

JOINT INSTITUTE FOR NUCLEAR RESEARCH

[Dzhelepov](https://web.archive.org/web/20031204011713/http:/dzhelepov.jinr.ru/) laboratory of Nuclear Problems

**FINAL REPORT ON THE**

**START PROGRAMME**

*SPD Hardware Database*

**Supervisor:**

Fedor Valerievich Prokoshin

**Student:**

Boris Dvizov, Russia  
North Ossetian State University

**Participation period:**

July 22 – September 14,

Summer Session 2024

Dubna, 2024

TABLE OF CONTENTS

Abstract 3

Introduction 4

History of development 4

SPD information ecosystem 5

Languages and frameworks 8

Rest API service 9

AUTH service 13

Testing 15

Containerization 15

Automatic deployment 16

Conclusion 18

Bibliography 19

Acknowledgments 20

АBSTRACT

Spin Physics Detector (SPD) is a experiment at the NICA accelerator complex at JINR, designed for studies in spin physics with a polarized beams. It will be composed of number of subsystem, using various types of detectors: drift chambers and tubes, MRPCs, Cherenkov counters, scintillation and silicon detectors. Signals from the detectors will be collected by the data acquisition system having few hundred thousand channels. Every component of the detector and DAQ systems will have its sets of parameters and configuration settings that have to be kept for use in operating and maintenance of these systems and especially helpful in knowledge transfer between team members. Some of this data must be also used for the processing and analysis of data collected at the the facility.

An information system is being developed to store parameters of the equipment components, and provide means to access it both for personnel and automatic systems. For each type of device, a set of parameters are defined that are common to all devices of this type. The values of these parameters can be specified for each device having unique Hardware ID, or default values of some parameters can be taken. Having in mind large number of equipment, a means for filling data for a groups of devices are being implemented. A web interface is being developed for creating records for a single device or a group of the devices and retrieving information based on several criteria.

INTRODUCTION

The SPD experiment is a research project aimed at studying the physics of elementary particles and searching for new fundamental knowledge about the interactions and structure of particles. The main goal of this project is to analyze the data obtained from the experimental SPD detector installed at the NICA particle accelerator, created at the Joint Institute for Nuclear Research, located in the city of Dubna, Moscow region.

The SPD installation is a large-scale system of detectors capable of registering and analyzing various events that occur during collisions of polarized deutron and protons with high energy.

To obtain, store and process a large amount of data obtained during the SPD experiment and to ensure successful operation of the experimental setup, it will be necessary to create complex information systems (IS). One of them should be Hardware Database - a service for storing and providing information about parameters of equipment components and their connections with other components.

HISTORY OF DEVELOPMENT

When developing the IS for the SPD experiment, it is planned to make extensive use of the experience gained in creating similar systems for existing experiments. A similar information system for storing parameters of hardware components was created for the CMS detector at the LHC accelerator. Hardware Database inherits the idea of this system, in particular the assignment of a unique identifier (HWID) to each component and the sorting of components into groups, but it is built on a different technical stack.

The first version of Hardware Database was written on FastAPI framework, and the structure of the database included 2 tables: groups and components. Parameters of groups and devices were stored in tables in JSON format, which caused problems with fast data validation and complexity of data processing. To improve the quality of work with the database and the speed of processing, a new architecture was developed (more details are described in REST API SERVICE).

To improve the quality and speed of the service, it was decided to switch from Python to Golang, thanks to which it was possible to introduce new features into the service and increase the speed of query processing.

SPD INFORMATION ECOSYSTEM

In addition to Hardware Database, various databases and information systems will be used for the SPD experiment:

• Database of event-level metadata (Event Index);

• Databases of data-taking conditions and calibrations;

• Distributed computing and data storage management systems;

• Database of physical metadata;

• Monitoring systems;

• Logging and accounting systems;

At this stage of the SPD experiment, the information and computing systems are in the initial stage of development. Due to the lack of real data, the Hardware Database is being developed on the basis of test generated components.

The Hardware Database SPD is being developed as an integrated information system that should provide :

• Obtaining information about the parameters of hardware components and their relationship with other components;

• Transmission of this information and recording in databases;

• Access to information for data processing and analysis programmes through APIs and applications;

• Access to information for users through interactive and asynchronous interfaces.

• Authorisation in the system by means of JINR SSO and granting access rights;

Figure 1 shows a preliminary diagram of the Hardware Database architecture.

