Image removed for purposes of this excerpted documentation

5.2 Software Architecture

The CYBEX-P web application consists of several components that work in concert to provide a comprehensive software experience. These include a Python/Django back end and a React.js front end that communicate via a RESTful API. This web application back end is not to be confused with the CYBEX-P back end, which hosts the core API for querying the CYBEX-P database. The CYBEX-P back end and related services are the subjects of other works [35]; this Chapter focuses on the architecture and design of the web application only. Figure 5.3 is a system context diagram that visualizes some example interactions between backend data and frontend

Image removed for purposes of this excerpted documenation

Image removed for purposes of this excerpted documentation

elements. The intended functionalities of these components will be explored in more detail in Chapter 6.

5.2.1 Back End Architecture

The web application back end is built in Python using the Django web framework [12]. Django utilizes a MVT design pattern, consisting of model, view, and template components. The model drives all logic regarding the data in the application. The CYBEX-P web application's model incorporates Neo4j [30] graph-based databases to persistently store user investigation data. The web application's model also extends

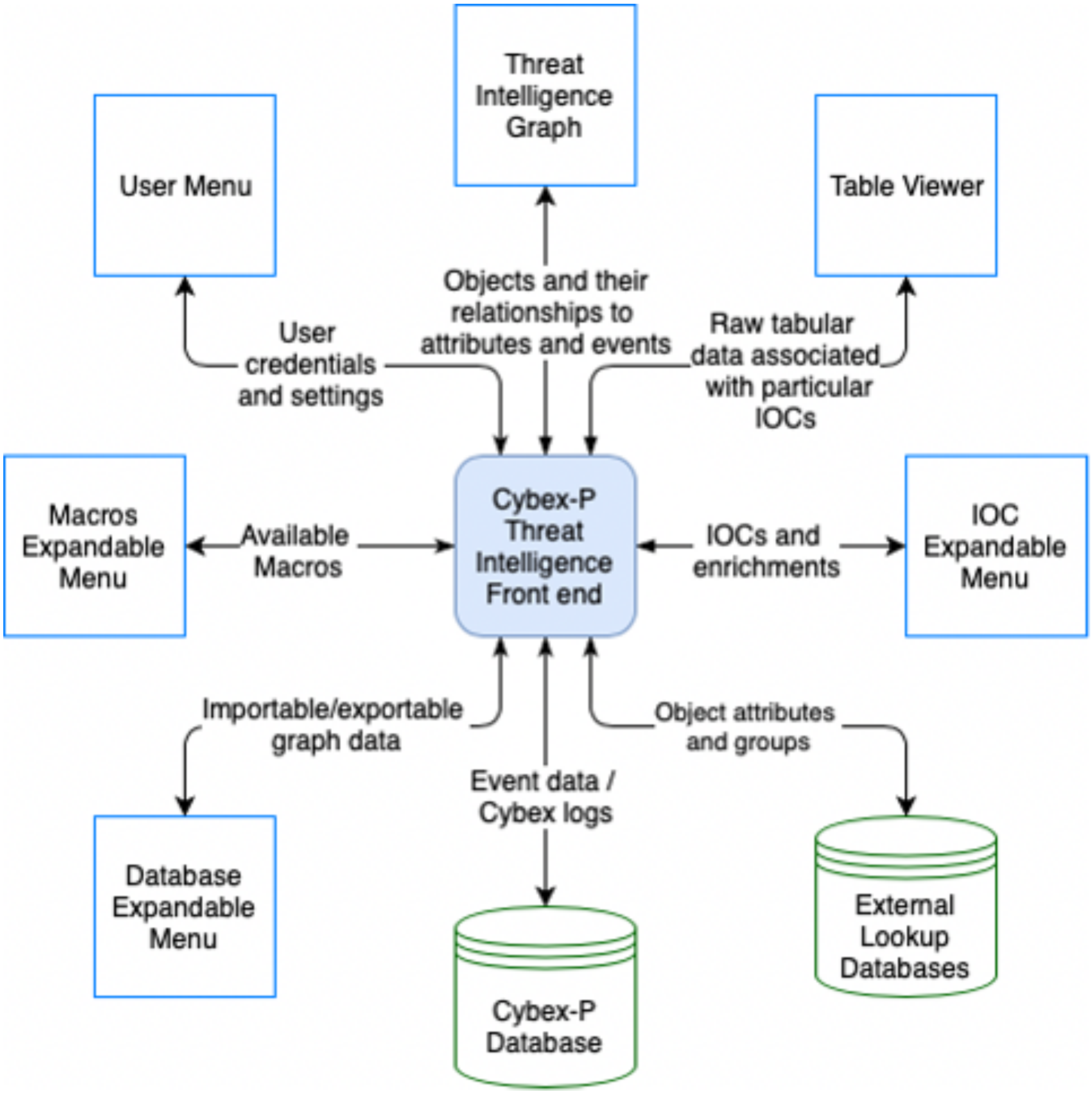
Threat Intelligence Graph
User Menu
Table Viewer
Objects and their relationships to attributes and events
User credentials and settings
Raw tabular data associated with particular IOCs
Macros Expandable Menu
Available Macros
Cybex-P Threat Intelligence Front end
IOCs and enrichments
IOC Expandable Menu
Importable/exportable graph data
Object attributes and groups
Event data / Cybex logs
Database Expandable Menu
Cybex-P Database
External Lookup Databases

