SUSTech HPC Monitor System Development

Liquan Wang
12011619@mail.sustech.edu.cn

Shuxin Li
12011438@mail.sustech.edu.cn

Zi'an Wang
12010117@mail.sustech.edu.cn

***Abstract*— SUSTech HPC currently has Taiyi and Qiming clusters with over 1000 machines to manage. So it's significant to develop a monitor and alert platform. We plan to develop the platform with the cutting-edge technology, middleware and Database.**

I. Introduction

Monitoring systems holds a crucial place in the management of supercomputing centers. The well-being of the devices is essential for the supercomputing centers' high quality service rending. Automated monitoring both help server administrators discover potential problems beforehand and enable attendance to errors in time.

The Center for Computational Science and Engineering of SUSTech[1] is in charge of two clusters namely Taiyi and Qiming with over 1000 servers, serving to large numbers of computing jobs from research labs. To maintain the computing center efficiently, a monitoring system that can monitor the health condition of the servers according to administrators' needs is required.

Various third-party monitoring applications had been developed for easy-to-use system management. Nagios and Zabbix are two classic monitoring systems widely adopted by supercomputing centers, both had been continuously developed for over 20 years. To meet the ever-growing needs of supercomputing centers along with the rapid development of computational power and demand, while also aware of the out-of-date problem of the older systems, emerged are new-generation monitoring systems such as Prometheus.

Proprietary systems are built on top of third-party monitoring applications to meet more specific requirements of supercomputing systems. In a survey of 10 supercomputing centers of Russia conducted by Vadim V. Voevodin et al. in 2021, almost half systems adopted proprietary solutions besides third party monitoring applications. Different supercomputing centers have different data mainly of interest, device management hierachy, alert system and user management. Therefore, customized monitoring systems supported by third-party core technologies are developed and deployed by supercomputing centers.

In this study, we aim to develop a monitoring system catering to the needs of the Center for Computational Science of Engineering of SUSTech. To keep our system in date with the newest softwares and devices, we select Prometheus as our backend third-party software. On top of Prometheus, we wish to create our own user-friendly front-end management web console suitable for Taiyi and Qiming and support customized user management. Additional supported functionalities to achieve include self-defined alert events and issue assignment and tracking.

II. Background

A. Third party monitoring systems

1) *Nagios:* Nagios is an event monitoring system that provides monitoring and alerting service for servers. It functions through user provided plugins to receive data to monitor, and therefore provides high flexibility. The Nagios scheduler helps to schedule the enquires to all server plugins. However, while the plugin based mechnism provides the ability to customize data to collect, due to the long development history of Nagios many available plugins turns outdated and no longer compats with the newer version of softwares.

2) *Prometheus:* Prometheus is an event monitoring platform which integrates an embedded time-series database, dashboard for visualization and alert system. It is a scrape-based system. Instead of targets pushing their metrics to the monitoring server, the monitoring server scrapes the metrics exposed through an endpoint from the targets. Through selectedd scraping, it reduces the communication bottlenect and improves monitoring efficiency. As a all-in-one platform, it reduces the problem of needing to keep being compatible with other collaborating softwares.

