



SAPIENZA  
UNIVERSITÀ DI ROMA

# Cloud Computing

## Game Discovery on Cloud

Master's degree in computer science

**Student:** Federico Ziegler (1808702)

**Email:** [ziegler.1808702@studenti.uniroma1.it](mailto:ziegler.1808702@studenti.uniroma1.it)

**Teacher:** Emiliano Casalicchio

**Academic Year:** 2023-2024

# 1. Index

Introduction .....	1
System architecture .....	1
Microservices .....	3
Authorization Service .....	3
Security token handling.....	3
Games Service .....	5
Preferences Service .....	10
Reviews Service .....	12
Gateway API .....	14
About Game Discovery .....	21
Aws Deployment .....	22
Scale-in/out.....	23
Tests .....	24
Test 1 .....	25
Test 2 .....	28
Conclusions .....	30

## 2. Introduction

The project started for the purpose of creating a game discovery platform where users would discover new games based on preferences or by searching on the engine. The simple idea began to grow and escalated quickly into a game platform that could also let users make reviews, submit their own games via an approval mechanism and protect all the resources behind gateway and authorization services.

The system for game platform is divided into four different sectors to enhance development and distribute the responsibilities:

1. Users' management
  - a. To add users and roles into the game platform
2. Users' preferences management
  - a. To add users' information such as wish listed games, favourite genres, etc... that could help for the game discovery engine.
3. Users' reviews management
  - a. To handle user game reviews and retrieval of critical information for the search engine such as review quantity and average rating of a certain game.
4. Games management
  - a. To handle game submissions and updates.
  - b. To handle game information such as genres, developers, publishers and so on...

Game administrators are the ones that will oversee approving game submissions and updates and will be the only ones that can access specific entry points of the application.

## 3. System architecture

The system architecture is composed of multiple layers:

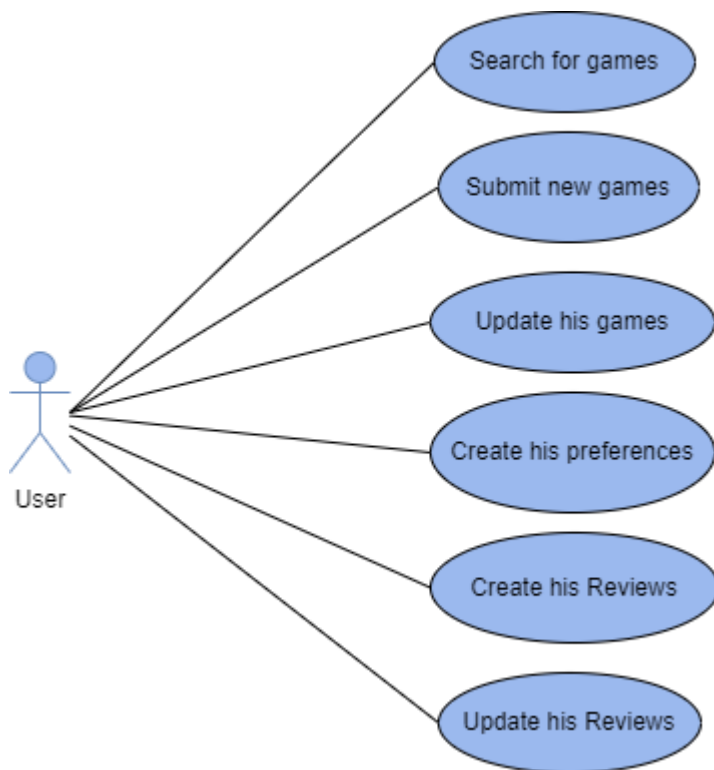
1. Presentation layer (Gateway API): that is the sole entry point for the whole application and acts as a reverse proxy that makes the lower layers be able to interact one with the other without being accessible from the outside. It is accessible via Swagger directly on browser even though it's not the only way to access it.

2. Business logic layer (Backend)
3. Persistence layer that interacts with the RDBMS.

The whole application has been developed using the following technologies:

1. Spring boot (with swagger) for the Gateway API.
2. Spring boot (with swagger) for the business logic layer using the microservices pattern to develop it.
3. MySql as the RDBMS for the persistence layer.

The development of the whole project began with some simple scenarios defined by the following use case:



*Figure 1: Simple use case scenarios*

## 4. Microservices

The backend layer is developed using Spring Boot with integrated Swagger and Spring Security, following a microservices architecture to ensure scalability and fault isolation. Communication between microservices and the Gateway is handled via REST APIs and each microservice has its own dedicated MySQL database for data persistence, adhering to the microservices pattern.

For enhanced security the microservices in production environments are only accessible via Gateway API and will be completely isolated even though they can stay in the same cluster.

Every microservice accepts JWT tokens in the request headers of its own REST API endpoints. However, only the Authorization Service can issue new tokens and validate existing ones, making it the first service to be presented in this report.

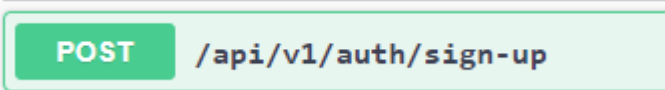
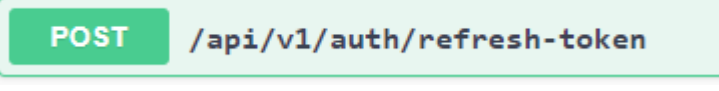
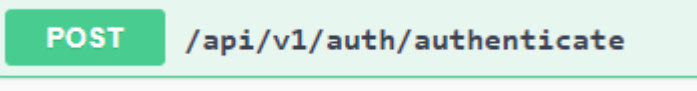
### 4.1 Authorization Service

The Authentication Service stores user sensitive information that is used in the authentication process to generate new JWT access tokens and refresh ones. It also stores role information related to the users of the application. In the current implementation there are only two roles possibly accepted:

- ADMIN
  - Has access to all exposed API endpoints.
- USER
  - Has not access to ADMIN authorized endpoints.

#### Security token handling

The main authentication process starts with the user registration or login, but the user has also the possibility to use its refresh token to generate new tokens.

