Report: Application Budget gamer

Author: Tjaša Domadenik

January 2023

1 Application description

As part of my cloud computing course I created a web application called Budget gamer which compares games by prices obtained from Steam and Gog online stores. It consists of 4 microservices. Figure 1 shows the microservice architecture. **Steam and Gog parser** are used to extract and convert game data from external sources. **Game data** combines the data obtained from different parsers (Steam and Gog) into lists, and the **Store comparator** microservice compares products according to the price and benefits offered by the individual online store.

2 Framework and development environment

The microservices were implemented with Java (Quarkus framework) and Angular was used for the frontend. I wrote the code in the IntelliJ IDEA development environment.

3 Github repositories

1. Microservice **Steam parser**: `https://github.com/rso-project-price-comparison/steam-parser`
2. Microservice **Gog parser**: `https://github.com/rso-project-price-comparison/gog-parser`
3. Microservice **Game data**: `https://github.com/rso-project-price-comparison/game-data`
4. Microservice **Store comparator**: `https://github.com/rso-project-price-comparison/store-comparator`
5. **Web application**: `https://github.com/rso-project-price-comparison/frontend-budget-gamer`

4 DockerHub repositories

1. Microservice **Steam parser**: `https://hub.docker.com/repository/docker/tjasad/rso-steam-parser`
2. Microservice **Gog parser**: `https://hub.docker.com/repository/docker/tjasad/rso-gog-parser`
3. Microservice **Game data**: `https://hub.docker.com/repository/docker/tjasad/rso-game-data`
4. Microservice **Store comparator**: `https://hub.docker.com/repository/docker/tjasad/rso-store-comparator`

5 Architecture scheme

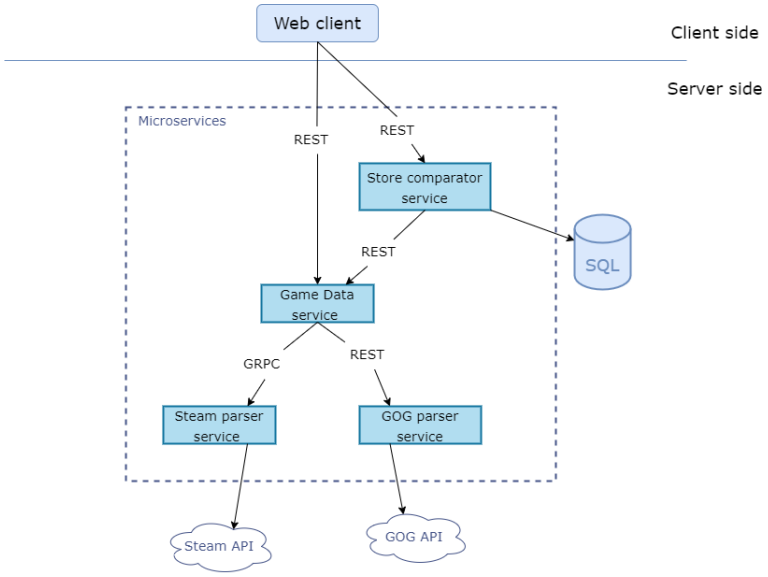

Figure 1: Architecture scheme.