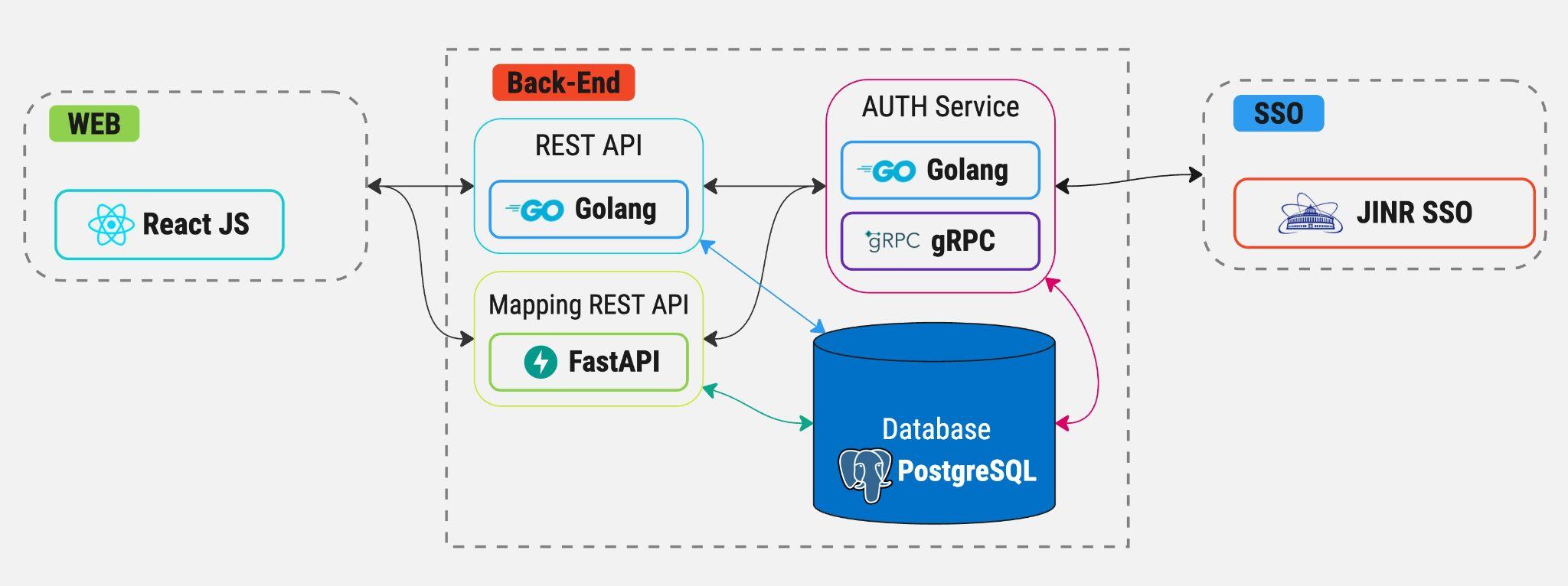


Figure 1: SPD Hardware Database architecture and schema

The choice of a platform for data storage and management was made based on the expected flows and volumes of information, the content of the Hardware Database record and the expected use cases.

Recording of equipment components into the database is realized by the following sequence of actions:

• The first step is to create a description for a group of components of the same type, which will include names of the component parameters, and their data types (integer, float, etc...) Optionally there can be specified default values of these parameters and allowed ranges of their values. Figure 2 shows a visual representation of the group