HTTP METHOD + API	NOTES
 A green button labeled 'POST' followed by the endpoint '/api/v1/auth/sign-up' in a light blue box.	Request Body passed with username, email and password
 A green button labeled 'POST' followed by the endpoint '/api/v1/auth/refresh-token' in a light blue box.	JWT token contained on the Authorization header. This method is used
 A green button labeled 'POST' followed by the endpoint '/api/v1/auth/authenticate' in a light blue box.	Request Body passed with email and password.

Once the user got the JWT access token he can then transmit it on the request header when accessing to other endpoints.

On the diagrams below I show with more detail how the authentication process is handled:

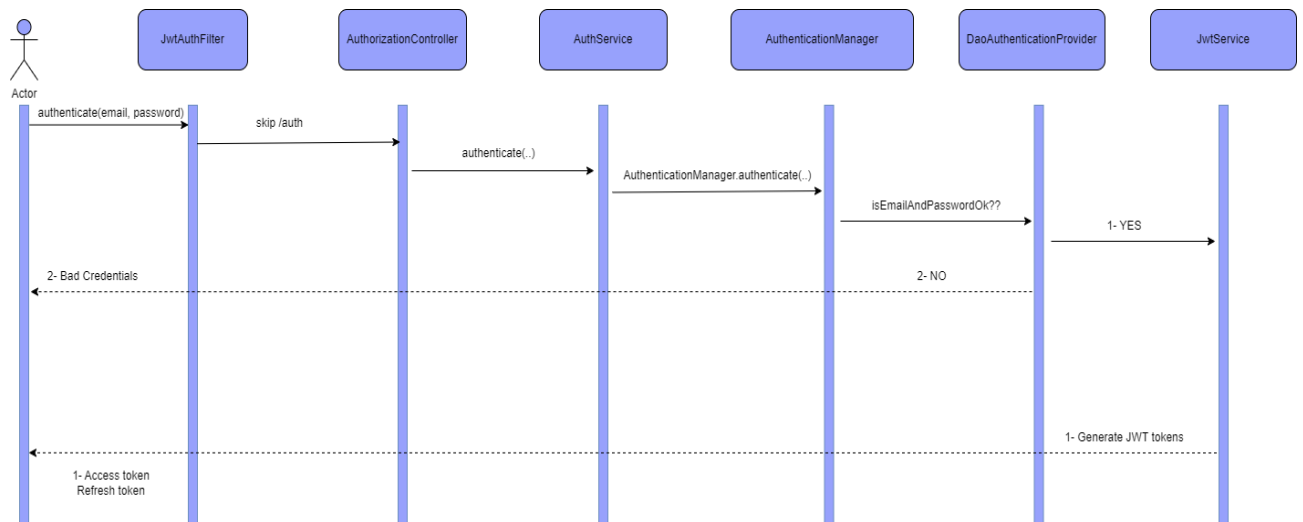


Figure 2: Authentication process via username and password

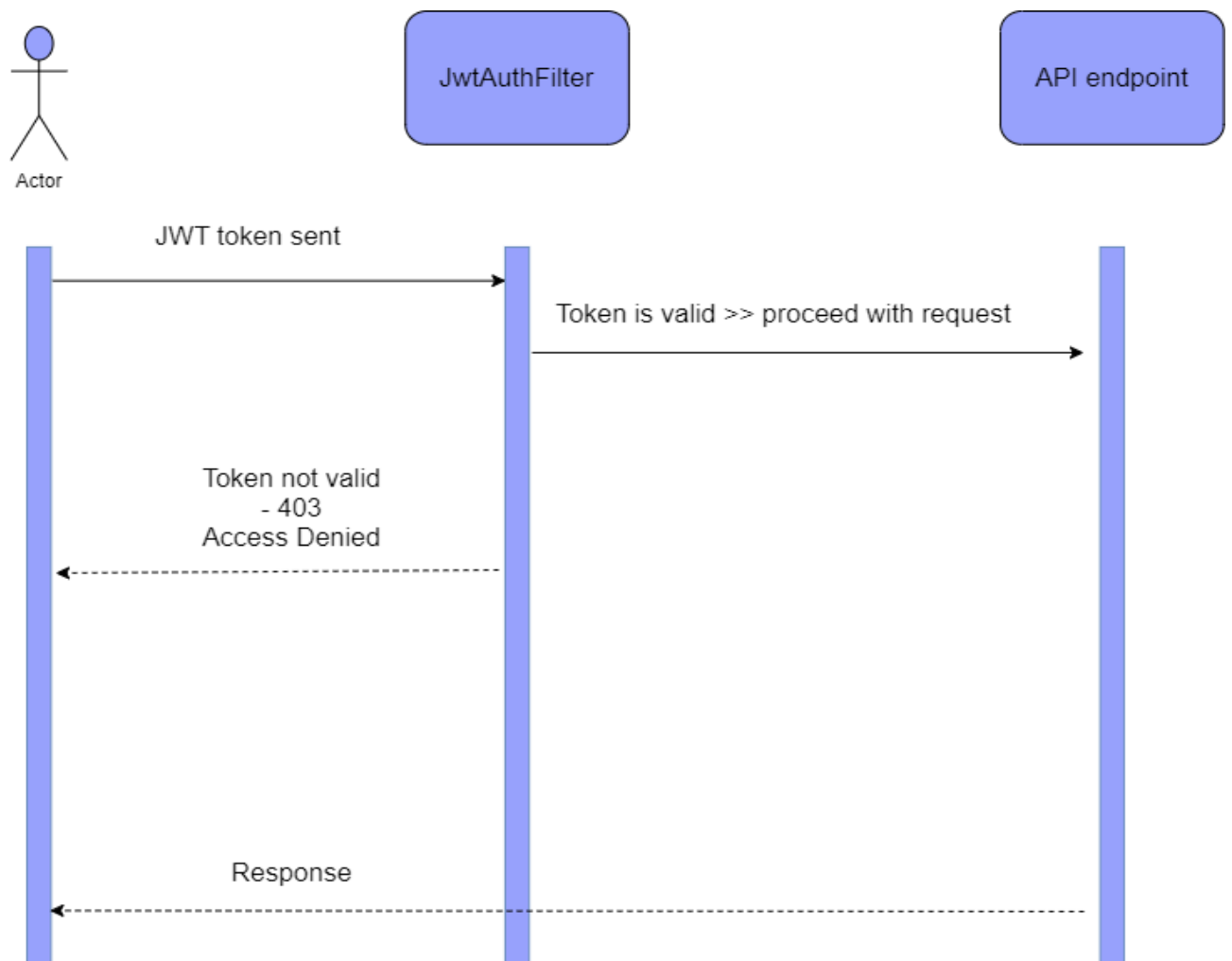


Figure 3: Authentication process when sending a JWT token (token validation)