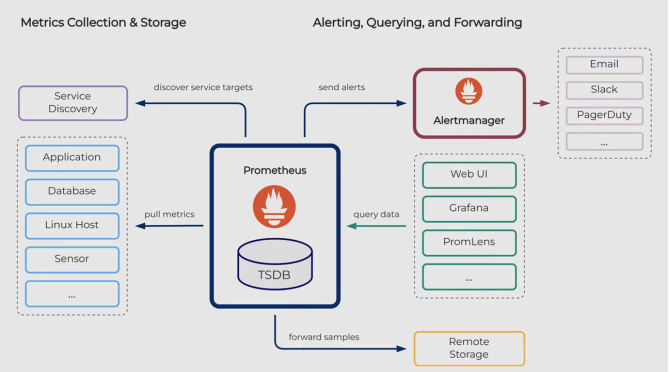

Figure 1: Prometheus architecture

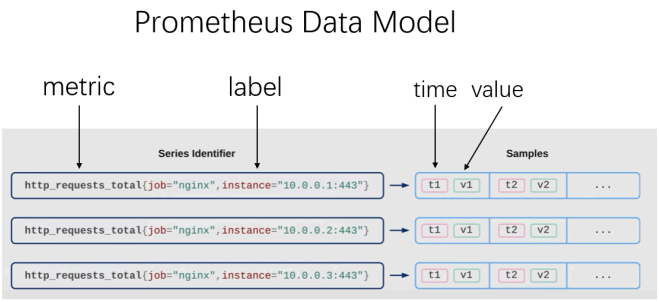

Figure 2: Prometheus data model

3) *Zabbix:*

Zabbix is a robust open-source monitoring solution designed to track and manage the performance and availability of IT infrastructure components such as servers, networks, and applications. It offers comprehensive monitoring features including real-time data collection, customizable alerting, and visualization through graphs and reports. Zabbix supports both polling and trapping. With its flexible architecture and scalability, Zabbix is widely adopted by businesses of all sizes to ensure the smooth operation of their systems and to proactively identify and address issues before they impact operations. However, Zabbix does not consider the peak period of jobs, which is continuous and periodic. The increase of data and machines will make the writing of the database encounter bottleneck. The maximum number of single machines given by the official website is 5000, so it is necessary to increase the proxy and increase the cost. Also, When the polling requests of machines increase significantly, the pull request will be blocked. This will cause delays in data collection. Deployment of Zabbix needs to install multiple Components. This is complex and prone to issues.

4) *Netdata:*

Netdata is a highly efficient, distributed, real-time monitoring and troubleshooting tool designed for modern IT infrastructures. It provides comprehensive insights into the performance and health of your systems and applications with unparalleled granularity and speed. As a new monitor system, it has multiple Feature. Netdata has real-time monitoring and comprehensive metrics. it has Operating system metrics, container metrics, virtual machines, hardware sensors, applications metrics, OpenMetrics exporters, StatsD, and logs. All data are collected per second and are on the dashboard immediately after data collection. Also, it has a good extensibility and efficiency, and useful out-of-the-box alerts.

B. Metrics to collect

1) *Baseboard management controller:* The baseboard management controller (BMC) monitors metrics of the computer hardware independent of the host systems' cpu, firmware and operating system. The metric it collects includes temperature of hardware components, fan speed, voltage and many others. The information from BMC is considered of great importance for abnormal readings for these values are likely to indicate hardware failure or danger state, and it provides information even when the host operating system is down.

2) *Operating system related metrics:* Operating system related metrics of interest includes CPU usage and memory usage. These operating system related metrics are closely related to a server's performance. Being aware of these operating system related metrics helps to diagnose the underlying fault for low performance or even non-functioning servers.

III. Frontend Design

A. Technology stack

The frontend stack comprises Vue along with VueRouter and Pinia for state management, utilizing the Vite build tool. Axios handles HTTP requests, while ES6 standards are adhered to for JavaScript development. Element-plus is employed for UI components and styling.

B. Project structure

The project structure is shown in the Figure 1.

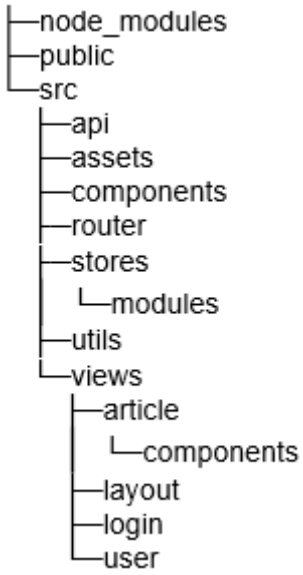

Figure 3: frontend project structure

C. Frontend Design

The frontend functionality modules are divided into three parts: user login, dashboard, and user information modification. The dashboard is further divided into four parts: Warning Management, Warning Detail, Cluster Observer, User Warning Design. Details of every machine are shown in Cluster Observer part. User can update their own warning message in User Warning Update module, which is inside

the Warning Detail module. The user information modification section is further divided into two part: user password modification and user personal information modification.

The structure is shown in the Figure 2.

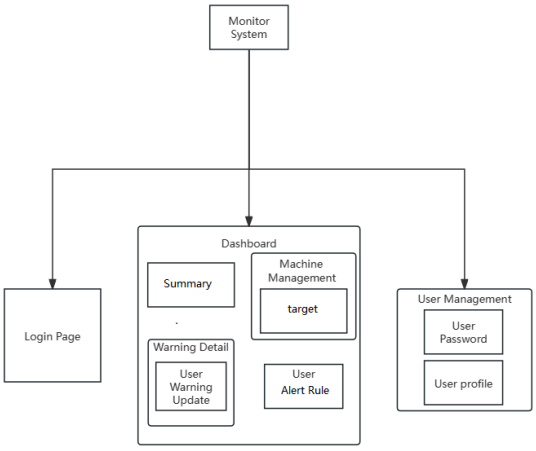

Figure 4: frontend page design

IV. Backend Design

A. Backend project structure

```
├─hpc-monitor-alert-->for Prometheus Alertmanager
│  └─src/main
│          ├─java
│          │  └─...
│          │
│          └─resources
│              └─application.yml-->configuration
│
├─hpc-monitor-db --> library for DB access
│  └─src/main/java
│              ├─dao-->Database Access Object
│              └─po -->Persistent Object
│
└─hpc-monitor-web --> HTTP API for frontend
    └─src/main
            ├─java
            │   ├─annotation --> Java Annotation
            │   ├─config --> SpringSecurity Config
            │   ├─controller --> HTTP API
            │   ├─filter --> JWT auth filter
            │   ├─handler --> Exception handling
            │   ├─model
            │   │  ├─param --> HTTP API parameter
            │   │  ├─dto --> Data Transform Object
            │   │  └─vo --> View Object for frontend
            │   ├─service --> service logical layer
            │   └─util --> utilities
            │
            └─resources
                └─application.yml-->configuration
```

