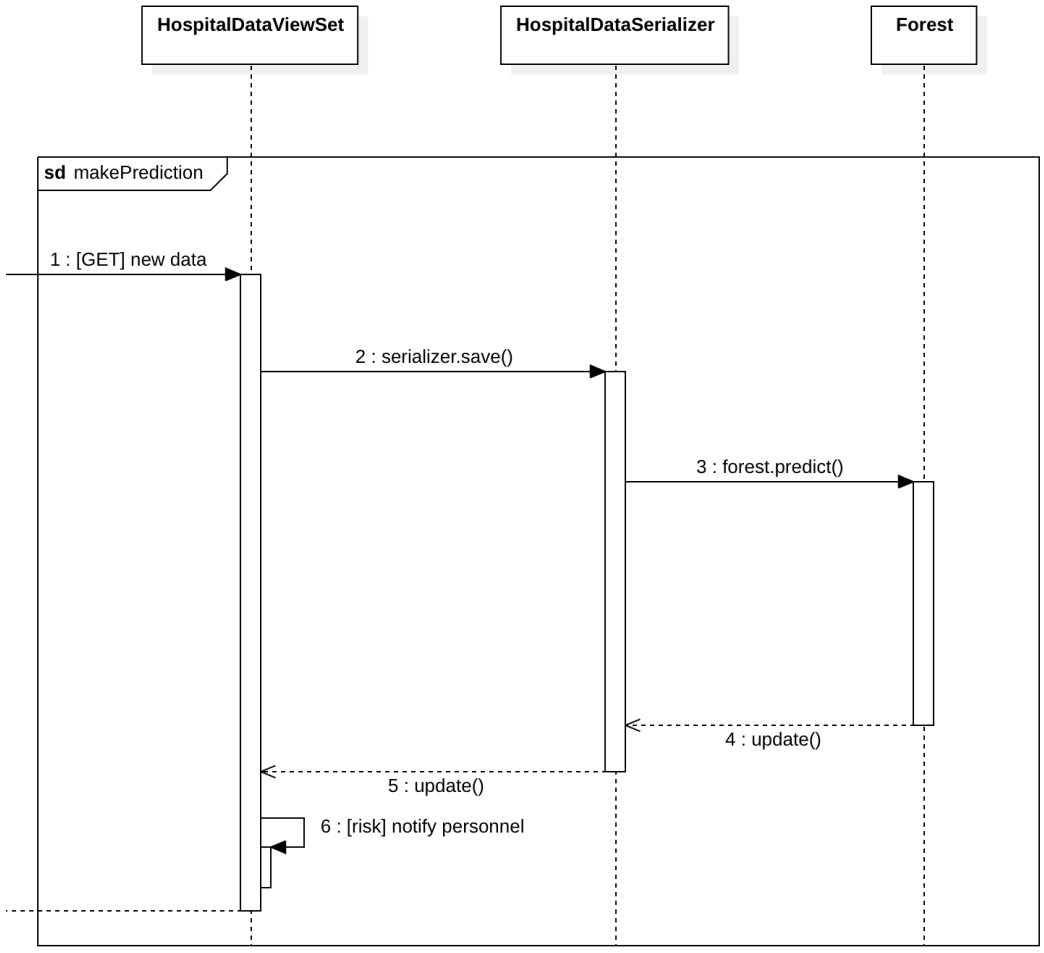

Figure 2.2.2: UML sequence diagram

Activity Diagram

An activity diagram is a visual representation that illustrates the flow of activities within a system or process. It showcases the sequence of actions, decisions, and interactions between components or entities. Similar to the process view, it provides a more thorough depiction of the system's behavior, aiding in comprehension, communication, and analysis of complex processes. Activity diagrams are pivotal in software design and architecture as they help developers and stakeholders visualize the system's workflow, identify potential bottlenecks or inefficiencies, and refine the design to enhance performance and user experience. They are invaluable tools for software engineers, designers, project managers, and stakeholders alike, fostering collaboration and ensuring alignment throughout the development lifecycle.