There are also other API endpoints but for simplicity I won't show all of them. This is an example of other endpoint this service has:

HTTP METHOD + API	NOTES
<b>PATCH</b> <code>/api/v1/user/{id}</code>	This API makes it possible to update current username sending it in the 'username' field of the request body.
<b>DELETE</b> <code>/api/v1/user</code>	A user can delete its own account passing its email as a request param and a JWT token generated for that email.

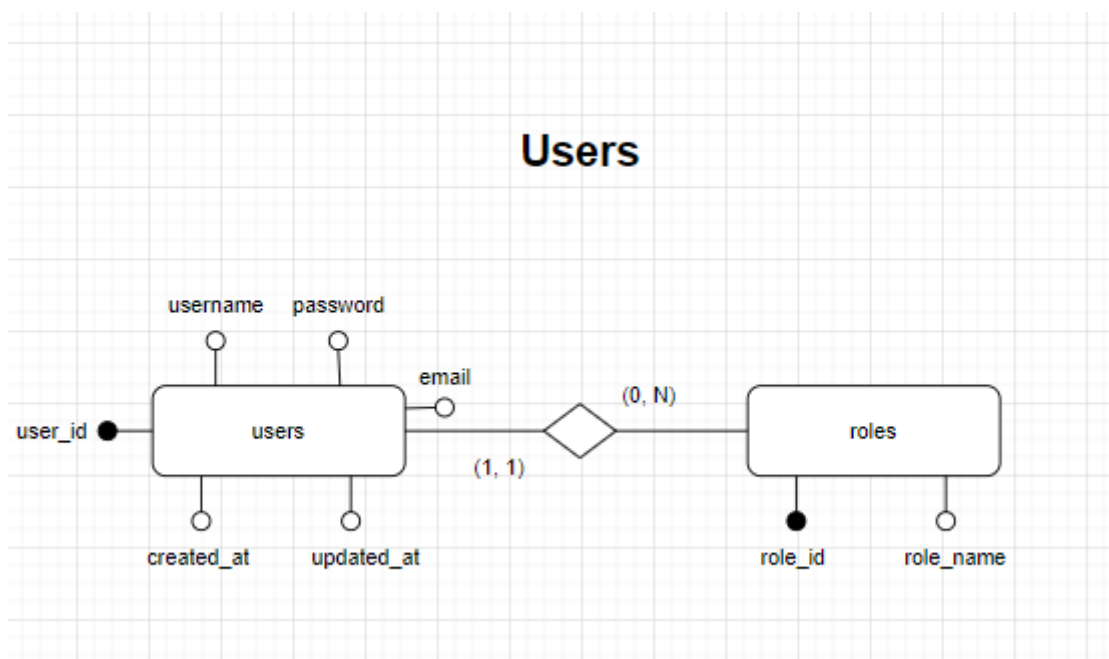


Figure 4: Authorization Service DB architecture.

## 4.2 Games Service

The games service is responsible for storing, adding, and removing games and their submissions. Game submissions have a status:

- **PENDING**
  - When game was just submitted and needs approval
- **APPROVED**
  - When the game has been approved by an administrator and can now be found by the search engine.
- **REJECTED**
  - When the game has been rejected and needs to be deleted.

When a game has been approved by an administrator it can then be updated, in that case there will be a new game entry in the 'updatable games' table that will need further approval. When approved it will directly update the current game entry. Whether it is approved or rejected the updatable game entry will be always deleted and a new one will be created for a new version.

Games have many attributes such as 2D, 3D, isPvP, isPvp and so on... shown on the database architecture at the end of the section.

On the table below I show some of the API endpoints exposed by the service:

HTTP METHOD + API	NOTES
<b>POST</b> <code>/api/v1/game</code>	API for game submission via request body.
<b>PATCH</b> <code>/api/v1/game/{id}</code>	API for updating averageRating and reviewQuantity of a game. These fields are inside the request body.
<b>GET</b> <code>/api/v1/games</code>	API for search by all the games parameters such as name (%like query), releaseDate, genres etc...
<b>GET</b> <code>/api/v1/games/with-pending-status</code>	API for administrators to fetch the game with pending status.
<b>GET</b> <code>/api/v1/games/by-discovery-criteria</code>	Acts like the <code>/api/v1/games</code> search but gives the possibility to exclude game ids. Mainly used for the discovery of games used by the Gateway.
<b>POST</b> <code>/api/v1/updatable/game</code>	API for the submission of game updates 'updatable games' entries via request body.