B. Technology stack

- For running environment, we choose JDK21 because JDK17 will receive updates under the Oracle NFTC[2](No-Fee Terms and Conditions), until September 2024[3]. By using **GraalVM**[4] technology, we can also compile the Java source code to native image, which can save lots of memory.
- Generally, we use **SpringBoot**[5] as our backend overall framework because of its convenient autowire feature, annotation based extension and thriving ecology. With SpringBoot and **Maven** dependency management tool, we can integrate other module easily.
- To collect the real-time BMC information, we use `ipmi_exporter`[6] in the monitor machine because the BMC can also be connected after the monitored machine breaks. As for real-time OS loading information, we use `node_exporter`[7] on each monitored machine to collect. The `node_exporter` opens an HTTP endpoint for Prometheus to fetch the data. To make the endpoint more security, we can configure basic authentication for `node_exporter`.

```
# HELP ipmi_bmc_info Constant metric with value '1' providing details about the BMC.
# TYPE ipmi_bmc_info gauge
ipmi_bmc_info{firmware_revision="4.29",manufacturer_id="Inspur(Beijing) Electronic Information Indust

# HELP ipmi_chassis_cooling_fault_state Current Cooling/fan fault state (1=false, 0=true).
# TYPE ipmi_chassis_cooling_fault_state gauge
ipmi_chassis_cooling_fault_state 1

# HELP ipmi_chassis_drive_fault_state Current drive fault state (1=false, 0=true).
# TYPE ipmi_chassis_drive_fault_state gauge
ipmi_chassis_drive_fault_state 1

# HELP ipmi_chassis_power_state Current power state (1=on, 0=off).
# TYPE ipmi_chassis_power_state gauge
ipmi_chassis_power_state 1

# HELP ipmi_fan_speed_rpm Fan speed in rotations per minute.
# TYPE ipmi_fan_speed_rpm gauge
ipmi_fan_speed_rpm{id="28",name="FAN_SYS_1"} 2304
ipmi_fan_speed_rpm{id="29",name="FAN_SYS_2"} 2304
ipmi_fan_speed_rpm{id="30",name="FAN_SYS_3"} 3264

# HELP ipmi_fan_speed_state Reported state of a fan speed sensor (0=nominal, 1=warning, 2=critical).
# TYPE ipmi_fan_speed_state gauge
ipmi_fan_speed_state{id="28",name="FAN_SYS_1"} 0
ipmi_fan_speed_state{id="29",name="FAN_SYS_2"} 0
ipmi_fan_speed_state{id="30",name="FAN_SYS_3"} 0
```

Figure 5: BMC information fetched by `ipmi_exporter`

- To persist the time series data, we use **Prometheus** as our TSDB(Time Series Database), which has good expandability and high performance. It also supports API authentication, which increases the security level. Prometheus also implements a query language called PromQL to support slicing and dicing of collected time series data in order to generate ad-hoc graphs, tables, and alerts. We can also use **Prometheus Alertmanager**[8] to generate the alert event by configuring the PromQL trigger.

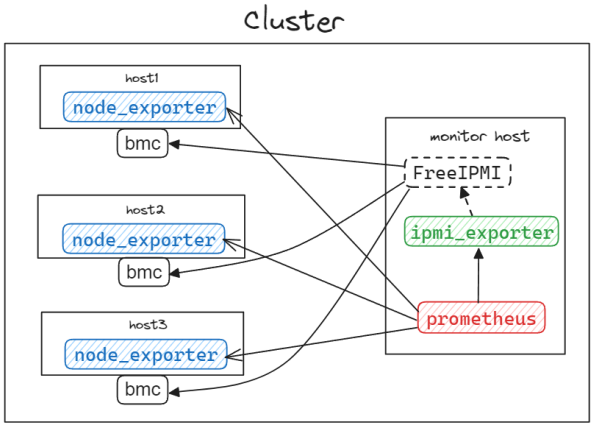

Figure 6: monitor structure

- For data persistence, we use **MySQL** as our relational database because it has many features like open-source, mature, simple configuration, good stability and excellent performance.
- For persistent layer, we use **Mybatis** as ORM framework. To simplify code, we also use Mybatis-Plus extension. Therefore, we don't need to write tedious `.xml` files. However, Mybatis-Plus can't support tables join query, so we use `mybatis-plus-join` dependency additionally.

```
<dependency>
  <groupId>com.github.yulichang</groupId>
  <artifactId>mybatis-plus-join</artifactId>
  <version>1.2.4</version>
</dependency>
```