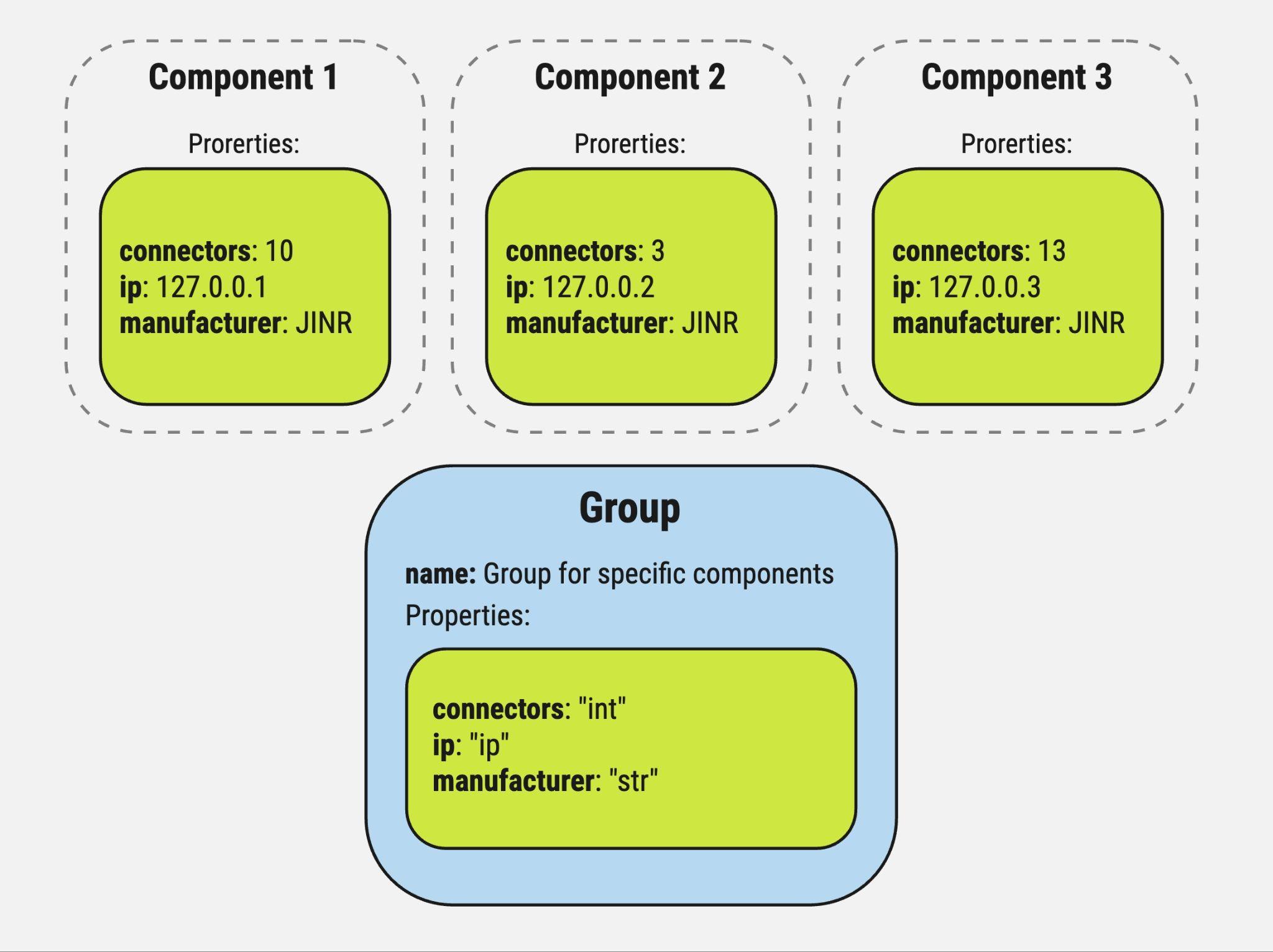


Figure 2: Visual representation of the group

• After creating a group, you can create components based on this group, and fill the values of the parameters. If allowed range of values has been specified for the group, user can enter values only within it. If default values were specified in the group description, they will be used for all components of this group if no value will be filled for specific component.

The PostgreSQL database was chosen for reliable storage and processing of structured data. It is widely used for information management in various applications, from small websites to large corporate systems. PostgreSQL supports SQL standards and provides a rich set of functions for working with data, including support for complex queries, transactions, indexes, views and stored procedures. One of the key features of PostgreSQL is its ability to process large amounts of data and support multithreading, which makes it an excellent choice for applications with high performance and reliability requirements.

The frontend part of the client interface was developed using the React JS framework. It provides tools for creating modern dynamic user interfaces, as well as provides effective interaction with the server and data manipulation. An React JS application is built from a set of components, each one representing a specific part of the user interface. Components can be nested into each other, forming a hierarchy. Each component includes an HTML template for displaying the user interface, TypeScript code for logic and data structure, as well as CSS styles for appearance.