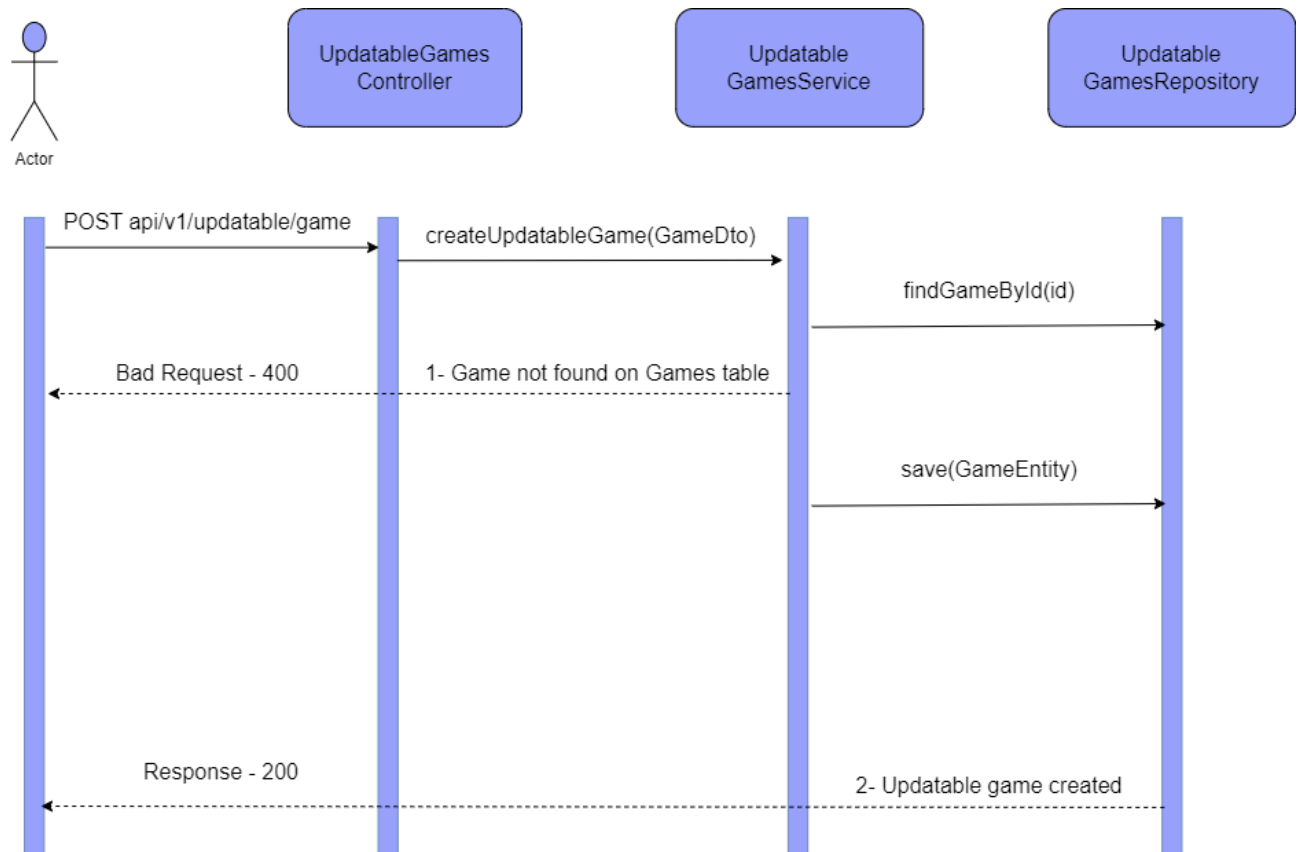


Figure 5: Diagram showing updatable game creation process.

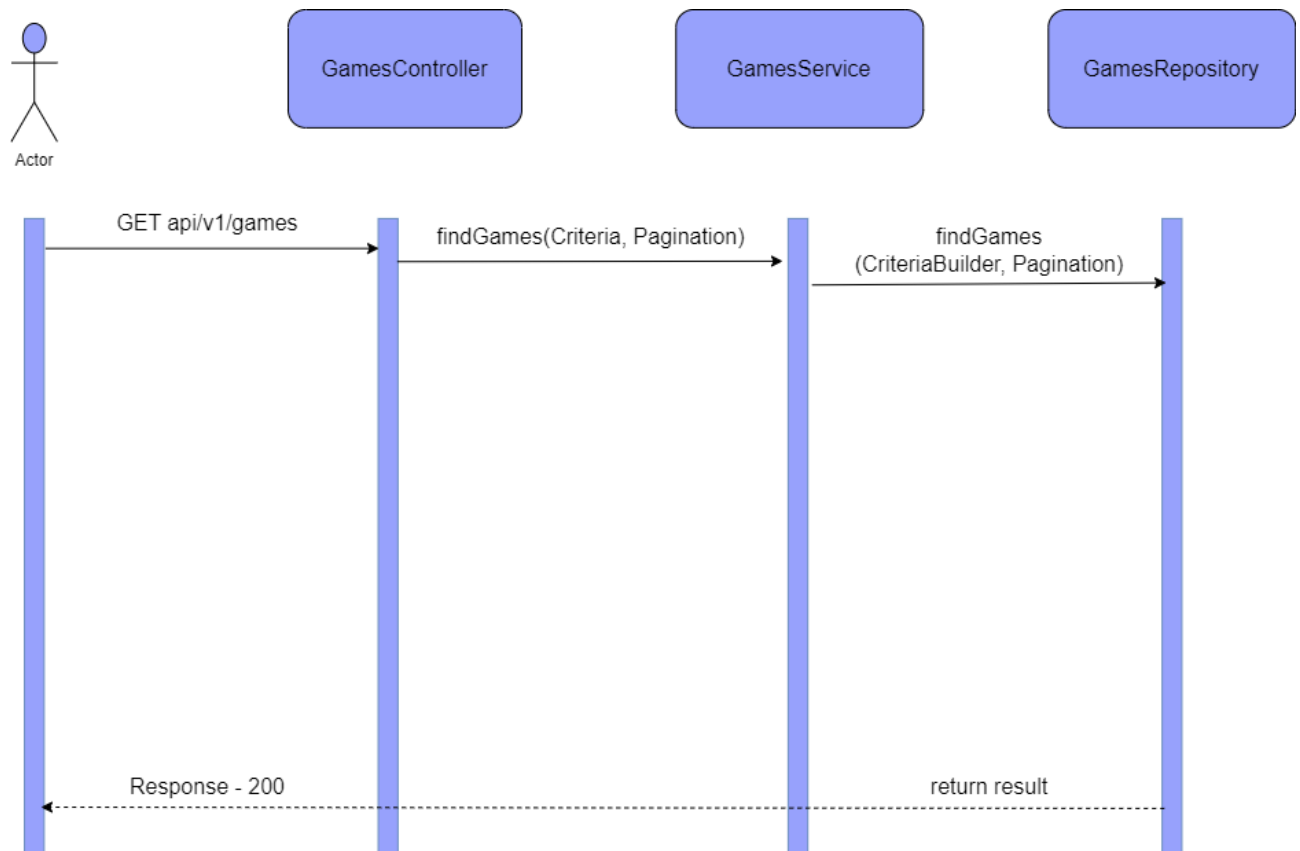
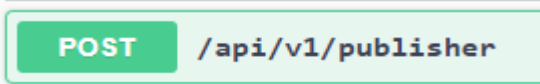
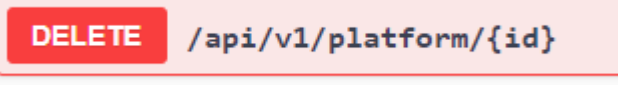




Figure 6: Games search engine that transform request parameters to criteria to query the DB.



