Building and Operating a Microservices Architecture for an Online Judgment System

NICUȘOR DUMITRU SAMI, BĂRBUȚ-DICĂ
UNIVERSITATEA DIN CRAIOVA
Facultatea de Științe
Specializarea: Informatică
Email: sbarbutdica@gmail.com

Abstract

This paper presents a comprehensive examination of the development life cycle of Pantheonix, an advanced online judging system (OJS) characterized by its microservices architecture. Leveraging a myriad of cutting-edge technologies, including Docker for containerization, Docker Compose for orchestration, and Dapr for service mesh integration, Pantheonix exemplifies the pinnacle of modern software engineering practices. By exploring the intricacies of domain-driven analysis, microservices design, and operational paradigms, this paper unveils the rich tapestry of technologies underpinning the Pantheonix platform. From MongoDB and PostgreSQL for database persistence, Envoy for API gateway management and Redis for state storage to Rust and .NET for backend and Flutter for frontend, each technology plays a vital role in ensuring the scalability, reliability, and security of the Pantheonix ecosystem.

Keywords: *Competitive Programming, Microservices, Sandboxing, Dapr, Docker, Judge0 Evaluator, .NET, Rust, Flutter*

Introduction

In the context of the extensive digitization process of modern society, the proliferation of various types of online platforms stands out, such as those for e-learning, competitive programming, or candidate selection for positions within companies, all aimed primarily at facilitating the integration process of their users into the IT&C field by developing skills and knowledge in computer science and software engineering. And in this regard, the main catalyst of the learning process through such a platform is indeed the integration of a robust and interactive system for automated evaluation of software programs developed by users to meet the requirements stored in the platform's archive.

Taking into account the functional requirements associated with such an online platform, this paper aims to present a software solution entitled Pantheonix - a web application that integrates both a generous archive of problems similar to those proposed in competitive programming [8]

contests and technical interviews, as well as a service for automated evaluation of solutions submitted by users (which we will continue to refer to as "submissions"), which verifies in real-time the correctness of the received programs in relation to the specifications of input/output tests, execution time, and memory consumption, specified by users with the role of proposer, for a certain problem. To meet the standards of scalability, security, performance, and robustness specific to an online evaluation platform (for which we will use the term OJS - "Online Judgment System" hereinafter), Pantheonix benefits from a distributed architecture based on a microservices ecosystem [10], bringing together a multitude of modern technologies, including: Docker for isolating services in containers (also known as "containerization") [5], Docker Compose for orchestrating containers [9], Dapr (Distributed Application Runtime) for their integration [4], and Judge0 for the actual evaluation of submissions in an isolated and carefully monitored environment (known as "sandbox") [6].

Preliminary Section

The transition to microservices architecture represents a paradigm shift in the development and operation of complex software systems. This paper focuses on the application of microservices principles to the design and operation of an online judging system, Pantheonix. By leveraging Docker for containerization and Docker Compose for orchestration, Pantheonix embodies scalability, flexibility, and resilience, offering an efficient platform for users to engage in learning, testing, and evaluation activities.

The primary aim of this paper is to elucidate the process of building and operating a microservices architecture tailored for Pantheonix. It delves into the analysis of business domains, the design and implementation of microservices, and the operational aspects involved in running such a system in production. By detailing each stage of development and operation, this paper provides insights into the challenges and best practices associated with microservices-based architectures in the context of an online judging system.

Functional requirements for an Online Judgment System

When it comes to the proper functioning of an online evaluation system, it must provide users with the ability to carry out their learning, testing, comparison, and analysis activities within an ecosystem that also includes learning materials. Generally, an OJS (Online Judgment System) receives online submissions containing the code of solutions developed by users in response to specific requirements, which are then evaluated for correctness and performance. From the user's perspective, the source code is executed and evaluated in the cloud, not locally on their workstation [3].

Within any OJS, three standalone entities are distinguished: a set of authenticated users in the system, an archive of requirements proposed for resolution by individuals acting as "proposers," and most importantly, a flow of submissions containing source code intended for evaluation by the system. Applied to Pantheonix, a submission received by the evaluator consists of a tuple in the form of (problem, user, source code, programming language, <test cases>), where the source code is written in one of the programming languages provided by the platform and constitutes the user's proposed solution for a problem. The tests, which consist of a list of tuples in the form of (input,

expected output, score), are accompanied by other information such as time and memory limits from the metadata associated with a problem at the time of its creation and inherent to the process of evaluating subsequently provided submissions.