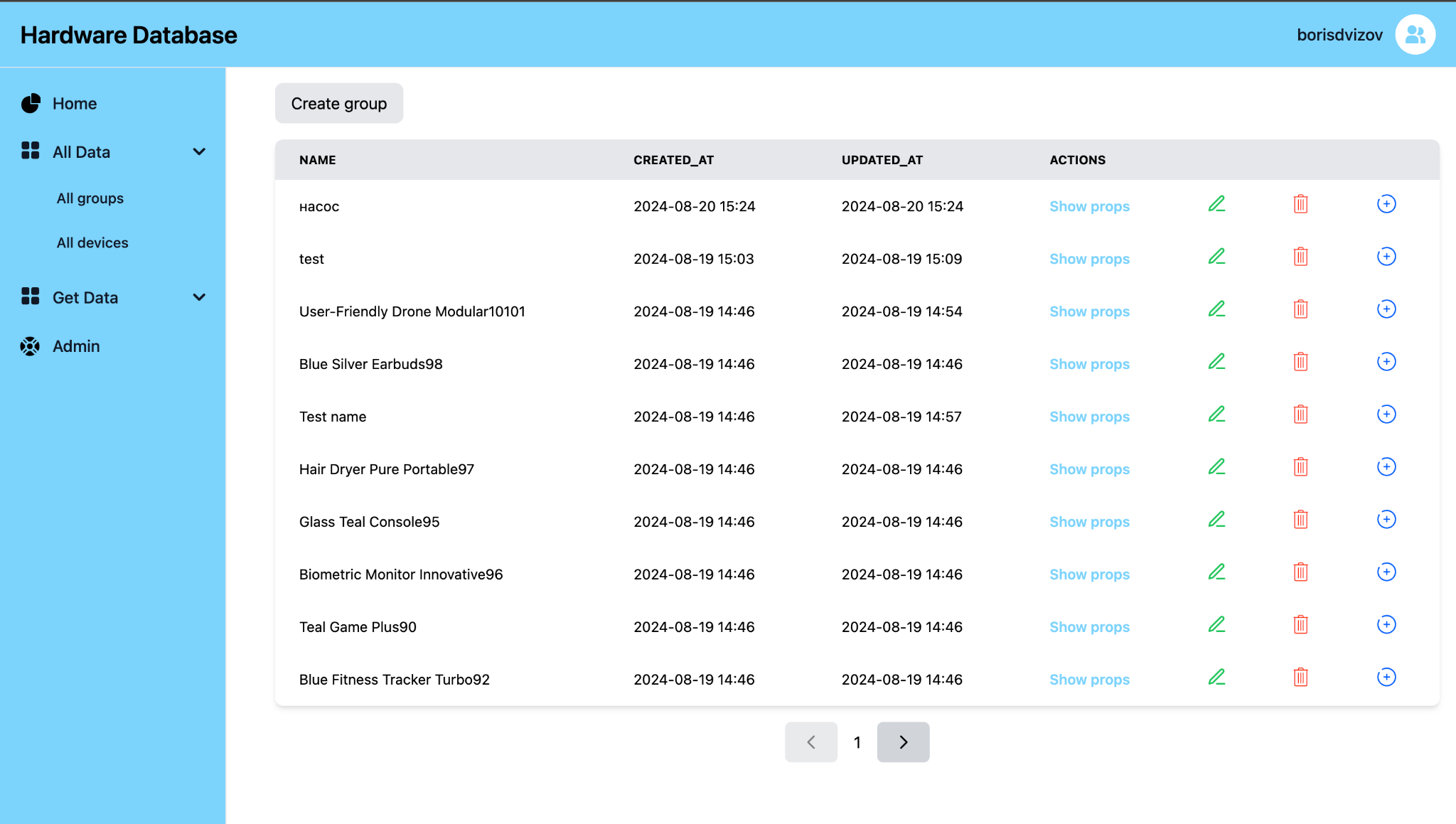


Fig. 3: Web interface

LANGUAGES AND FRAMEWORKS

The architecture of the project is built on the basis of microservices, which allows to divide the logic of different tasks into separate components that communicate with each other. This approach allows to increase the level of availability and fault-tolerance - in case of failure or breakdown of one microservice, the rest continue their work, as this approach allows to easily scale the system and increase the flexibility of development.

The Golang (Go) language was used to develop the REST API and AUTH microservices. Go is a programming language created at Google for developing software products. Go was developed to solve specific problems that programmers face when creating large software projects, including slow program builds, uncontrolled dependencies. The main advantages of the language are:

• Performance - code in Go can run 5-10 times faster than Python without special optimization thanks to its own macro assembler.

• Reliability - rational use of memory and computational resources ensures stability of programme operation.

• Goroutines and channels - support of parallel execution of functions and convenient data exchange between them.

• Automation - code auto-formatting, garbage collection, automatic documentation and built-in testing tools to improve development efficiency.

The FastAPI framework written in Python was used to develop the Mapping service. FastAPI is a lightweight asynchronous framework for Python, which is mainly used for developing API services. The main advantages of the framework are:

• Speed of operation. FastAPI beats all Python frameworks in terms of performance.

• Asynchrony. FastAPI uses ASGI servers by default, when in the same Django you have to deal with the configuration of the application from WSGI to ASGI, which takes enough time. Flask, unfortunately, does not support asynchrony and works only under WSGI

• Built-in data validation

REST API SERVICE

This service is designed to store and retrieve data on parameters of equipment components and to sort components by groups. The service operation is divided into several stages:

• Receiving HTTP requests via REST API and returning data requested by the user.

Each HTTP request is accompanied by middleware, which is responsible for receiving the JWT token from the request and sending it to the authorization microservice to provide information about the user and his access rights. If the user's permissions are not sufficient to access the requested resource, the request returns an error. If the request successfully passes the validation, it proceeds. Currently, we have implemented HTTP requests for working with groups, components, and user permissions. Figures 4, 5 and 6 show the actual HTTP request made.