- To ensure security, we use **SpringSecurity** as our security framework for authentication and authorization, which can implement the API's access control based on permissions. It also provides multiple authentication methods by using filter chains before entering API. For token generation, we use **JWT**(Json Web Token)[9] to identify the client, which is stored in user's browser and checked by server's private key.

C. Deploy

Generally to say, we are going to deploy the frontend and backend seperately, which has many advantages:

1. If only the frontend file changes, the CI/CD can only redeploy the frontend. Therefore, the backend service will not be affected at all.
2. Due to the frontend's static files(html,js,css,images) are deployed seperately, if the client only requests the static web pages, the backend service needn't to resolve the request, this can help to reduce the backend's load and increase the system's stability.

In application deployment, **Docker** enables us to make environment isolation and dynamic deployment. Beside the convenience, Docker's virtual environment in containers is lighter, which can help us to save more server resources. Since we need to deploy several containers, we choose to use **Docker Compose**'s `.yml` file to configure the containers more conveniently, so we deployed our platform using Docker Compose in five containers:

1. **Nginx**: For frontend deployment, we use Nginx web server, which provides many utility functions, like reverse proxy and load banlance. Besides, Nginx also has excellent performance in concurrency. We can config HTTP proxy as follows:

```
http {
    include mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;

    server {
        listen       80;

        location / {
            root   /etc/nginx/html;
            index  index.html index.htm;
        }
    }
}
```

2. **Java21**: For backend deployment, since the backend is using SpringBoot, so we deployed the container in JDK21 environment, this container provides the necessary APIs for the frontend.
3. **IPMI exporter**: This container can provide the monitor metrics for Prometheus, because it can get the target machine's health data through IPMI interface, then Prometheus can store the metrics with a timestamp.
4. **Prometheus**: Since the backend needs to get the monitor data, which is stored in Prometheus, the Prometheus container should persist the fetched data using the volume provided by Docker. So the container can stop or restart without losing the data.
5. **MySQL**: This container stores the relational data, because the time sequence data is stored in Prometheus. For data persistence purpose, this container's storage should also be mounted to the host machine through Docker volume.

Since the above five containers have dependency relations, the launching order of the containers should also follow the dependency relation, below is the topology graph. MySQL and Prometheus container have persistent data, so they have shadow in the figure. Five containers can access each other through the service name defined in the `docker-compose.yml`, because they are linked through Docker's virtual local network, in this application the network's name is "hpc-monitor".

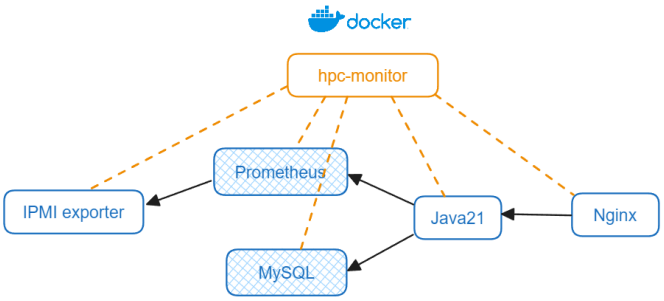

Figure 7: containers in Docker Compose

V. Frontend Implementation

A. Static Frontend Page

According to the frontend design before, Ten static pages have been created to fulfill our needs, including LoginPage.vue, LayoutCotainer.vue, WarningManage.vue, WarningDetail.vue, MachineManage.vue, ClusterObservers.vue, UserOwnWarning.vue, UserUpdateWarning.vue, UserProfile.vue, UserAvatar.vue and UserPassword.vue.

B. Frontend Storage

User store has been created by Pinia to store the related data about user, in order to meet the other needs in next stage of project development.

C. Frontend Requests

Axios has been wrapped request.js and has begun integrating with the existing backend interfaces. Currently, all the developed interfaces have been debugged, are operational, and are able to receive data from the backend.