The activity diagram for the WPV API consists of 4 swim lanes: IT technicians, hospital administration, security personnel, and the application itself. It is important to note that the application is designed to run continuously without constant human interaction; the following diagram represents a single instance of an incident occuring. The activity begins with the technicians checking the status of the system. If the system is working with no issues, a member of the hospital administration logs in and assess the prediction. The prediction will either notify security personnel or do nothing when a workplace violence incident is more likely to occur or not, respectively. Upon being notified, the security personnel will investigate the potential incident and report their findings. If the system is inoperable, the member of administration can submit a request to perform maintenance. Then, an IT technician performs the necessary maintenance and reports the incident.

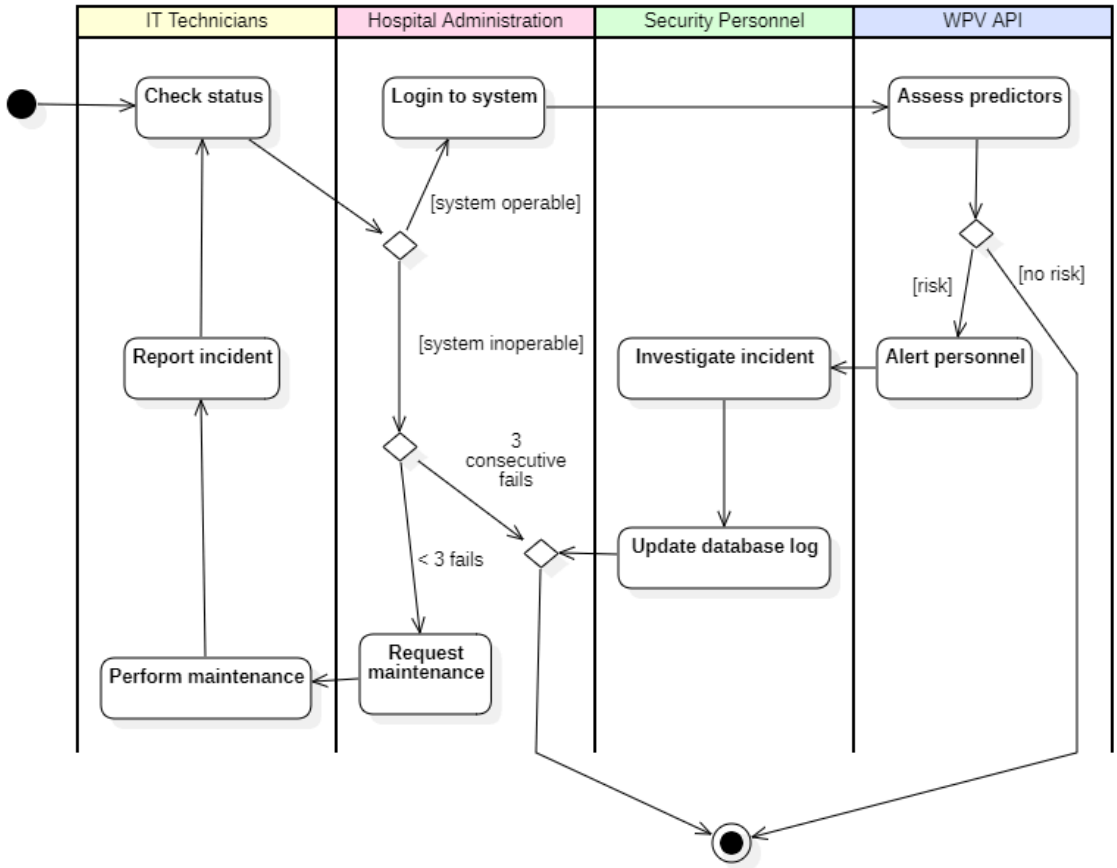

Figure 2.2.3: UML activity diagram

State Diagram

A state diagram, also known as a state machine diagram, is a visual representation used in design and architecture to illustrate the various states an object or system can exist in, as well as the transitions between those states. Its primary purpose is to provide a clear understanding of the behavior and functionality of a system, helping designers and developers identify potential issues, refine logic, and ensure comprehensive coverage of all possible scenarios. State diagrams are particularly useful for software engineers, system architects, and project stakeholders, offering a structured way to analyze and communicate complex system behaviors, thereby facilitating more efficient development, debugging, and maintenance processes. By visually mapping out states and transitions, stakeholders can gain insights into system behavior, aiding in decision-making and ensuring alignment with project requirements and user expectations.