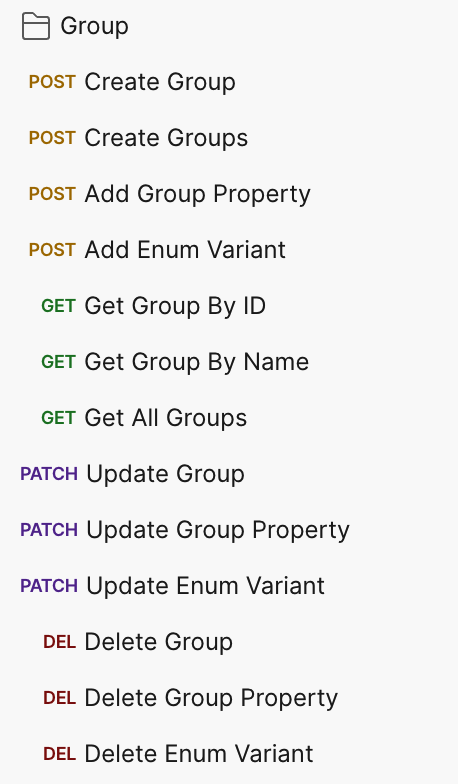


Fig. 4: Group HTTP requests

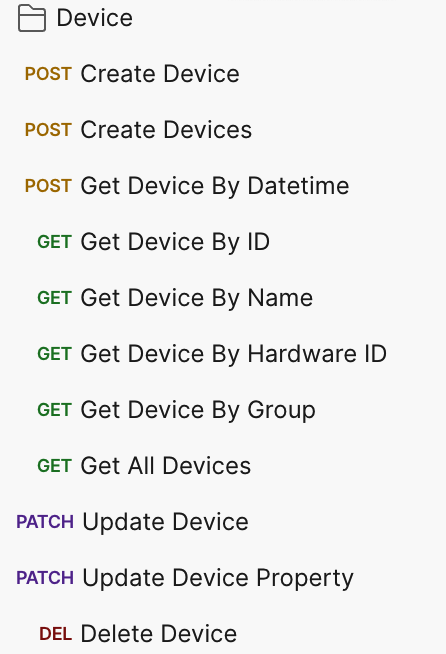


Fig. 5: Component HTTP requests

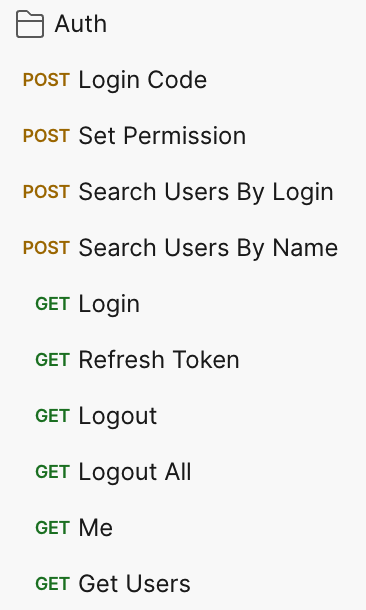


Fig. 6: AUTH HTTP requests

• Execution of business logic - This phase includes data validation, sorting, and conversion of data from database models to custom models (models used for business logic and user response models).

• Database Queries - Queries to add, retrieve, update and delete data from the database take place at this stage. The database structure consists of the following tables: a table with information about groups, a table with group parameters, a table with enumeration options for certain parameters, a table with information about components, 8 tables with component parameters divided by data type, and a table with migration version. Figure 7 shows a visual representation of the database structure.



Fig. 7: The architecture of the database

AUTH SERVICE

The authorization service is responsible for obtaining information about the user and his rights to use the service. Interaction with the REST API service is performed using the gRPC protocol.

gRPC uses HTTP/2 as its transport protocol. HTTP/2 provides a more efficient use of network resources compared to HTTP/1.1 by allowing multiple requests and responses to be transmitted in parallel within a single TCP connection. This reduces latency and improves overall performance. ProtoBuf allows data structures and service interfaces to be defined in special “proto files”. These files are compact, efficient, and allow you to automatically generate source code for various programming languages.

JINR SSO is accessed via HTTP request, after which the database records information about the user, his rights, unique session indexer, after which the REST API service returns 2 tokens: Access token - necessary for user indexing (lifetime 1 hour) and Refresh token - token for issuing a new pair of tokens (lifetime 7 days).

The database architecture of the AUTH service consists of three tables: a table that stores user information, a table that stores user sessions, and a table that stores the database migration version. Figure 8 shows a visual representation of the database architecture.