The evaluation process itself consists of a series of sequential steps, with the submission passing successively through the following phases:

- Preprocessing: In this stage, the evaluation environment of the submission is prepared in advance, creating local copies of input and output data for testing on the operating system hosting the evaluator.
- Compilation (only for compiled languages): In this stage, the source code of the submission is compiled using the compilers available within the evaluator, taking into account the programming language chosen by the user, with the result being the executable artifact.
- Execution: For each individual test, an isolated process is launched in a sandboxed environment (details of which will be explained later) containing the executable obtained in the previous step or the interpreter/just-in-time (JIT) compiler used to run the source code of the submission. This process has access to the current test's input data via standard input, provides the output data obtained via standard output, and restricts access to system resources according to the time/memory limits associated with the problem.
- Grading: In this stage, if no compilation (Compilation Error) or execution (Runtime Error) errors were previously recorded, the output data obtained in the previous step is compared to the correct data attached to the problem, and a corresponding score is awarded based on the result obtained. This score is subsequently returned to the user along with any reported errors.
- System release: In this post-evaluation stage, the resources associated with the system's evaluation process are released to avoid subjecting the evaluation system to redundant stress.

With that being said, within an online evaluation system, the following primary functional requirements can be identified, whose proper understanding and prioritization from the initial stages of development undoubtedly lead to the implementation of an ergonomic and appealing solution for end users [1]:

- Execution of source code: The platform must allow users to execute their own source code online, using compilers installed on the operating system hosting the evaluator.
- Support for real-time evaluation monitoring: Source code execution itself is irrelevant without a mechanism for reporting the evaluation result in real-time to the user, allowing them to monitor progress and the occurrence of any compilation/execution errors.
- Secure execution of source code: The actual evaluation of source code must be conducted by the platform in a controlled environment, isolated from the host operating system, both to prevent possible fraud attempts or compromise of the OJS through unauthorized access to the file system and extraction of test output from files, termination of other running processes, or network communication, and to monitor the resource consumption of the program submitted by the user in relation to the predefined limits within the problem [2].
- Support for a multitude of programming languages: The platform ideally should support a wide range of programming languages to accommodate a large user base, ideally providing the latest stable versions of their compilers as long as the integrity of historical submission evaluations is not compromised.

- Portability: The system must provide ergonomic installation mechanisms on various hardware platforms, depending on preferences, without compromising the predictability of the evaluation process. This characteristic can be guaranteed by using modern virtualization solutions, among which containerization through tools like Docker or Podman stands out for performance, security, and determinism.
- Flexibility for further development: An OJS must be extensible and allow for easy development of new functionalities in a reasonable time frame, achievable by opting for a distributed, microservices-oriented architecture that facilitates the integration of new technologies without compromising the structural integrity of the existing system.
- Consistency in submission evaluation: In an online evaluation system, the performance and correctness of a solution for a requirement are compared with those of other solutions, making it imperative that the evaluation process be carried out in a deterministic manner relative to the memory and CPU resources held by the hosting system of the OJS.
- Scalability: An online evaluation platform must be capable of supporting increasing data traffic and accommodating a considerable number of users, for which the system must be designed with scalability in mind, for example, adopting an architecture based on microservices that facilitates independent horizontal scaling of various component services.
- Availability: An OJS must be available for use continuously, even in the event of errors in the evaluation system's operation, a difficult task to implement for a monolithic architecture but inherent to a distributed system, which can support the partial operation of the platform even in the event of a failure of one of the services by limiting the dependencies between microservices.
- Accuracy in submission evaluation: An online evaluator must provide, above all, the most precise and objective results possible following the evaluation of submissions, which can be used as valid parameters in the process of comparing solutions in terms of performance and correctness.
- Management of a submission queue: The system must implement a mechanism for managing concurrent evaluations of submissions submitted by users, using management queues to prioritize their execution order based on available resources.
- Interoperability: In general, an OJS must support integration with other development systems and tools, such as continuous integration (CI) systems, version control systems (VCS), or reporting and statistics issuance.
- Implementation of anti-fraud strategies: The system must minimize attempts to defraud the evaluation of submissions by running the source code in a secure environment, restricting access to other users' solutions during competitions, or providing progressive solution guidance at the user's request.