The primary stateful piece of this application is the HospitalData object. This serves as the core object for the database that represents the conditions of the hospital at a given time. There are 3 states for a HospitalData object: not evaluated, risk, and no risk. Once the data is retrieved (the first step in the sequence diagram), it is stored in the database ready for the random forest model to read it. The model then reads the data from the database and makes a prediction on whether those conditions can cause an increased risk of workplace violence. If it is determined that **yes**, there is an increased risk of violence, then the row in the database corresponding to that HospitalData object is updated with a "Risk" value. Otherwise, if it is determined that **no**, there is not an increased risk of violence, then the row in the database corresponding to that HospitalData object is updated with a "No Risk" value.

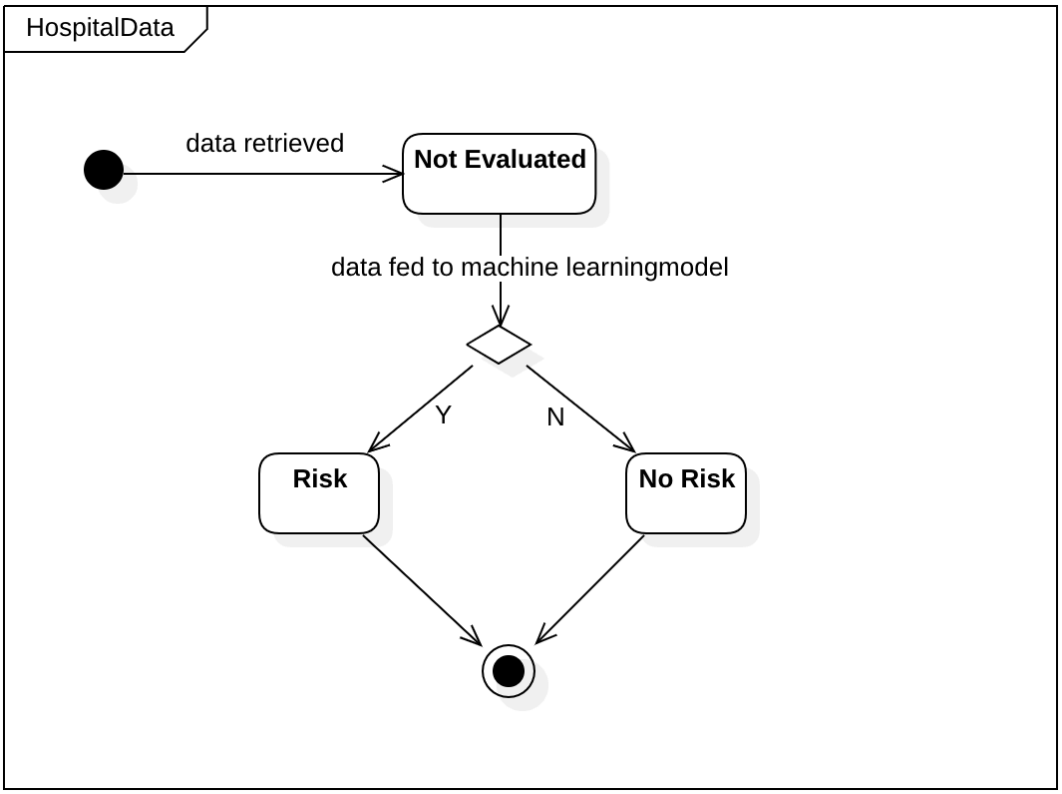

Figure 2.2.4: UML state diagram

Architectural Models

Component Diagram

A component diagram serves as a visualization delineating the high-level structure of a system and the interconnections between its constituent components. Its purpose is to explain the system's modular organization, depict the relationships among components, and provide a blueprint for system development and maintenance. This diagram is indispensable for various stakeholders, including software architects, developers, project managers, and quality assurance teams. Architects leverage it to conceptualize the system's architecture, identify reusable components, and ensure adherence to design principles. Developers refer to it for implementing specific functionalities within designated components, fostering collaboration and maintaining code consistency. Project managers utilize it to streamline resource allocation, track progress, and mitigate risks. Moreover, quality assurance teams rely on it to devise comprehensive test strategies and validate system behavior across its constituent parts.

As the structure of this application is not as “object-oriented” as some other softwares, it may be more helpful to developers to view the components as files rather than objects. Each component depicted in the diagram either interacts with or offers an interface. In this diagram, the interfaces represent a function or class contained in the parent component.