D. API Management

Detailed API documentation can help the interaction between frontend and backend. We use **Apifox**[10] to manage HTTP API, which is a professional API management platform, the API documentation is deployed at https://apifox.com/apidoc/shared-d6a2f4cb-8bd7-4fd1-a914-0c2692b70406

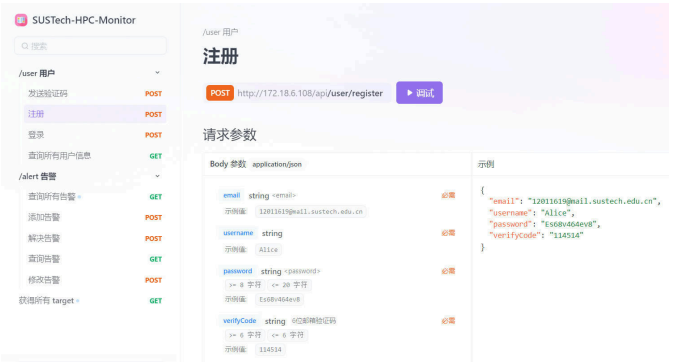

Figure 8: ApiFox online documentation

E. Page Implementataion

As figure 4 showed, we have implement our pages, we will explain the usage of these pages.

1) *Login Page:*

The users will first login in this page, getting authorization token from backend. All API only can be used after the frontend get the token. Also, if a user doesn't get authorization, the router guard will stop the user from accessing other pages. If a user haven't obtained an account, this page also allow user to sign up with the e-mail account. In order to prevent CSRF and other possible internet attacks, our page support verification code for safety.

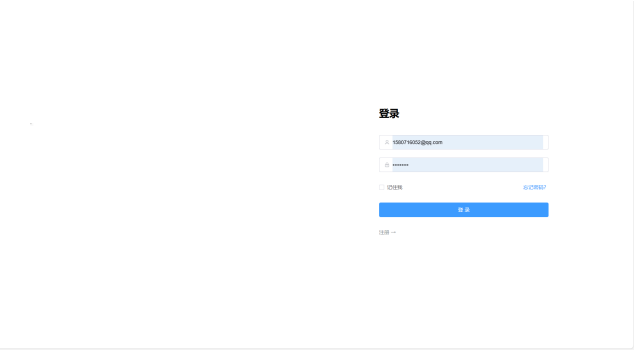

Figure 9: Login Page design

2) *Summary Page:*

This page shows the the summary of the warning, including the solve status and severity. Below the summary shows all the warning.

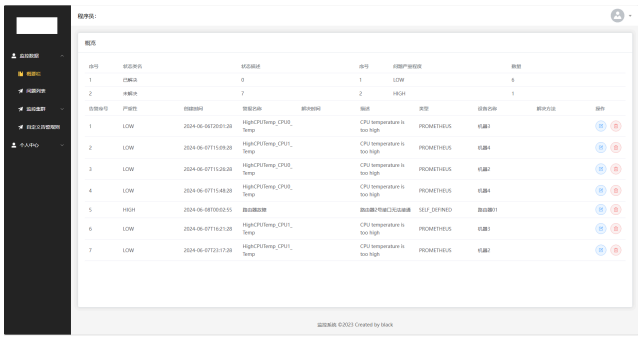

Figure 10: Summary Page design

3) *Warning Detail Page:*

This page shows all the warning. It support search warning by warning name, warning status, and start time. Users can add, solve, edit warning in this page.

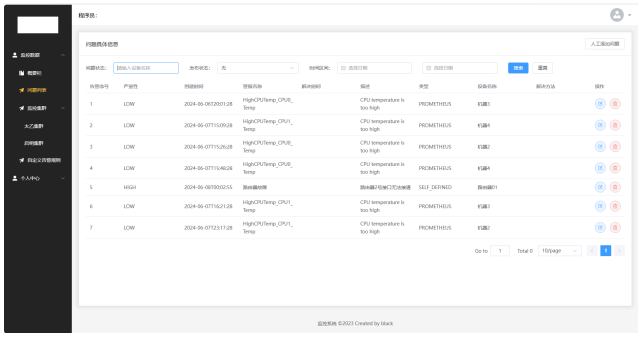

Figure 11: Warning Detail Page design

4) *Machine Manage Page:*

This page shows all the machine in two cluster, Taiyi and Qiming. The machines which has warning will have red background. Users can add machine into cluster in this page by form or excel file, and multiple principals can be set up for each machine in this page. Users can also edit and delete machine in the cluster.

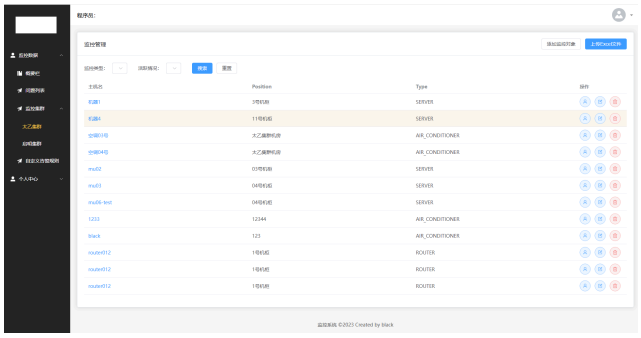

Figure 12: Machine Manage Page design

5) *Target Manage Page:*

This page shows all the target in the machine. Users can add, edit and delete target in the machine.