Overview of the application architecture

Taking an overview, the platform is structured according to the client-server architectural paradigm, where the client (named Midgard) is represented by a Single Page Application (SPA) implemented using the Flutter cross-platform frontend framework in combination with the MVVM architectural style offered by the Stacked meta-framework, while the server (named Asgard)

consists of a cluster of interconnected microservices within a service mesh using the distributed systems runtime provided by Dapr, exposing a REST API to the client.

Microservices represent a modern architectural style widely used in the development of applications on distributed systems that need to be robust, scalable, autonomous, and extensible [10]. A microservices-oriented architecture consists of a set of compact, independent services with limited responsibility within a bounded context, providing a standalone and homogeneous group of functionalities. A bounded context is a fundamental concept in modeling a business domain, as stated by the Domain-Driven Design (DDD) architectural paradigm promoted by Eric Evans [7]. It represents a natural partitioning that delimits a logical and explicit business context in which a model exists.

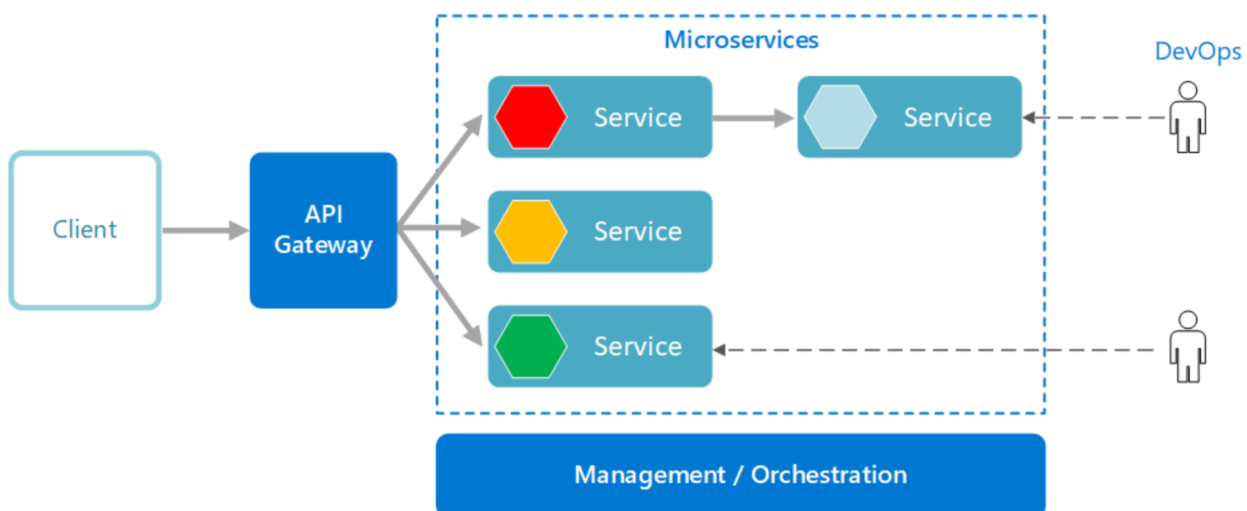

Photo 1 - Minimal Microservices Architecture

Source: https://learn.microsoft.com/en-us/azure/architecture/includes/images/microservices-logical.png

Unlike a monolithic architecture, developing an application using microservices involves a decision-making and technical process with a high level of complexity, structured into three main steps, which will be exemplified through the architecture adopted by Asgard - the backend of the Pantheonix platform:

1. Domain analysis;
2. Design and implementation of microservices;
3. Operating the system in production.

During the domain analysis phase, the primary goal is to understand and identify the primary functional requirements of the software solution that needs to be developed, followed by refining them into business contexts using the strategic DDD method for macro-architectural design. Regarding a platform for automated program evaluation like Pantheonix, the key functionalities that such a system must provide include:

- Support for user authentication and account creation within the platform;
- Management of user profiles allowing personalization and viewing of personal data within them;
- Support for proposing problems within the platform that can be solved by users;

- Management of proposed problems still in the creation stage, which should be accessible to users in an interactive interface for viewing, solving, or editing (by the author);
- Support for managing archives of input/output tests associated with problems and required for evaluating submissions;
- Support for submitting solutions to problems in the form of source code to be evaluated according to their specifications;
- Support for viewing the results of submission evaluations within the platform.