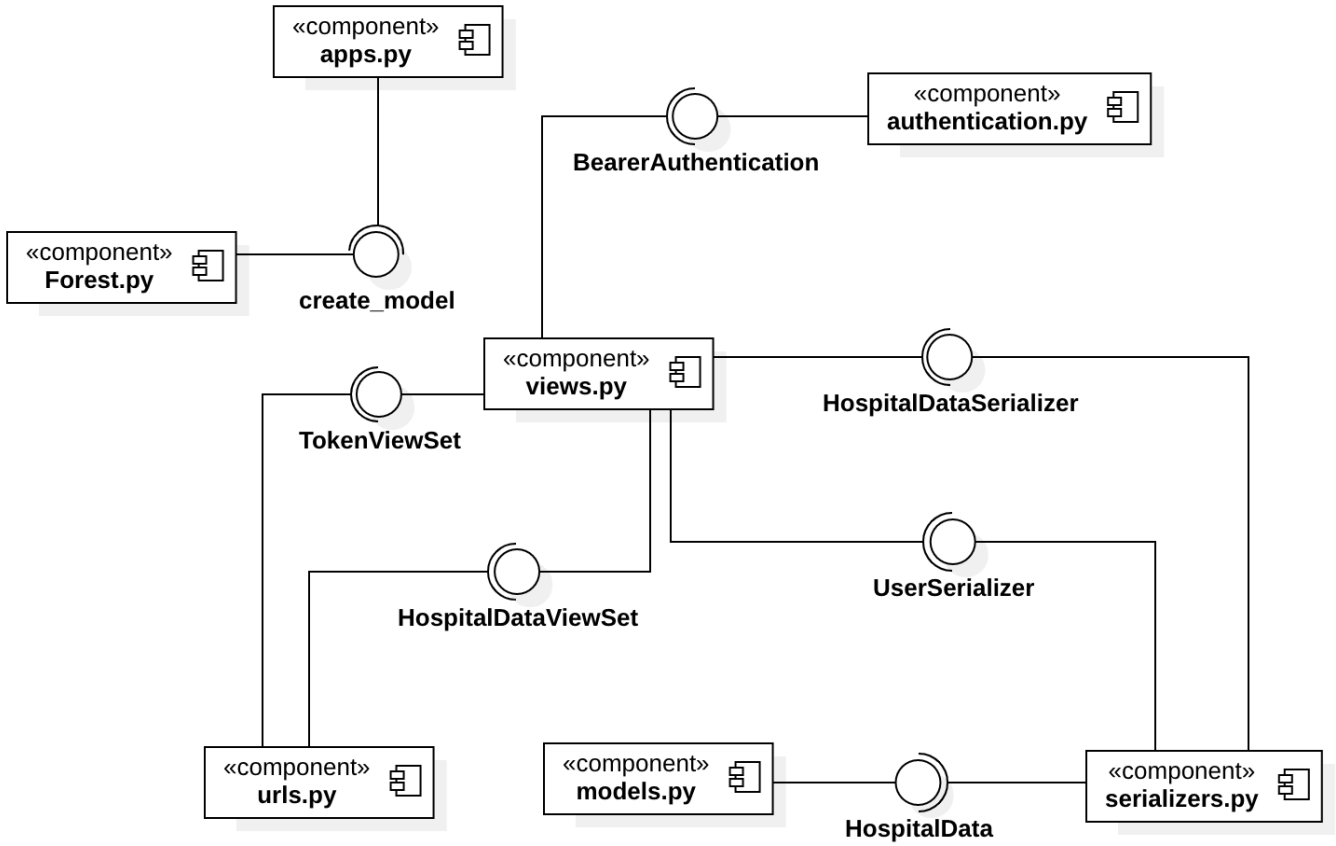

Figure 2.3.1: UML component diagram

Deployment Diagram

A deployment diagram serves as a visual representation of the physical deployment of software components across various nodes in a network. Its purpose lies in providing a comprehensive overview of how the system is distributed across different hardware elements, such as servers, clients, and databases, as well as how these elements interact with each other. This diagram is particularly useful for developers, system architects, and project managers, as it helps them understand the system's deployment topology, facilitating discussions around scalability, performance optimization, and resource allocation. Additionally, it aids in identifying potential bottlenecks and single points of failure, enabling teams to design robust and efficient deployment strategies.