This service also allows to create developers, publishers, genres, and platforms related to games although genres and platforms can't be added by common users and in the final app there are restrictions on the number of publishers and developers they can create.

Below some of the endpoints related to the four entities just mentioned:

HTTP METHOD + API	NOTES
 A green button with the text "POST" followed by the URL "/api/v1/publisher" in a light green box.	API to create a new publisher via request body.
 A red button with the text "DELETE" followed by the URL "/api/v1/platform/{id}" in a light red box.	API to delete a platform by an existing id.
 A teal button with the text "PATCH" followed by the URL "/api/v1/genre/{id}" in a light teal box.	API to update a genre name found via id.
 A blue button with the text "GET" followed by the URL "/api/v1/developer/{id}" in a light blue box.	API to get a developer by its id.

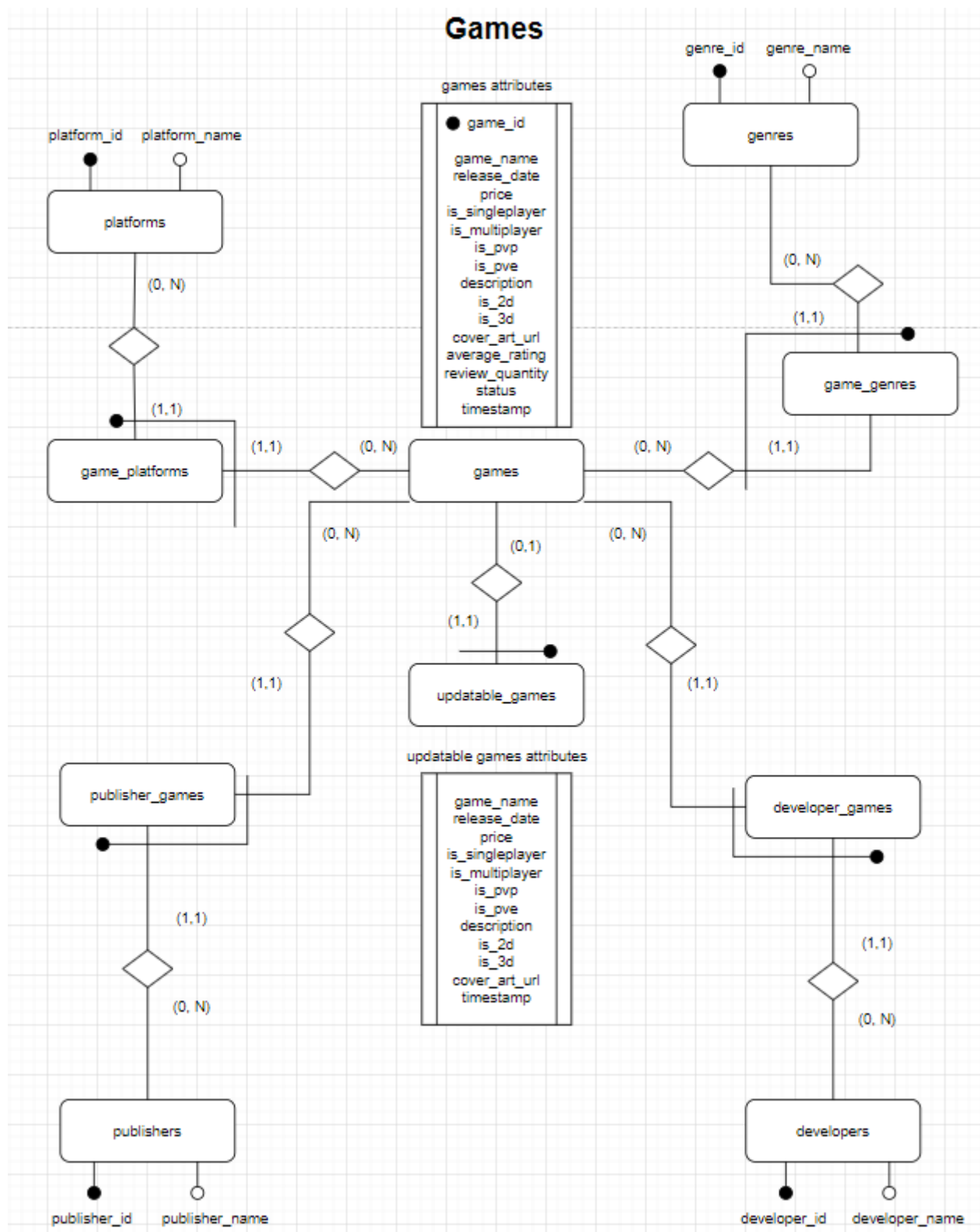


Figure 7: Games DB architecture

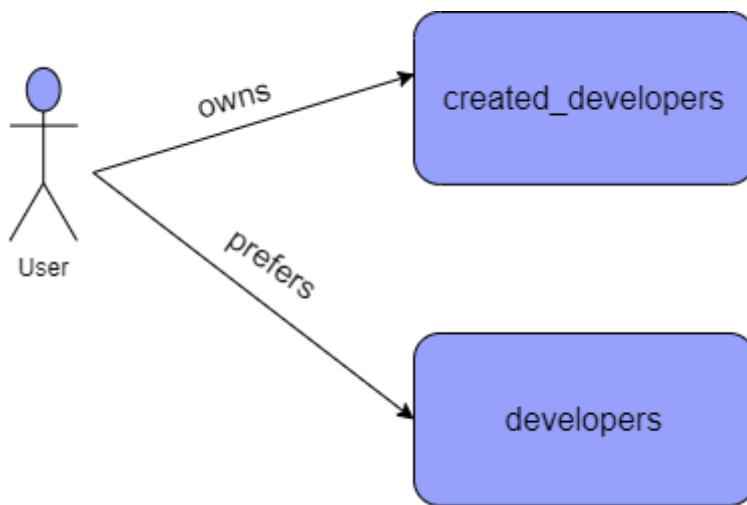
### 4.3 Preferences Service

This service is responsible of storing users' preferences such as his favourite developers, genres, platforms, publishers, and many games characteristics the user may like such as 2d, 3d etc... Other than that, this service stores user created games, wish listed games, games on library and user owned developers and publishers.