With that being said, the following fundamental contexts and subdomains, as well as the relationships between them within the business of an online evaluator, stand out:

- Problem and submission management are the core contexts of the platform, with the main rationale behind an OJS being the causality relationship between requirements and solutions.
- Problem proposal and submission evaluation are subdomains of the core contexts with roles in augmenting them.
- Test and user profile management are standalone secondary contexts that extend the basic functionalities of the platform's business.

During the design and implementation phase of microservices, the starting point is the identification of the main bounded contexts made in the previous domain analysis stage. Each context is associated with a dedicated microservice, following an empirically used rule in designing distributed systems. Thus, considering both the benefits and drawbacks of adopting a microservices-oriented architecture for Pantheonix, we will present the structural profile of the platform alongside the design decisions that led to it.

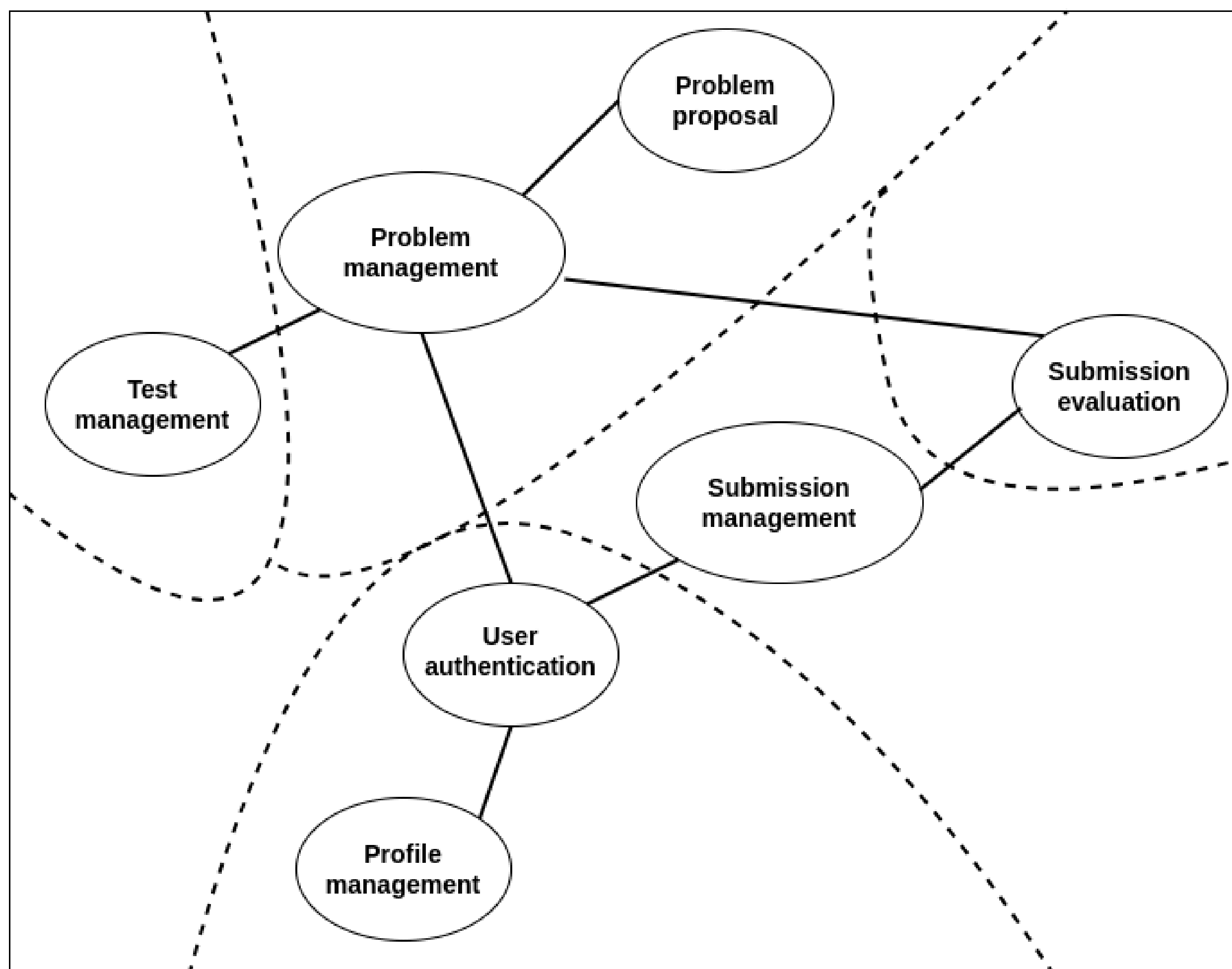

Photo 2 - Pantheonix Bounded Contexts

Author: Sami Bărbuț-Dică

Given the flexibility offered by microservices in integrating a variety of technologies, Asgard is organized into three hierarchical levels of services, depending on the degree of exposure of their APIs to the outside. Each of these is implemented using a set of frameworks advantageous for the bounded context they serve:

- Primary level - includes an API Gateway (named Odin) implemented using Envoy's reverse proxying technology, acting as the main mediator of traffic between the Midgard web client and the backend microservice cluster. It plays a key role in enforcing security policies and optimizing the resilient interaction between microservices and end applications.