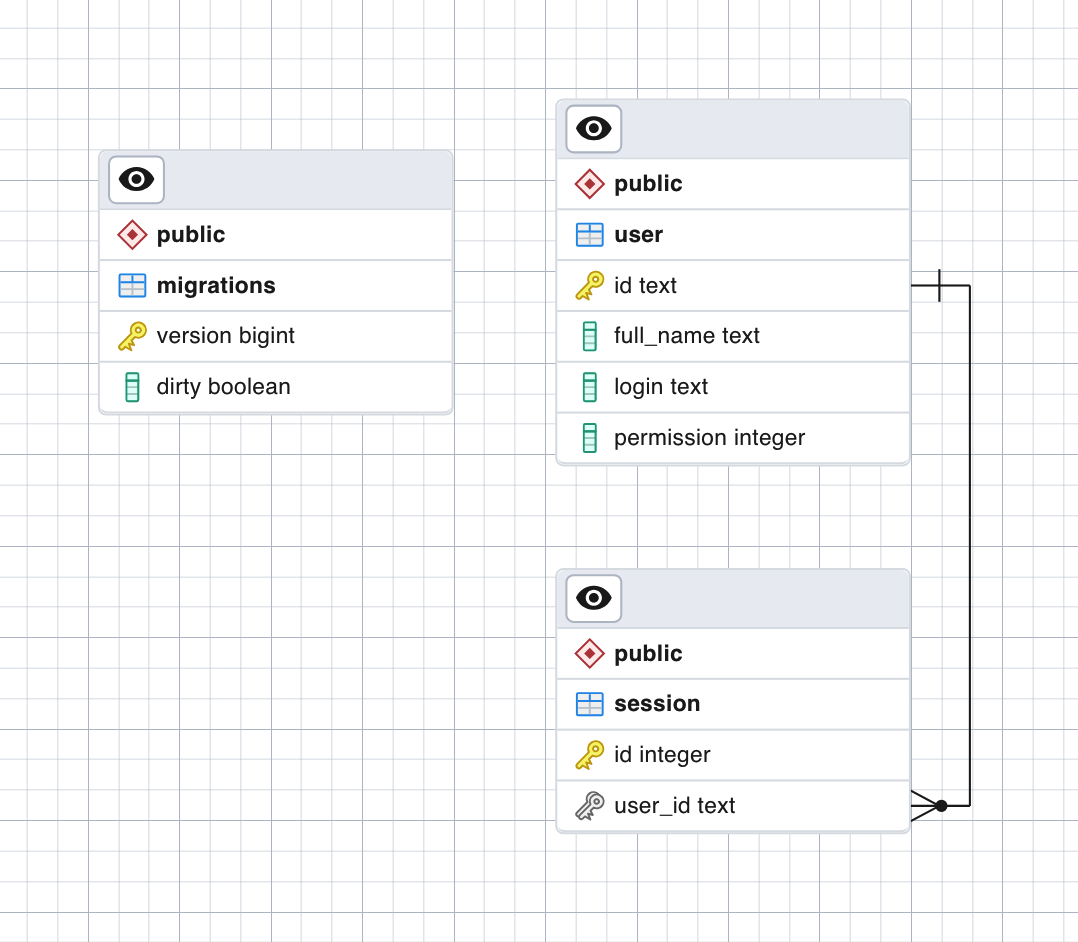


Fig. 8: The architecture of the AUTH service database

Currently, 10 RPC requests to the AUTH service are implemented. Figure 9 shows requests to the AUTH service.

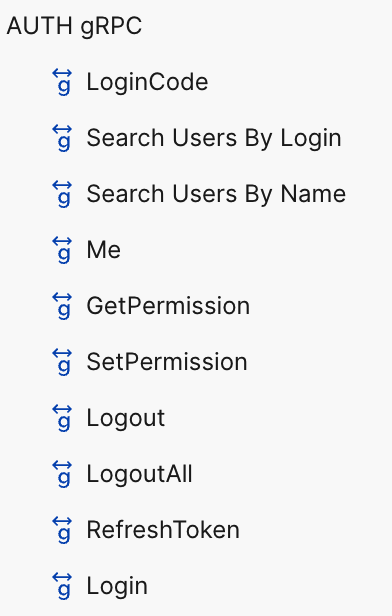


Fig. 9: RPC requests for AUTH service.

TESTING

Several levels of testing have been implemented for the REST API service:

• For database queries, unit tests have been implemented to check the correctness of the stored data and to handle possible erroneous events.

• Integration testing - tests that verify the relationship between business logic and database queries.

• Functional tests - they are responsible for checking the entire service for correct operation and error events. The service is launched separately from the tests, after which requests are executed using the REST API.

• E2E tests - these tests are designed to simulate the behavior of real users and create load on the service. They are performed similarly to functional tests - the service and the tests are started independently. E2E tests are built in the following way - in the test configuration file constant values are set for the number of created groups, components and the number of requests to receive them, then fake data is generated and distributed to several workers and sent to the service. When all groups and components are sent, requests to receive them are executed - at each request one of them is randomly selected from the list and sent to the service. When the sending of requests is finished, a report is generated, which contains all the fulfilled requests, the number of successful and failed requests, as well as the average time of fulfillment of requests. Figure 10 shows the report generated by creating 100 groups, 500 components, and running 1000 data retrieval queries.