The Preferences Service, on its own, might not seem significant. However, when integrated with other microservices, it becomes the primary keeper of user data, storing information that can be beneficial for other services, for example a game discovery feature based on user preferences.

To distinguish user preferred data and user created data a simple rule is applied:

1. resources that have a **'created'** tag indicate a resource that is owned by the user, all the others are user preferred resources as shown in the simplified diagram below.



On the table below I show some of the API endpoints exposed by the service:

HTTP METHOD + API	NOTES
<b>POST</b> <code>/api/v1/preference</code>	API to create a new user preference entry via request body. The json fields are completely optional, every field that is not inserted will be created with default values.
<b>PUT</b> <code>/api/v1/preference/{id}</code>	API to update an existing user preference via request body and id. All the parameters that are not inserted will not have any change. Inserting an empty list means massive deletion of occurrences.

<b>GET</b> <code>/api/v1/created/publishers/{publisherId}/user/{userId}</code>	API to get user preferred publisher by publisherId and user id.
<b>DELETE</b> <code>/api/v1/created/game/{gameId}/user/{userId}</code>	API to delete an entry from user created games by user id and game id.
<b>GET</b> <code>/api/v1/wishlist/games/user/{userId}</code>	API to get all the wish listed games of a user by userId.
<b>DELETE</b> <code>/api/v1/publishers/user/{userId}</code>	API to delete all the entries from users preferred publishers of a certain user via user id.
<b>POST</b> <code>/api/v1/created/publishers</code>	API to create a user created publisher entry via request body.

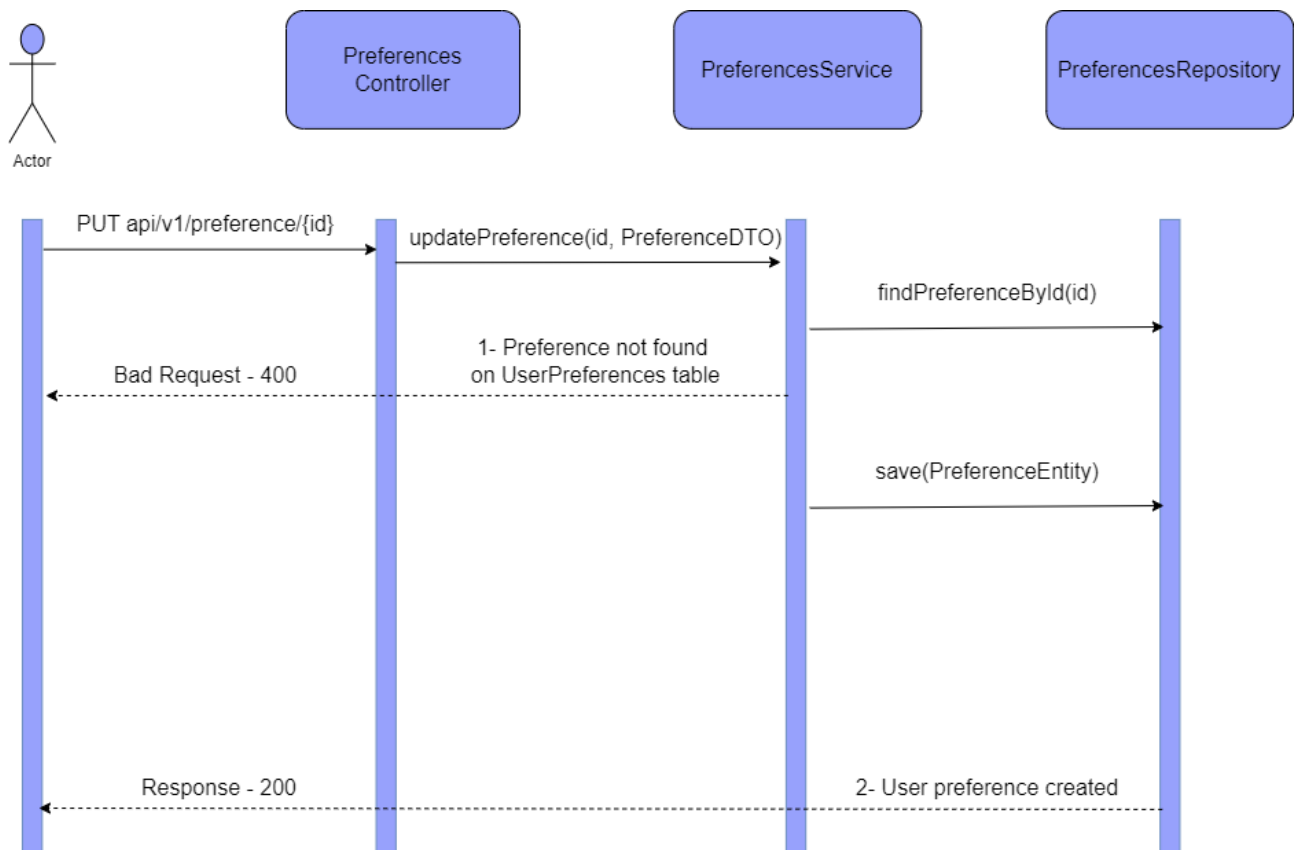


Figure 8: Update of existing preference diagram. Notice that is not 'update(PreferenceEntity)' since save(...) works for both creation and update on the repository.