6 Diagram of interactions between microservices

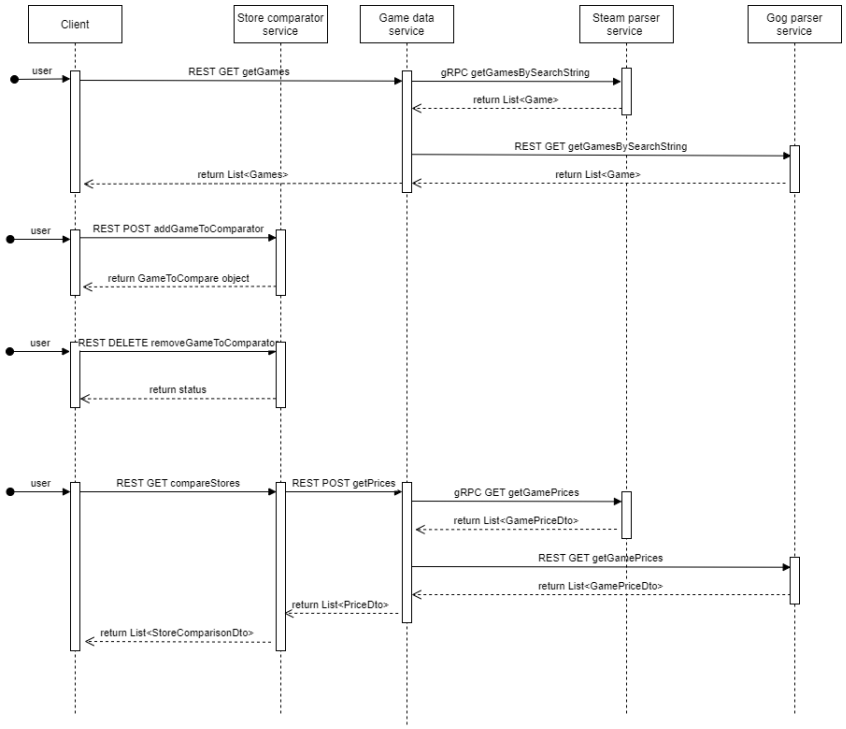

Figure 2: Diagram of interactions between microservices.

7 Functionalities

7.1 Store comparator microservice

1. Functionality 1: Compare online stores
2. Functionality 2: Add game to the comparator
3. Functionality 3: Delete game from the comparator

7.2 Game data microservice

1. Functionality 1: Fetch games from all online stores based on search string
2. Functionality 2: Get game prices from different online stores

7.3 Steam parser microservice

1. Functionality 1: Fetch Steam games based on search string

2. Functionality 2: Retrieve Steam game prices

7.4 Gog parser microservice

1. Functionality 1: Fetch GOG games based on search string
2. Functionality 2: Get GOG game prices

8 Use cases

- Search for games by search string,
- add games to the comparator,
- delete games from the comparator,
- game comparison.

9 Kubernetes

With the help of deployment.yaml files I added the Docker images of microservices to Kubernetes hosted on the **Google Cloud** platform. I also enabled the use of a single address for all the microservices by including ingress (in the ingress.yaml file).

10 Configuration

10.1 Configuration files

I used configuration files (application.properties file in Quarkus) for non-sensitive basic configuration (application name, version, log format,...) and values that are used in local development (for example data for accessing the local database).

10.2 Environmental variables

I saved the username and the dockerhub token (used for automatic creation of docker images with github actions) in github secrets.

10.3 Configuration server

I managed to add Consul to kubernetes, but due to connection problems and time constraints, I had to store data in a different way.

11 Health checks

I implemented health checks in the **Game data** and **Store comparator** microservices.

11.1 Game data

I added a **liveness health check** test. The state can be changed via REST calls:

- POST: `/game-data/api/v1/gamedata/liveness/disable`
- POST: `/game-data/api/v1/gamedata/liveness/enable`

Besides that I also added a **readiness health check** which monitors the connection to the Gog parser microservice.

11.2 Store comparator

Added a **readiness health check** that checks the connection to the Game data microservice, in addition, Quarkus automatically adds a health check that checks if the database works properly.

12 Metrics

I used the Datadog service to collect Kubernetes environment metrics. The configuration file for setting up the datadog agent is available at `https://github.com/rso-project-price-comparison/gog-parser/blob/master/k8s/datadog-values.yaml`.

In image 1 we can see the display of metrics about pods (consumption according to CPU and RAM) in Datadog.

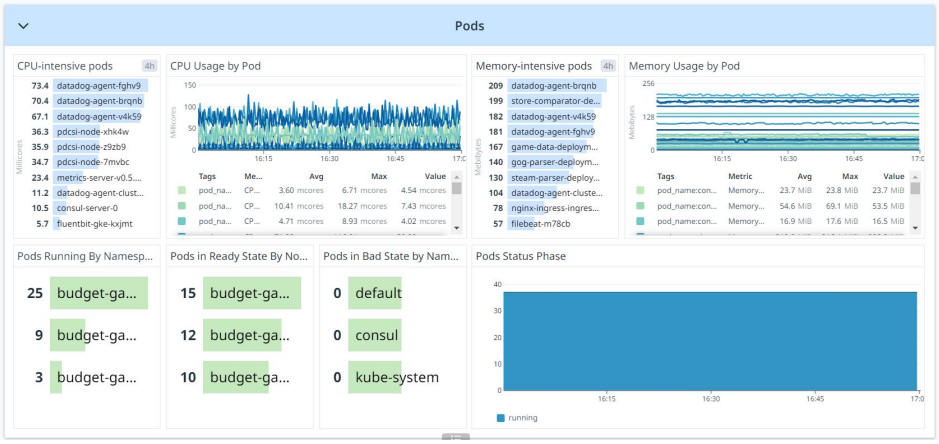

Figure 3: Pods metrics.

13 External APIs

I obtained data about games and their prices from 4 different external APIs.