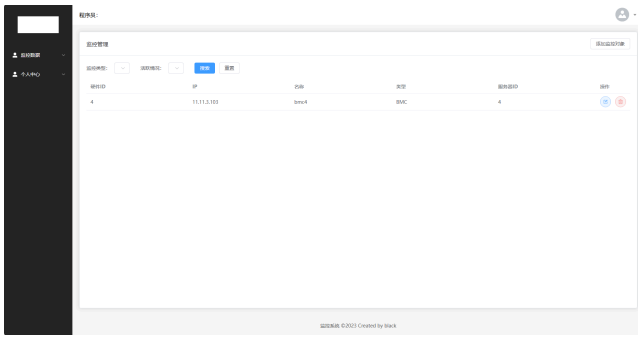

Figure 13: Target Page design

6) *Alert Rule Page:*

This page shows all the alert rule in the machine. Users can add, edit and delete alert rule.

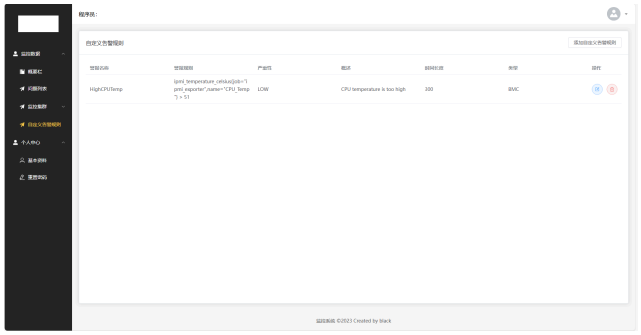

Figure 14: Alert Page design

7) *User Profile Page:*

This page can change the profile of the user, including name and e-mail.

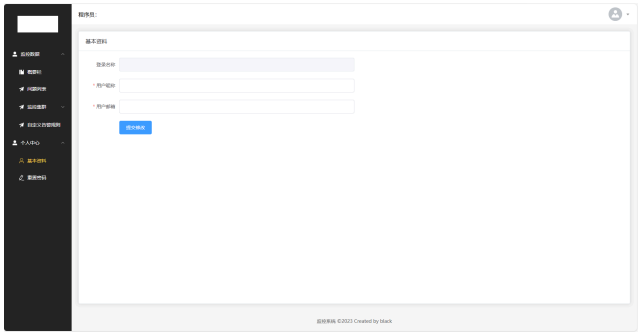

Figure 15: User Profile Page design

8) *User Password Page:*

This Page can change the password of the user.

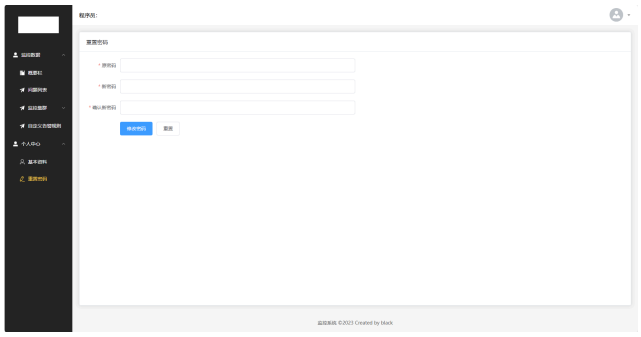

Figure 16: User Password Page design

VI. Backend Implementation

A. Database Management

We use database for supplement information management. Prometheus groups information by the monitoring exporter. It does not provide the advanced functionalities such as grouping hosts across multiple exporters, or maintaining cluster level information. To support the management of SUSTech HPC center which involves the hierarchy of two clusters Taiyi and Qiming, also to support customization of hardware and software monitoring, we designed the following database structure. We also implement our own user management system.