- Secondary level - comprises the microservices responsible for implementing the fundamental business contexts of the platform and are directly connected to the API Gateway through the service mesh provided by Dapr:
    - User management service - also known as Quetzalcoatl, is implemented as an authentication microservice using the FastEndpoints backend framework based on .NET 7 Minimal API and Identity Framework to manage user identities (personal data and roles) securely. Quetzalcoatl does not directly communicate with other

microservices; its role is to generate access tokens used to validate requests to other services using authentication/authorization middleware within them.
    - Problem management service - also known as Enki, is implemented using the ABP backend framework based on .NET 7 to model the bounded context associated with problems according to the DDD architectural style. Enki exposes a REST API for manipulating problem data through two separate sets of endpoints associated with both normal platform users and proposers.
    - Submission management service - also known as Anubis, is implemented using the Rocket backend framework based on Rust to serve submission management functionalities.
- Tertiary level - includes internal use microservices subordinate to those in the secondary level to extend their API in a distributed manner:
    - Test management service - also known as Hermes, consists of a standalone web server implemented in Dart, exposing a gRPC API used by Enki to upload and query data about the evaluation tests of proposed problems.
    - Submission evaluation service - consists of a cluster of local instances of the Judge0 evaluator exposed through an Nginx load balancer that evenly distributes submission evaluation requests issued by the Anubis microservice concerning each test of a problem and its associated time and memory limits. Within an evaluator instance, submissions are evaluated in a sandbox to minimize security risks to the rest of the system following the compilation and execution of code from untrusted sources.