Figure 5.3: System context diagram illustrating the high-level structure of some of the application's interactions.

Django's built in user model (implemented using SQLite [43]) for user authentication and other user data. The application's templates describe instructions for generating and displaying both static and dynamic HTML output. Views represents the user interface that is displayed in the web page. They are responsible for taking data from the model and rendering it into templates. The views for this application are constructed using a number of other technologies that are detailed later in the next section.

This application extends Django with the Django REST framework, which is used to serve the RESTful API that the front end will make requests to [13]. Anytime frontend users perform an action that requires data to be queried, the back end application is responsible for handling these requests. The back end may then need to make subsequent requests to other services to gather the required data. This includes third-party APIs like Whois, GeoIP, and more. These lookups power the standard enrichments that are detailed in Chapter 6. Similar requests are made to the CYBEX-P API for specialized enrichments that reveal threat contexts. Once the Django application processes these data, it passes them back as responses to the frontend so that the results can be interacted with.

As mentioned previously, the Neo4j graph database must also be communicated with. Unlike the prior examples, this is not an API request to an external service. The Neo4j database exists on the same server as the web application back end. This database directly represents the data of the user's current graph. Separate instances of this database are isolated within Docker containers for each unique user [14]. This allows the application to save and persist every user's graph data in between usage sessions. Direct queries are made to the current user's Neo4j database within their corresponding container. These queries either modify the database or simply return its current graph representation so that it can be processed and rendered in the web application client. Figure 5.4 visualizes the architecture described in this section. See Figure 2.5 for a summary of the web application's position within the larger CYBEX-P platform architecture.

5.2.2 Back End Deployment

The Django application is deployed on a web server that allows it to be accessed publicly. The application is executed using a Gunicorn application server. This is a HTTP web server gateway interface that allows the Django application to be run as multiple concurrent python processes [19]. Such a configuration is important because several users may make requests to the system at the same time. A NGINX web server is also used to put the Gunicorn process behind a reverse proxy. This allows the application to be executed on the server's localhost, disconnected from direct outside requests. Instead, NGINX takes the responsibility of fielding requests from clients, serving as a middleman that then directs requests to the local application. This approach has multiple advantages, allowing load balancing of client requests and increasing security for the backend services [31]. Lastly, Supervisor is used to control and monitor the Gunicorn process [44]. This is used to start, stop, and reload the application. Supervisor constantly monitors process status and can restart the application if it crashes or if the server gets rebooted. A summary of the aforementioned deployment details is depicted in Figure 5.4.

5.2.3 Front End

The frontend of the web application consists of two parts. The first is a homepage that is rendered statically using HTML5/CSS. The homepage consists of mulitple static HTML files that are served as a series of templates from the Django backend. This site serves as an informational home for the CYBEX-P project, allows users to sign in with their accounts, and acts as a portal to the main application

The majority of the front end is contained within the threat intelligence graph application. This is a single-page React application built using 'create-react-app' [34]. This means that rather than having several page links to different application views, only one view is used. Components are the building blocks of the user interface that form this single page. Components are defined by a collection of logic and visual properties that together serve some functional purpose. For example, the navbar at