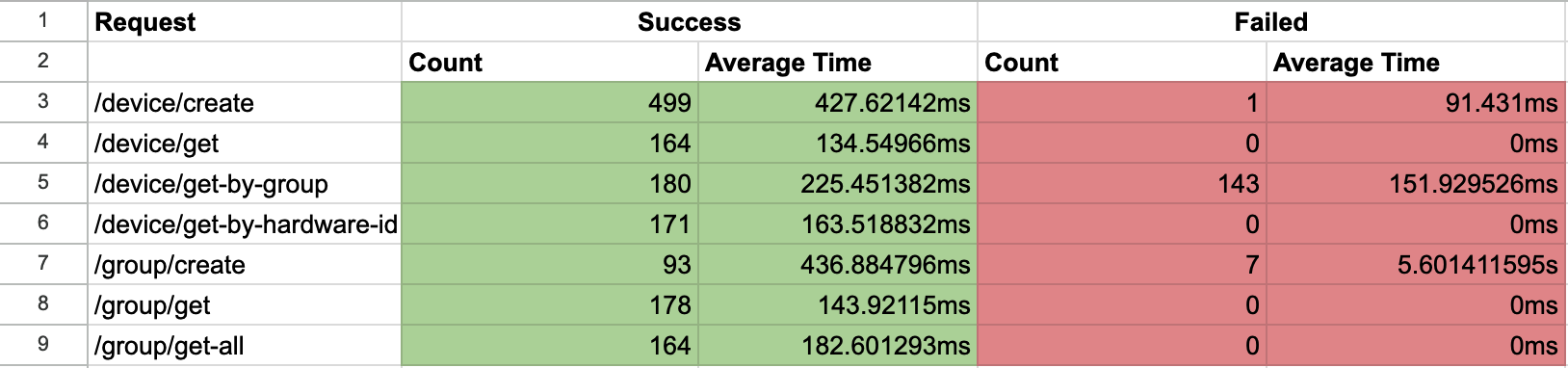


Fig. 10: E2E test report.

CONTAINERIZATION

Containerization is a technology that allows an application and all of its dependencies to be packaged into a single, isolated container. Containers provide a lightweight alternative to virtual machines, allowing multiple containers to run on a single host with minimal resource consumption.

Each component of a hardware database project is isolated and can be developed, deployed, and updated independently of the others. To provide this independence and simplify service management, containerization is used. It allows each service to be packaged in a separate container with its dependencies and environment, making it independent of the underlying operating system and other elements.

Advantages of using containerization:

• Isolation: Each container is isolated from other containers and from the host, preventing conflicts between applications.

• Portability: Containers can be easily moved between environments, whether development, test, or production.

• Scalability: Containers are easily scalable, simplifying load management in microservice applications.

• Rapid deployment: Out-of-the-box container images allow new application instances to be deployed in seconds.

Currently, the base images (Golang, PostgreSQL) for the Hardware Database project are taken from the public Docker Hub repository and used to create Docker images of services (Hardware Database Backend, Hardware Database AUTH), which are further stored in GitLab (container registry). This provides tighter version control and security by restricting access to images to authorized users. This topic will be covered in more detail later, including CI/CD processes and image management in a hardware database microservice architecture.

AUTOMATIC DEPLOYMENT

CI-CD (Continuous Integration/Continuous Deployment) is the practice of automating the stages of software development, from code writing to deployment.

Continuous integration (CI) involves frequent merges of code added by developers into the main repository. Each such merge is accompanied by automated testing and builds, allowing you to quickly identify and fix bugs.

Continuous deployment (CD) is the process of automatically deploying modified code to multiple environments after it has successfully passed all CI checks. This approach accelerates the development process and reduces the risks associated with deploying new versions of an application. After successfully passing all stages, service images are stored in the Container Registry, a private repository that provides centralized storage for containers.

This provides several important benefits:

• Security: Images are stored in a isolated repository with access restricted to authorized users. This protects sensitive data and prevents unauthorized access to containers.

• Version Control: The GitLab Container Registry allows for easy version control of images, providing easy access to all previous versions and the ability to roll back to stable versions when needed.

• CI/CD Integration: Because containers are stored in GitLab, they can be automatically used in CI/CD processes. This makes it easier and faster to deploy new versions of system components.

The process of building Docker images within the Hardware Database CI/CD project is automated using a yml file. This file defines all stages of the build and deployment process, such as:

• Build: This stage creates a Docker image from the project's source code.