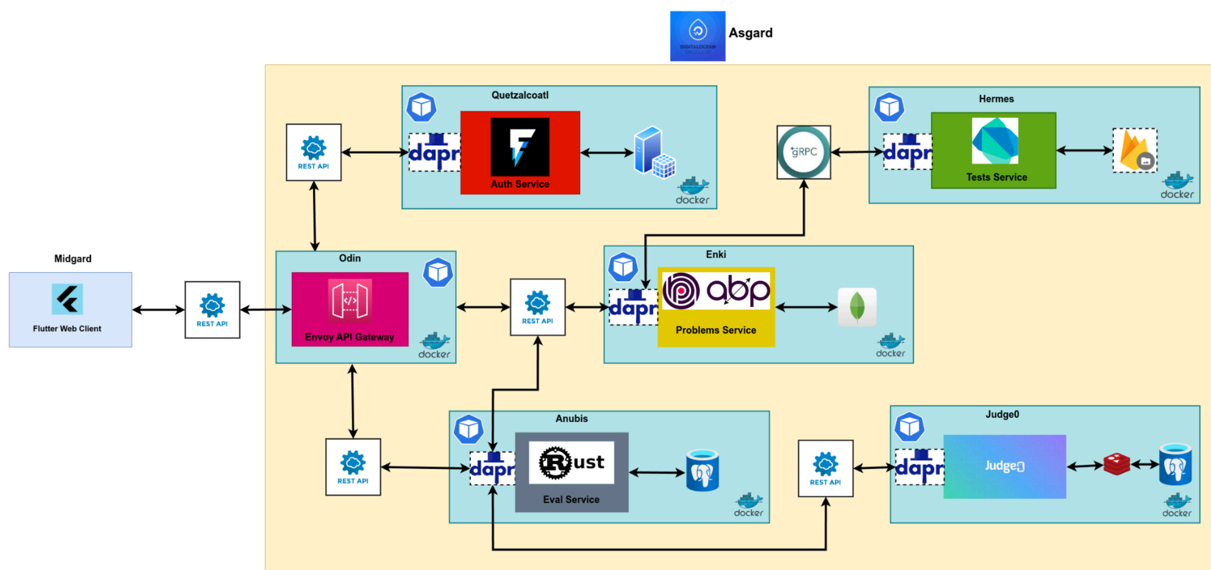

Photo 3 - Pantheonix Architecture

Author: Sami Bărbuț-Dică

Furthermore, the distributed architecture imposed by the use of microservices can also be applied at the data persistence level by creating isolated databases dedicated to each microservice. Similar to the source code of a microservice, updating the schema of a database does not affect others, minimizing the risks that may arise from integrating new functionalities. Each microservice

can use a DBMS specifically tailored to the requirements of the implemented business context, as observed in the case of services in Asgard:

- Quetzalcoatl utilizes Microsoft SQL Server for user data persistence due to its native integration with .NET 7.
- Enki uses MongoDB for persisting problems along with the metadata of input/output tests due to the native compatibility of NoSQL data schema with DDD modeling principles.
- Anubis uses PostgreSQL for persisting the results of submission evaluations to reduce inconsistencies between its local schema and that used by Judge0, which also uses PostgreSQL.
- Hermes utilizes Firebase Cloud Storage for storing input/output test files in a remote filesystem due to the native compatibility between the Firebase SDK and Dart.