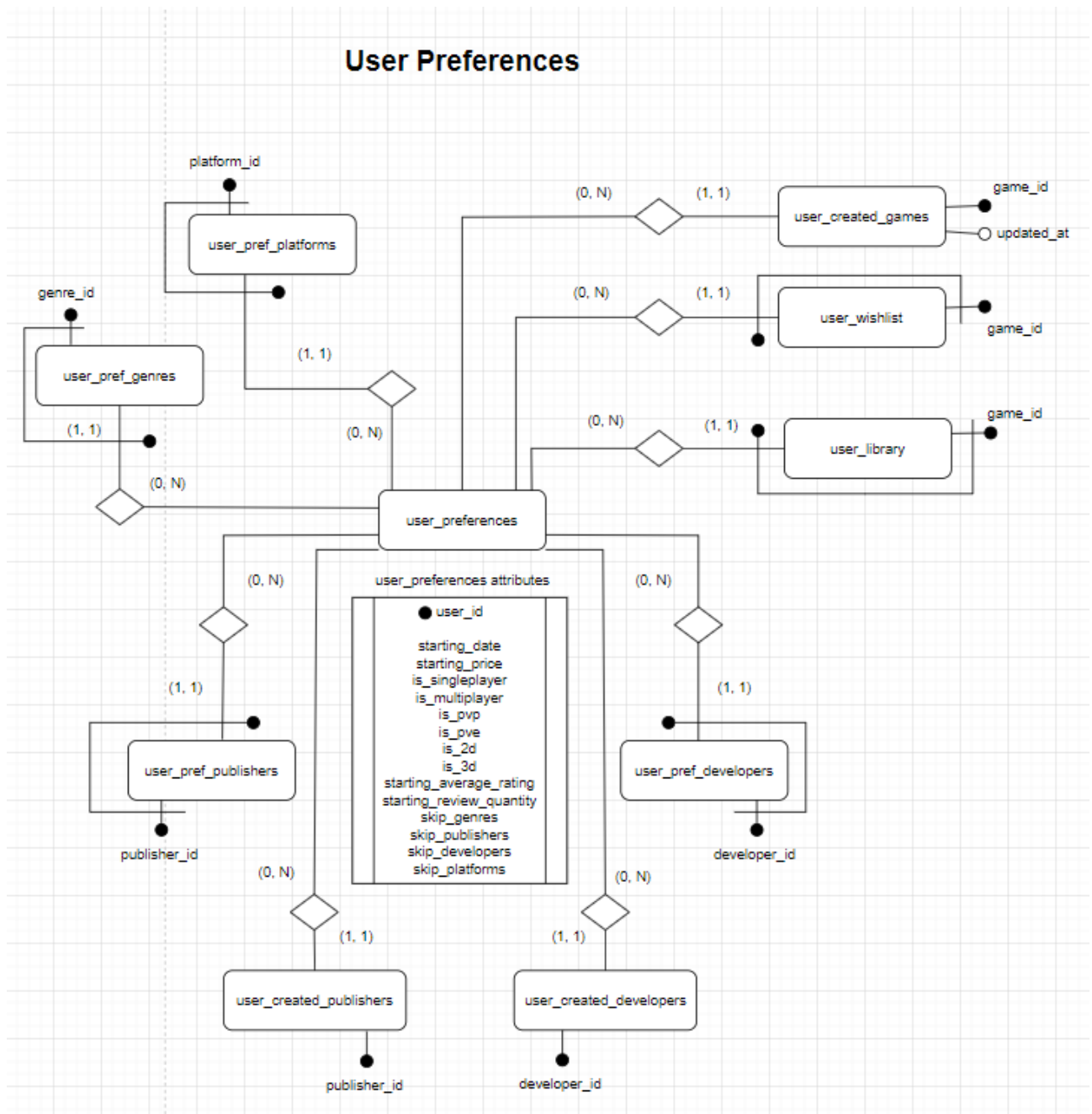


Figure 9: Preferences DB architecture.

## 4.4 Reviews Service

The reviews service has one responsibility: the management of game reviews. In fact, users can get, create, update, and delete reviews, even for games they do not own.

The reviews are composed of:

- Vote.
- Description.
- Game id.
- User id.

The service, with this data, can calculate the exact number of reviews and the average rating for a particular game. This ephemeral data is crucial for the game search engine and should be used to update the game service tables to ensure accurate search results.

Below I show some of the APIs exposed by the reviews service:

HTTP METHOD + API	NOTES
<b>POST</b> <code>/api/v1/review/game</code>	API to create a game review via request body.
<b>GET</b> <code>/api/v1/review/game/{gameId}/count</code>	API to get the review count of a certain game by its id.
<b>GET</b> <code>/api/v1/review/game/{gameId}/average-rating</code>	API to get the average rating of a game by its id.
<b>DELETE</b> <code>/api/v1/review/games/user/{userId}</code>	API to delete user reviews by user id.