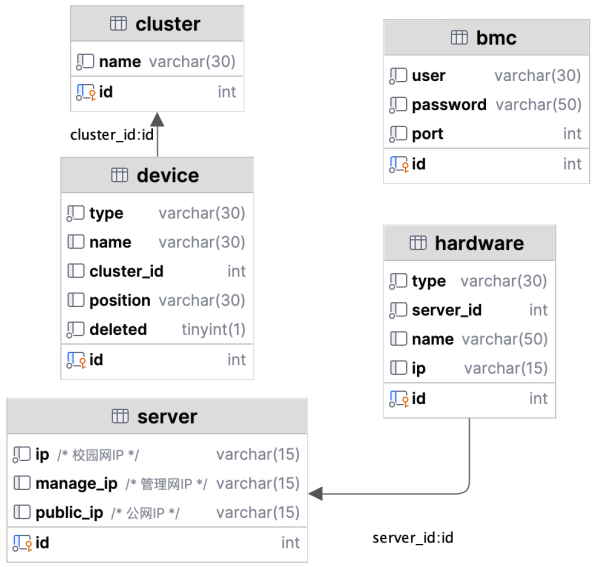

Figure 17: Target management database design

For target management we include three main tables, cluster, device and hardware shown in Figure 17. Each device belongs to a cluster, and devices could be of multiple types such as server, air-conditioner and router. Each hardware links to a server and is a target for Prometheus monitoring. For each type of hardware, since the information we need is hardware specific and cannot be known beforehand, we dynamically create more tables for additional supported hardware types. The example bmc table shows the information we collect for managing bmc boards, which includes the IP address, username and password for authentication.

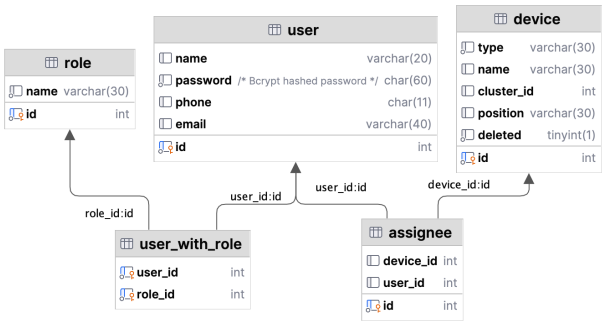

Figure 18: User management database design

In the user table we manage necessary information for the user management of the monitor system. This includes the username, password, phone number and email of the registered users.

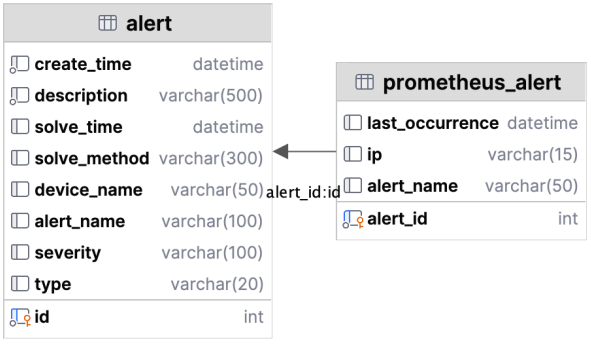

Figure 19: Alert management database design

The alert table and the Prometheus alert table is used for alert management. The alert table records basic information of both solved and un-solved, manually added and auto-detected alerts. For Prometheus alerts, additional information such as the IP address of alerting hardware, last detected firing time is recorded in the Prometheus alert table.

In the assignee table, we link users to be responsible for devices. These assignees would be informed when alerts fire on their responsible devices.

B. User Management

1) *Register:*

1. The user first enters an email for receiving the verification code.
2. The backend then generates 6 bits verification code and stores key-value pair as `email-code` in a Hashmap, since the map is modified by multiple

threads, we use JDK's native implementation class `ConcurrentHashMap`.

3. If the user sends the register request within 2 minutes, the register will success. Otherwise, the verification code will expired because the server will remove the code from the hashmap cache after 2 minutes.
4. After the registration, the server will use secret key to generate JWT(Json Web Token)[9] and send it to the client for later requests.

2) *Authentication:* The SpringSecurity[11] framework can integrate into SpringBoot[5] application. We can insert our custom `JwtAuthenticationFilter` to SpringSecurity's native filter chains to implement the JWT Authentication. The `JwtAuthenticationFilter` first uses server's secret key to parse the JWT from the client request, then stores the user id into SpringSecurity's context, which provides useful information for application's logic layer.

C. Device Management

The system supports management of devices of multiple types, such as server, air-conditioner, router, switch, and so on. The information of devices are stored in the device table of the database. When users add a device, basic information includes an unique device name, the cluster that the device belongs to and the position of the device. Additional type specific information such as public IP of servers could be supplied through a JSON format parameter.

The monitoring system supports batch addition of devices aside from single device addition. This is achieved through providing an excel file which includes all necessary information of the devices. The system parses the excel file and adds the devices described in the excel file into the system.

D. Alert Management

1) *Alert Rules:* Prometheus allows users to set up customized alert rules through `.yml` files and the PromQL language.

An alert rule includes multiple configurations. It includes the expression to identify an abnormality, how long should the abnormality be observed for the alert to fire and the critical level of the alert.

For each exporter that we monitor, we create a `.yml` file for controlling its alert rules. The front-end provides the necessary configurations for the alert rule, and our back-end is responsible for rewriting the user-provided configurations into the yaml file format that Prometheus requires.