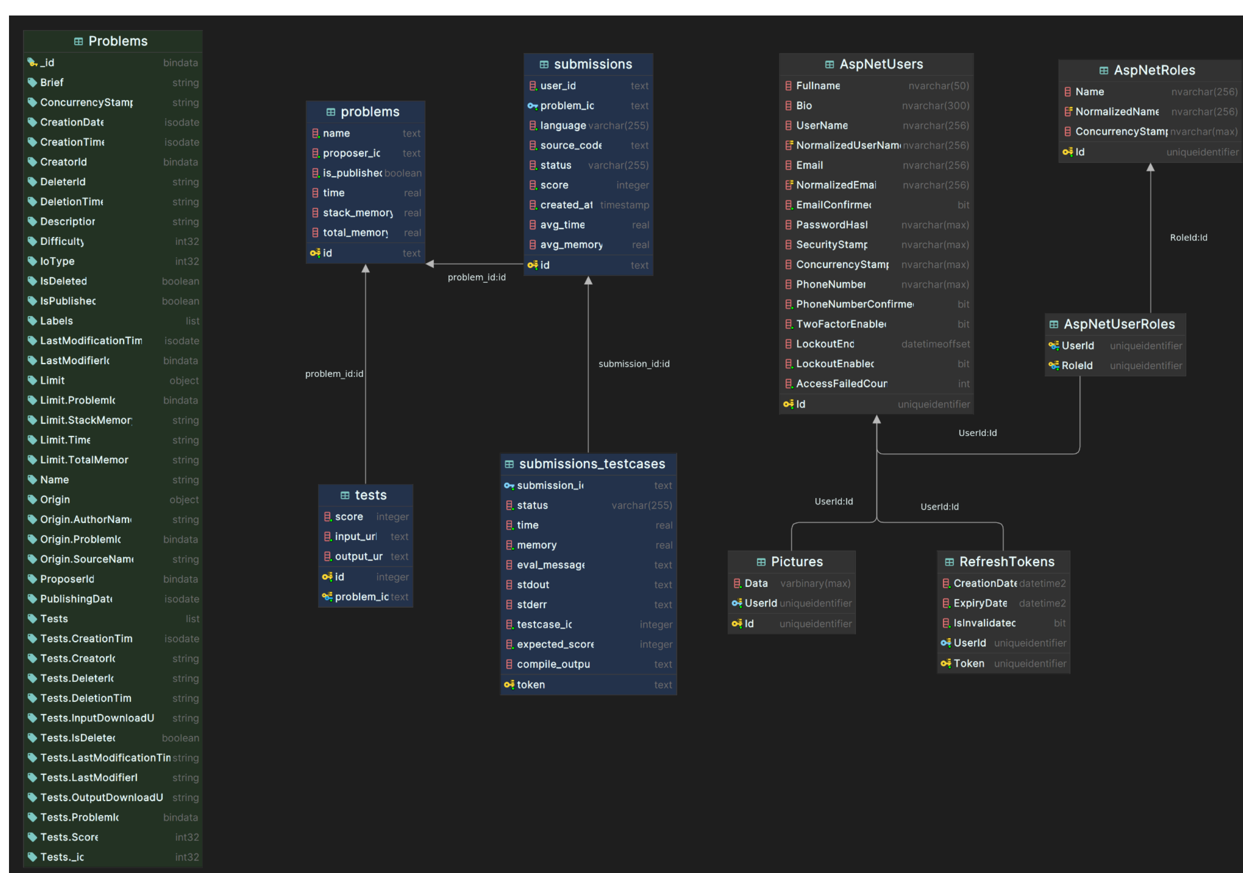

Photo 4 - Pantheonix ERD (from left to right: Enki with green, Anubis with blue and Quetzalcoatl with gray)

Author: Sami Bărbuț-Dică

The final step in developing a microservices-oriented application is operating it in production, by choosing the appropriate DevOps technologies. In this case, Pantheonix uses Docker for containerizing the microservices and Docker Compose for orchestrating them as follows:

- For each microservice, a Docker image is defined using a Dockerfile that specifies all the SDKs and dependencies of the service in question.
- All containers are orchestrated by Docker Compose using a docker-compose file that includes YAML definitions for:

    - Microservice containers, API Gateway (Envoy), and load balancer (Nginx).
    - Database containers.
    - Containers associated with Dapr, including sidecars attached to the processes of microservice containers, the Zipkin telemetry service, the Redis state store service, and the RabbitMQ pub-sub service.
- Volumes used for persisting data on the host system where Pantheonix is installed.
- Each service is configured through a configuration file with environment variables provided to the container at runtime by Docker Compose based on the specifications in the docker-compose file defined for Asgard, as well as directly specified in the docker-compose file regarding dependencies between containers, the context/path of the Dockerfile, exposed ports, used volumes, available system resources, etc.

Conclusions

Pantheonix represents a platform designed for the automated evaluation of programs online, primarily aimed at providing its users with an engaging and intuitive environment for both learning and mastering programming concepts necessary for passing an exam at relevant institutions, participating in programming contests, or even securing desired job positions.

The main mechanism through which the platform supports the e-learning process in its current development stage is the diverse archive of requirements formulated according to the pattern adopted by competitive programming problems. These requirements accept solutions written in a programming language chosen by the user from a series of ten languages supported by the evaluator, such as C, C++, Rust, Java, C#, Golang, etc. Users can utilize the platform's built-in code editor to edit and submit their solutions for evaluation. These submissions are run in a secure sandbox environment against the input tests associated with the problem to be solved. The feedback obtained from analyzing the correctness of the output data and the consumption of time and memory is then provided in real-time to the user through the platform's interface.

In addition to regular users, Pantheonix supports a role-based authorization system for proposers, who are granted rights within the platform to create new problems with statements in Markdown format and tests for them uploaded as zip archives. Proposers have veto power over the status of a problem, which can either be published and thus visible to all users for solving, or in an experimental stage, accessible only to the author for editing and analyzing the behavior of submissions in relation to the chosen tests and time and memory limits.

To meet the functionality and performance standards specific to an online evaluation system, the Pantheonix platform is designed based on a microservices-oriented architecture and developed using modern technologies and frameworks to meet the needs of a distributed system-based OJS (Online Judgment System). The main features of the platform and the technologies facilitating their implementation include:

- Scalability: Utilizing a distributed architecture allows the system to dynamically adapt to platform traffic by horizontally scaling individual microservices independently using container orchestration technologies such as Docker Compose.
- Portability: Platform microservices are containerized using Docker and orchestrated through Docker Compose, enabling Pantheonix to run independently of the operating system and be delivered to users in both multi-tenant and single-tenant modes.