• Test: The image is checked for bugs using automated tests.

• Deploy: After successfully passing all tests, the image is automatically deployed to one or more environments.

For example, hardware database project component images are built autonomously in the Container Registry, passing through all stages of testing before deployment to a virtual machine. This increases security and simplifies container version management, which is especially important in a microservice architecture project.

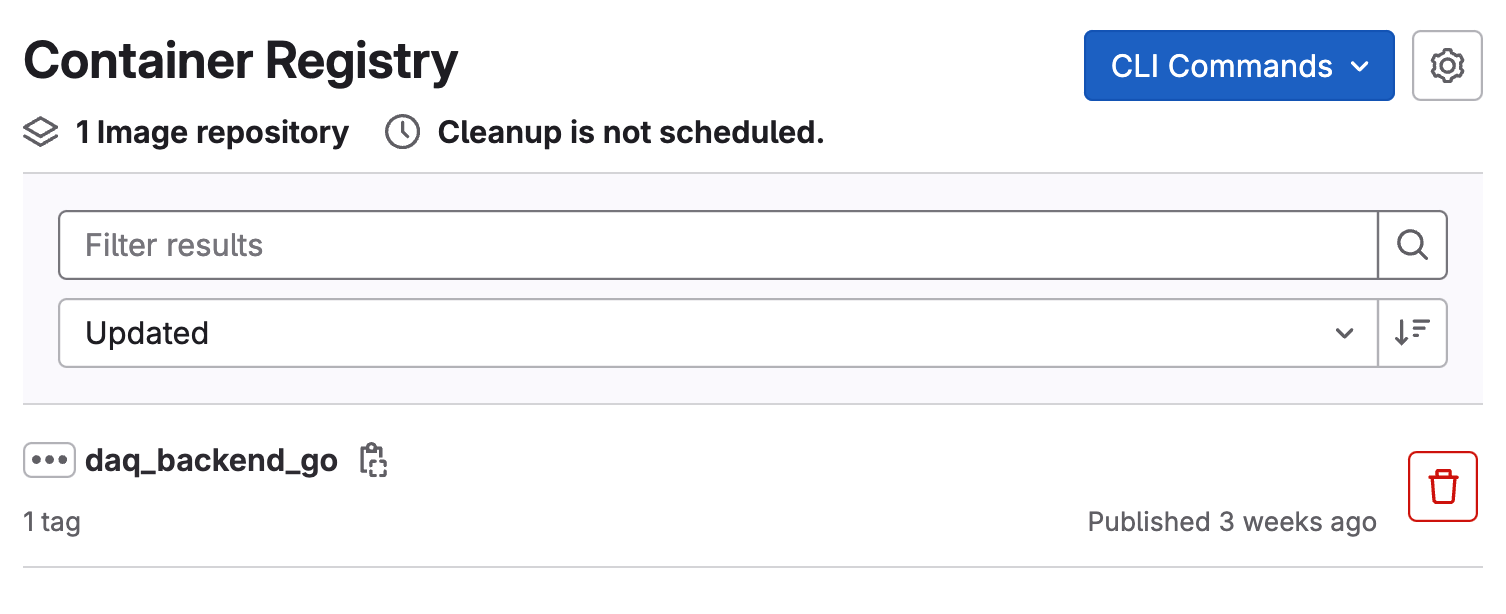


Fig. 11: Hardware Database project images in Container Registry.

CONCLUSION

In the course of further development of the Hardware Database project, the following tasks are expected to be performed:

1. Test the service with real data.
2. When the centralized PostgreSQL service is ready, deploy the production version of the service.
3. Evolve the service in response to user requests.

The implementation of these tasks will be carried out in cooperation with the development of other IS installations and depending on their readiness

BIBLIOGRAPHY

[Golang Documentation: Effective Go](https://go.dev/doc/effective_go)

[PostgreSQL Documentation 16.4](https://www.postgresql.org/docs/16/index.html)

[FastAPI Documentation 0.112.1](https://fastapi.tiangolo.com/)

[About gRPC](https://grpc.io/about/)

[Information systems for data taking and data processing, SPD collaboration meeting, 2023](https://indico.jinr.ru/event/3575/contributions/20669/attachments/15249/25742/2023-04-26.Information%20systems%20for%20data%20taking%20and%20data%20processing.pdf)

[TechnicalDesignReport\_SPD2023](http://spd.jinr.ru/wp-content/uploads/2023/03/TechnicalDesignReport_SPD2023.pdf)

ACKNOWLEDGMENTS

The author expresses gratitude to Fedor Prokoshin and Inga Tvauri for providing comprehensive guidance and moral support in the process of work, as well as for technical support and valuable recommendations. I am also grateful to the organizers of START for the opportunity to complete the program.