The deployment diagram for the WPV API contains two groups of systems: internal and external. Internal systems are those that are deployed and host the application and database themselves, while external systems are the systems the WPV API communicates with. These external systems are assumed to exist, and are only shown in this diagram to give the reader a sense of understanding regarding where the API is sending and receiving data when not within the application. The WPV API WebServer is the device on which the API is hosted, and the DataServer is where the database is hosted. These devices communicate with each other over TCP (or a UNIX socket if deployed on the same system) to read and write data to the database. Externally, the API communicates with the dummy API using the HTTP protocol when simulating pulling data from a hospital system. The API also communicates with systems (only one depicted, but any others would act similarly) accessible to hospital staff to send emails over SMTP.

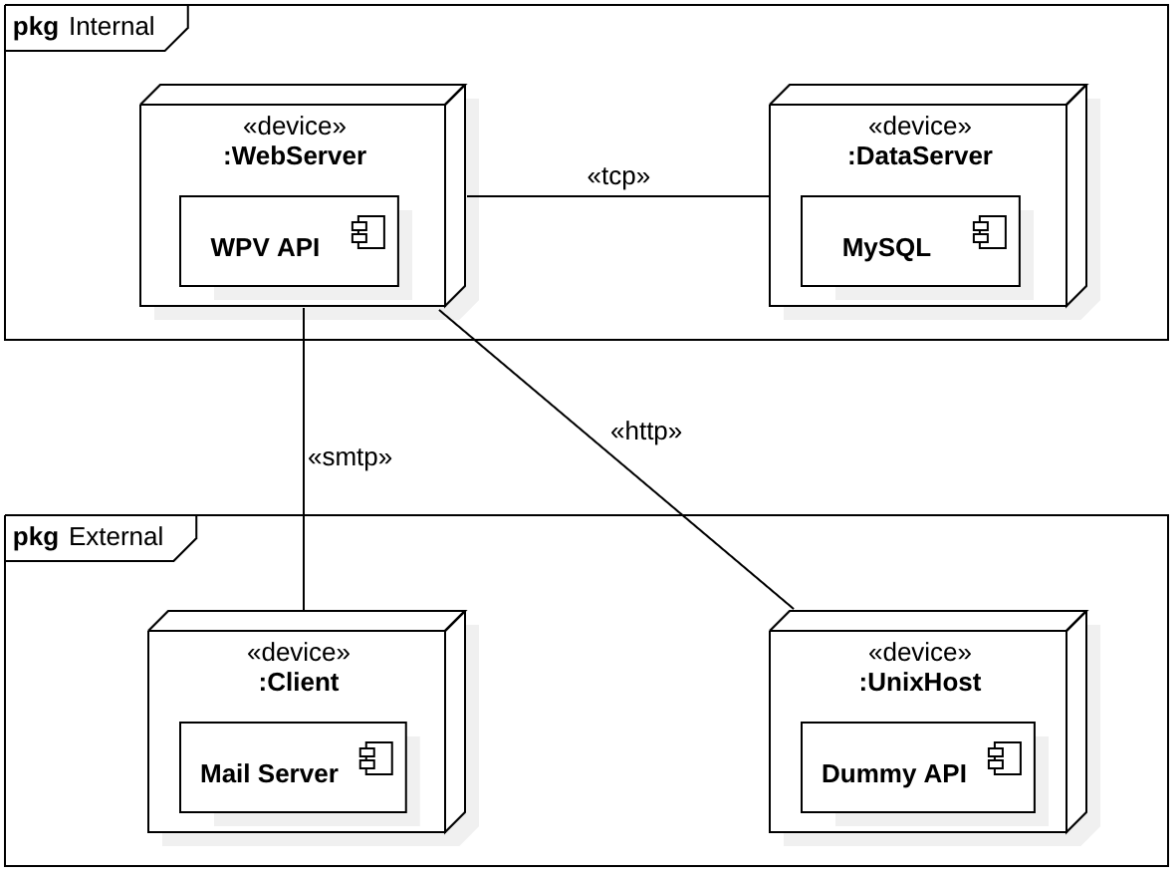

Figure 2.3.2: UML deployment diagram