- Security: Security is a key reference point within the platform, implemented at all levels using best practices:
  - Submissions are executed in sandboxed environments using the evaluator provided by Judge0.
  - Microservices are isolated in containers and grouped in a private network using Docker.
  - Microservices communicate indirectly with the outside through an API Gateway defined using Envoy.
  - Users are authenticated and authorized based on roles within the system using JWT access tokens and refresh tokens validated at each microservice level through a dedicated middleware.
- Extensibility: Utilizing microservices facilitates the integration of a wide range of technologies and paradigms to independently meet the functional requirements of each microservice, from web frameworks (such as Rust-based Rocket, .NET Core-based FastEndpoints and ABP for backend, and Flutter-based Stacked for frontend) to databases (Microsoft SQL Server, PostgreSQL, MongoDB, Firebase Cloud Storage), and from protocols (REST HTTP, gRPC) to architectural styles (Domain Driven Design, Clean Architecture, MVVM).
- Performance: To streamline the evaluation of submissions and provide results to users in minimal time, the evaluation microservice uses an Nginx-based load balancer to distribute traffic among multiple Judge0 instances and a Redis-based cache to store test data.
- Resilience: Communication between microservices is facilitated by a robust API provided by the Dapr distributed services runtime, defining a service mesh that allows microservices to interact with other microservices, state store components, or pub-sub through a homogeneous HTTP or gRPC-based API.
- Correctness: At the microservices level, functionalities are validated using unit and integration testing frameworks such as XUnit for .NET Unit Testing, TestContainers for dynamic creation of containers for a microservice's external dependencies in Integration Testing, and native testing frameworks in Dart and Rust.

Thus, Pantheonix aims to provide its users, whether students or teachers, individuals preparing for technical interviews, or institutions wishing to use a customized instance of the platform in single-tenant mode, with an ergonomic, versatile, and modern software solution that unifies key functionalities for a learning environment tailored to deepening knowledge in the field of computer science. Although these functionalities may be encountered in other platforms or reproduced manually in a local development environment, their use, unlike the Pantheonix experience, involves redundant cognitive effort on the part of end users and distracts from the truly essential aspects for them: deepening the specialized knowledge necessary to achieve their goals and dreams.

Bibliography

[1] Kamod, Nachiket Devendra, and Rahul Namdev Jadhav. "*A Secure and Scalable System for Online Code Execution and Evaluation using Containerization and Kubernetes.*" *Journal of Emerging Technologies and Innovative Research (JETIR)*, vol. 10, no. 2, 2023, p. 151-159. *jetir.org*, https://www.jetir.org/papers/JETIR2302226.pdf.

[2] MAREŠ, Martin, and Bernard BLACKHAM. "A New Contest Sandbox." *Olympiads in Informatics*, vol. 6, no. 100-109, 2012, p. 10. *mj.ucw.cz*, https://mj.ucw.cz/papers/isolate.pdf.

[3] Watanobe, Yutaka, et al. "*Online Judge System: Requirements, Architecture, and Experiences.*" *International Journal of Software Engineering and Knowledge Engineering*, vol. 32, no. 6, 2022, p. 917-946. *World Scientific*, https://arxiv.org/pdf/2304.14342.

[4] *Dapr - Distributed Application Runtime*, https://dapr.io. Accessed 1 May 2024.

[5] *Docker: Accelerated Container Application Development*, https://www.docker.com. Accessed 1 May 2024.

[6] *Judge0 CE - API Docs*, https://ce.judge0.com. Accessed 1 May 2024.

[7] *Wikipedia*, https://en.wikipedia.org/wiki/Domain-driven_design. Accessed 1 May 2024.

[8] "Competitive programming." *Wikipedia*, https://en.wikipedia.org/wiki/Competitive_programming. Accessed 1 May 2024.

[9] "Docker Compose overview." *Docker Docs*, https://docs.docker.com/compose. Accessed 1 May 2024.

[10] "Microservices architecture design - Azure Architecture Center | Microsoft Learn." *Learn Microsoft*, https://learn.microsoft.com/en-us/azure/architecture/microservices. Accessed 1 May 2024.