Steam store APIs:

- ISteamApps API [1] provides general information about games.
- StoreFront API [2] provides information on the price of games.

GOG store APIs:

- GOG API [3] provides general information about games.
- Data about the search of games according to the search string was extracted from the official Gog website [4].

14 OpenAPI

All microservices except Steam parser are documented using OpenAPI. The Swagger UI is available at:

- Game data: `/game-data/q/swagger-ui/`
- Gog parser: `/gog-parser/q/swagger-ui/`
- Store comparator: `/store-comparator/q/swagger-ui/`

I removed the OpenAPI from **Steam parser**, as it communicates with other microservices internally via gRPC.

15 Centralized logging

I implemented central logging in microservices with the open source Filebeat agent, which I added to Kubernetes via the configuration file filebeat-kubernetes.yaml. The logs were sent to the online platform logit.io.

I logged entries and exits to methods (marked with ENTRY and OUT markers). In addition, I also logged possible errors and added special logs to the Store comparator when store products were compared (marked with STORE_COMPARISON_MARKER marker).

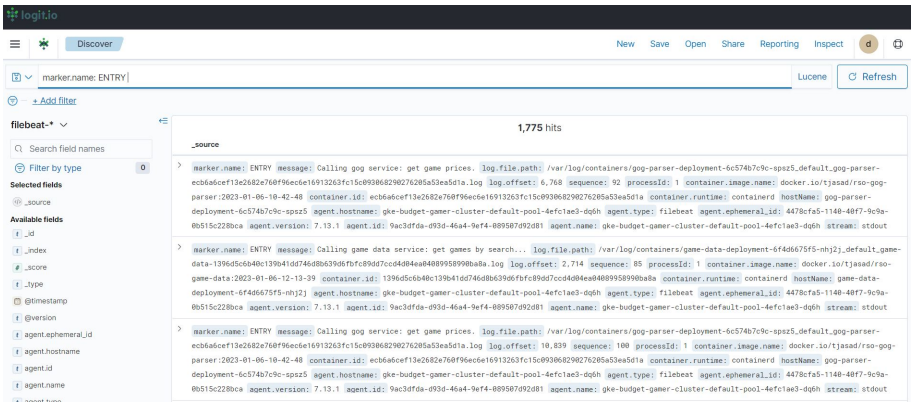

Figure 4: Display of ENTRY marker logs.

16 Isolation and fault tolerance - demonstration

I've added a **circuit breaker** to the **Game data** microservice, which can be demonstrated with a POST call to the url: `/game-data/api/v1/gamedata/price?circuitBreakerTest=true`

An "artificially" created error is randomly fired in the method, causing some calls to fail. If we try to call the service a few times, we will notice that after a certain number of attempts we can no longer send requests. The events can also be monitored in the logs as shown in the picture 5.

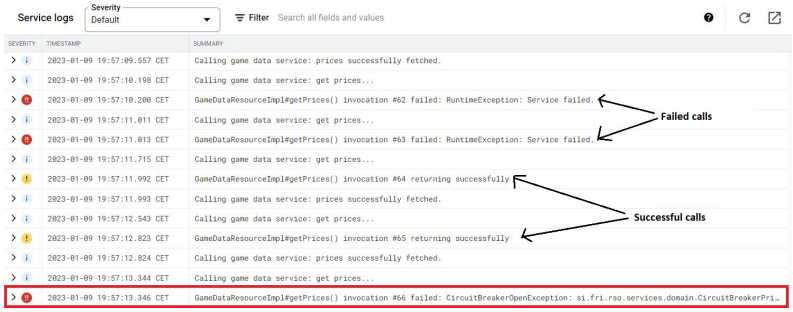

Figure 5: Circuit breaker demostration in logs.

In addition, I also added a demonstration for **retry, fallback** and **timeout** annotations. In the Gog parser microservice, I added a timeout annotation and logic to the *getGamesBySearch* method, which ensures that the timeout is triggered randomly. In the Game data microservice (when calling the mentioned Gog parser method) I added a retry, which tries to call the method 2x. If an error occurs in both cases, it calls the fallback method, which executes the second Gog parser method without a timeout.

References

[1] *ISteamApps*. URL: https://partner.steamgames.com/doc/webapi/ISteamApps (visited on 08/01/2023).

[2] *StorefrontAPI*. URL: https://wiki.teamfortress.com/wiki/User:RJackson/StorefrontAPI (visited on 08/01/2023).

[3] *Gog api*. URL: https://api.gog.com/v2/ (visited on 08/01/2023).

[4] *Gog store*. URL: https://embed.gog.com/ (visited on 08/01/2023).