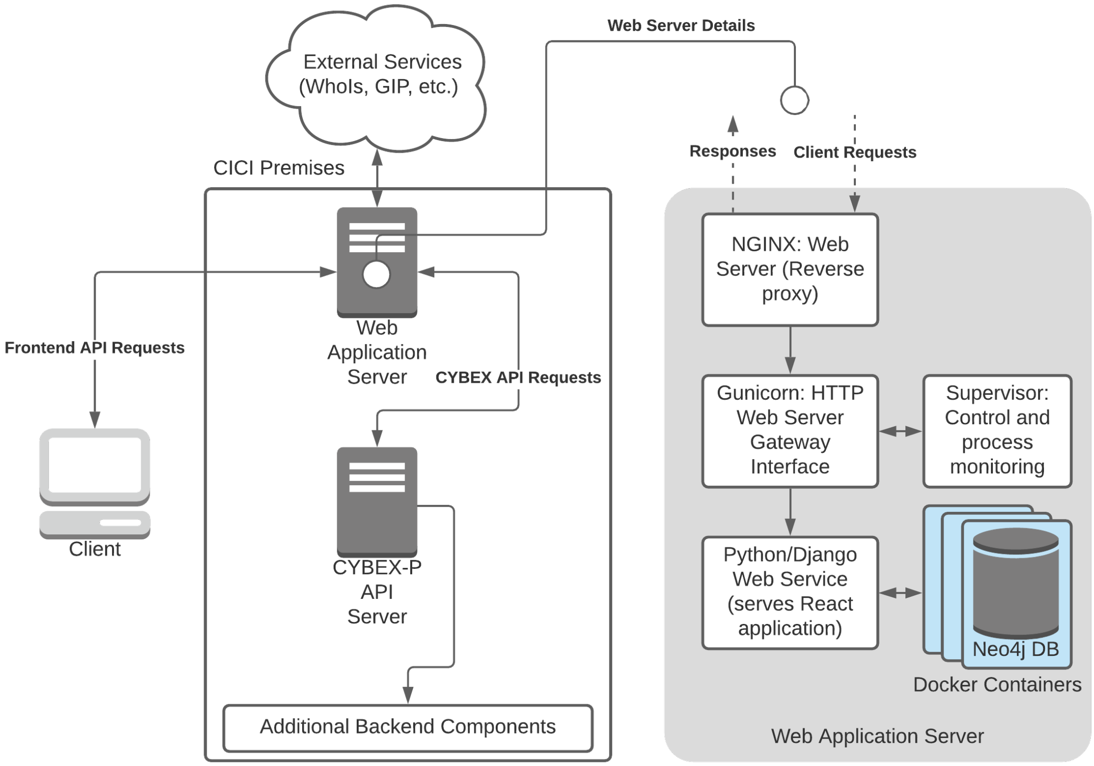
External Services (WhoIs, GIP, etc.)
Web Server Details
Responses
Client Requests
CICI Premises
Frontend API Requests
Web Application Server
CYBEX API Requests
Client
CYBEX-P API Server
Additional Backend Components
NGINX: Web Server (Reverse proxy)
Gunicorn: HTTP Web Server Gateway Interface
Supervisor: Control and process monitoring
Python/Django Web Service (serves React application)
Neo4j DB
Docker Containers
Web Application Server

Figure 5.4: High-level architectural diagram for the CYBEX-P web application.

the top of the application facilitates a common set of actions, so it is designed as a component. Within it, there can be multiple child components, such as the main dropdown menu that is available to users. The navbar itself is a child of a master component which contains the entire application. The combined functionalities of these components are demonstrated in Chapter 6.

React can implement either class-based or functional components. The CYBEX-P web application opts for functional components that conditionally render their contents. This means that the UI will automatically and selectively render whenever application states change. State is an important React concept that describes the data components use. This data changes over time, and thus is referred to as 'changing state'. An example of a state variable could be the value of a dropdown, which changes when the user selects a new choice. The UI or backend logic may be expected to react or change according to this state change. Automatic rendering refers to the fact that the programmer doesn't define 'how' the components should visually update; rather, they just define 'what' the UI should look like in different scenarios. Conditional rendering means that UI will only re-render if state data it is directly attached to changes. This is far more efficient that re-rendering the entire page if just one value in one sub-component gets updated.

With few exceptions, most components are custom-built rather than using off-the-shelf component libraries. This includes writing custom JavaScript functions for component logic, as well as defining custom CSS to achieve the desired visual design. All project components are written using JSX, which extends standard JavaScript. JSX is designed to be used declaratively to describe the desired appearance of the UI. It can be thought of as incorporating HTML directly into JavaScript logic, and can be incredibly efficient for rapid UI iteration. All components in the CYBEX-P web application are written in separate JSX files that isolate their respective logic and styling. This pattern of development seeks to keep all aspects of components localized. This is different than alternative methods which abstract styling and logic away from the UI elements they describe. It is worth noting that React doesn't enforce any one particular design pattern (such as Model-View-Controller (MVC) or Model-View-ViewModel (MVVM)); however, the general approaches described in this section are standard best practice.

The central graph component is one of the more complex pieces of the application. To help implement visual graph representations, the vis.js graphics library [46] was used as a foundation. In its default form, this library was not sufficient for achieving the project's usability goals. As a result, the threat-intelligence graph customized the base functionality and appearance of this tool. The tailored integration of vis.js into this application had several benefits. It allowed the backend data and frontend visuals to have a one-to-one relationship, where data is represented in a graph data structure in both the Neo4j database and React code. The result is efficient visualization and modification of graph data, with changes concurrently reflecting in users' browsers and the backend database.