There are multiple configurations that are both hardware specific and metric specific. We provide the possible metrics and configuration options through lists after user specified the hardware type and metric to observe for alert firing sequentially. A challenge in generalizing the configuration options lies in the self defined label names depending on specific exporter implementation. We adopt manual observation by observing the /metric endpoint of specific exporters. The metric endpoint is exposed through a Prometheus defined format, therefore we first use a tool Prom2json[12] to translate it into json format for easy parsing. By studying the pattern of each exporter, we then conclude how to enable the translation between simple front-end specification and the back-end yaml file configuration rules. Taking the BMC hardware and the ipmi exporter as an example, after specifying the metric name, specification of the name label of the metric such as 'CPU0_Temp' or 'PCIE_Temp' is essential for accurate description of the alert rule. At the same time, we wish to be able to specify the same rule for all CPU temperature regardless it is 'CPU0' or 'CPU1'. Based on these observations, we save a /metric endpoint sample data in its json format to be analyzed by our system to get all the name labels for each metric. We then extract all the numbers from the names and then enable front-end selection. After the name is selected, we again translate it into alert rule language understood by Prometheus through regex matching of letting the part which is previously concrete numbers to match any numbers. In this way, we generalized and also simplified the alert rule specification through hiding the translation details from front-end through manual observation. It is extendable to other hardwares through carefully extracting the core functions to hardware-specific implementations.

Prometheus needs to be reloaded after updating rule files and configuration files. We reload Prometheus through an http POST request to the api Prometheus provides in its web service.

2) *Alert Retrieval:*

Prometheus Alerts: We retrieve alerts' information through the HTTP API Prometheus provides. Through parsing the JSON response from Prometheus, we could access the information of currently firing alerts for front-end display. The information includes the alert name, the host triggering the alert, and the time-stamp when the alert was triggered. However, we wish to inform users of all alerts which had fired and also provide users the functionality of solving the alerts while recording the solving method.

To achieve this, we retrieve firing alerts from Prometheus at a certain interval. The newly discovered alerts we would save them into the backend database for storage. The way we distinguish new alerts are through identifying the name, filtering conditions and the device of the alert. It the tuple consisting of these identified factors are un-recorded in the unsolved alerts recorded in the database, we identify it as a newly discovered alert to be saved.

Manual Alert Creation: Some alerts can't be auto-detected by Prometheus' exporters and are manually added to the system. we also implement the corresponding API.

3) *Alert Notification:* The system supports assigning users to be responsible for devices, and be notified through email if any alerts fires on these devices. We use the database to save the binding relationship of users and devices. Each time a new alert is detected, we look up users responsible for the device the alert is fired upon and send a notification email to the user. The email contains basic information of the alert, and the user could login in the web page of the system for more details.

E. BMC Monitoring

1) *IPMI Exporter:* We use the open source IPMI Exporter[6] of Prometheus, which supports collecting the IPMI metrics of over a thousand BMC boards and exposing them on a /metric endpoint on the server. We deployed the ipmi exporter on our monitor server.

2) *Target Configuration:* The ipmi exporter uses an yaml file on the Prometheus server side for target configuration. It is worth noting that though the target IPs are configured on the server side, the additional information such as the username and password for ipmi authentication and communication protocol between the exporter and the BMC board are to be configured on the exporter side. This part of our backend system takes over all the tedious details that needs to be taken care of through configuring the yaml files, while providing a simple interface for users. We support both adding and deleting targets. Prometheus supports hot-loading of the target file, therefore we could simply re-write the yaml files without the need to restart Prometheus. Through working together with the database management, we support connecting the bmc information to specific hosts and also locating them to their belonging clusters.

3) *Target Retrieval:* Prometheus provide a target API which provides all current target information. We retrieve the information from this target API and extract the core information such as host IP and monitor jobs for front-end display.

References

1. Center for Computational Science and Engineering of SUSTech. https://hpc.sustech.edu.cn/
2. Oracle No-Fee Terms and Conditions. https://www.oracle.com/downloads/licenses/no-fee-license.html
3. JDK17. https://www.oracle.com/java/technologies/downloads/#java17
4. GraalVM. https://graalvm.org/
5. SpringBoot. https://spring.io/projects/spring-boot
6. https://github.com/prometheus-community/ipmi_exporter
7. Prometheus Node Exporter. https://github.com/prometheus/node_exporter
8. Prometheus AlertManager. https://prometheus.io/docs/alerting/latest/alertmanager
9. Json Web Token. https://jwt.io/
10. ApiFox: Postman + Swagger + Mock + JMeter. https://apifox.com/
11. SpringSecurity. https://spring.io/projects/spring-security
12. Prom2Json Github Repository. https://github.com/prometheus/prom2json