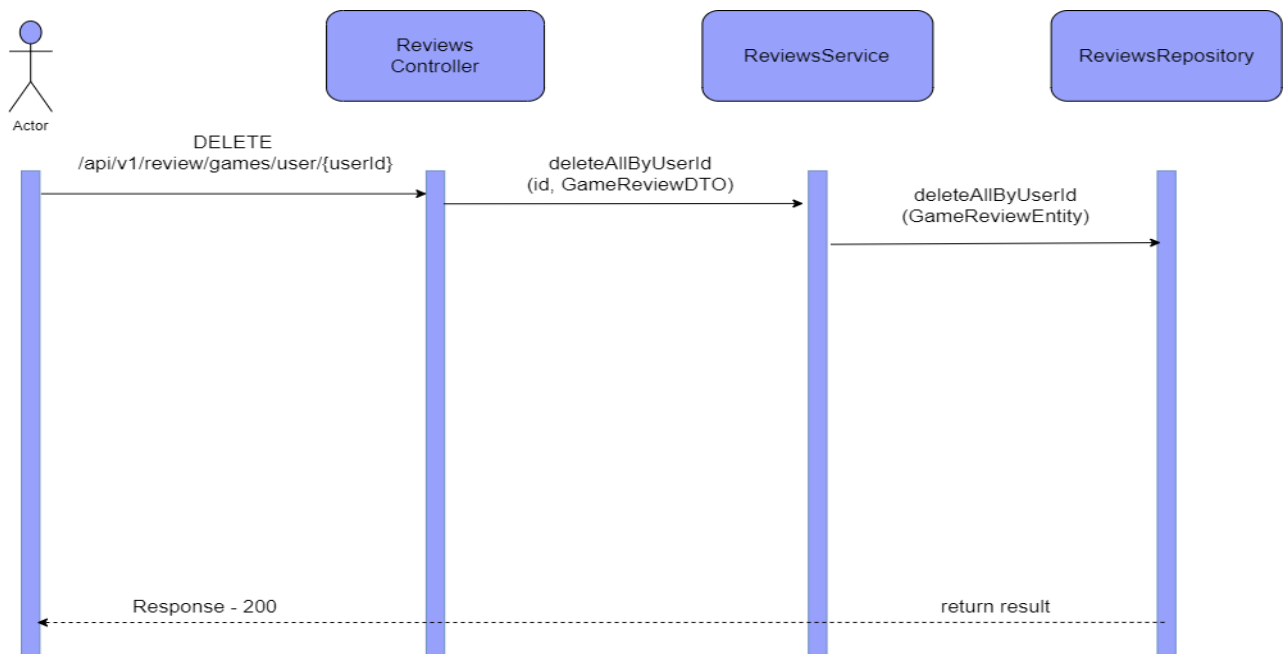


Figure 10: Diagram showing the deletion of game reviews of a certain user by its id.

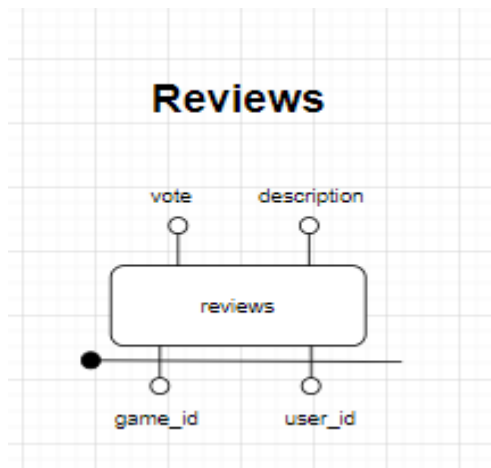


Figure 11: Reviews DB architecture.

## 5. Gateway API

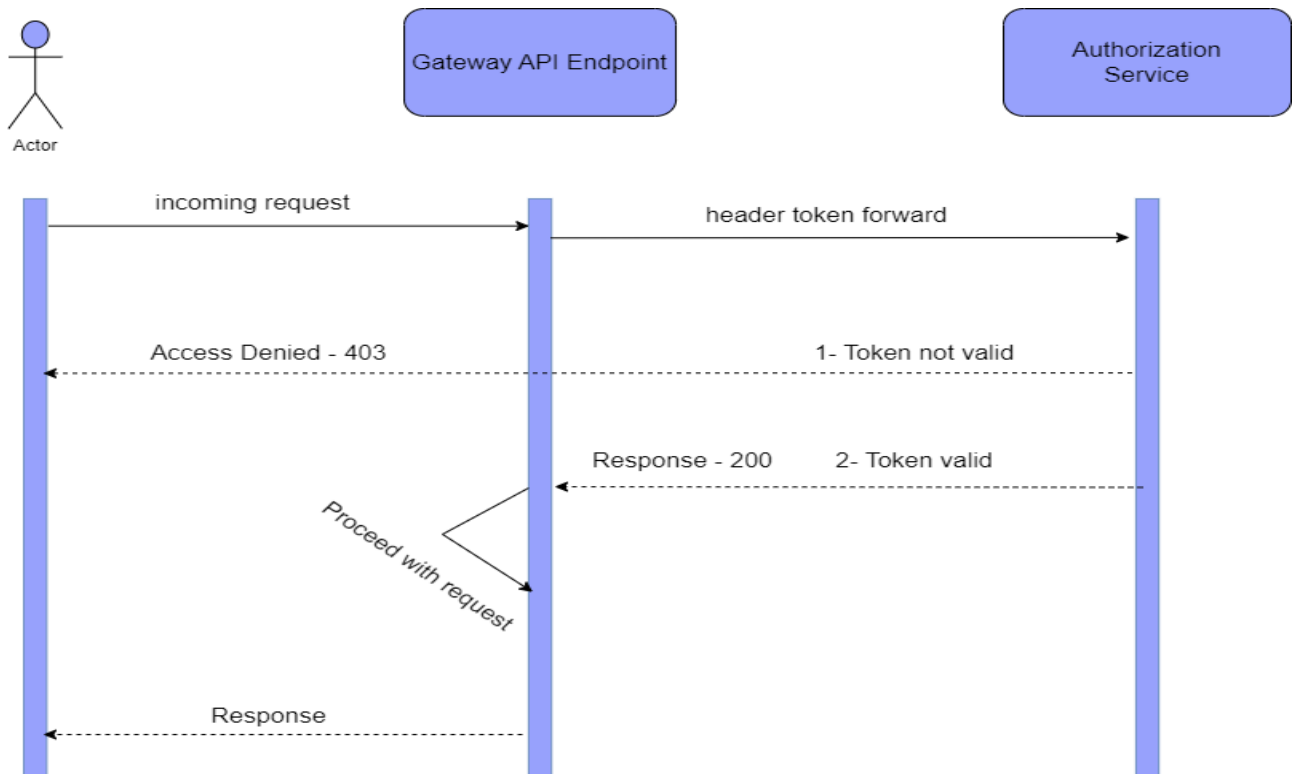
The Gateway is a RESTful API that uses Spring WebClient to communicate with each service in the underlying layer. Spring WebClient is a non-blocking, reactive web client for performing HTTP requests. However, in this case, WebClient is used in a blocking manner because it needs to wait for service requests to be processed before making subsequent requests with the processed data.

This is the main access point of the entire application that uses Spring Boot and integrates Swagger to provide comprehensive documentation and an interactive interface for all externally accessible REST API endpoints.

It has two main responsibilities:

1. Intercept client's requests and forward them to the correct service.
2. Ensure consistency of lower layers (backend, persistence layer).

The Gateway acts as a secure entry point for the application but cannot ensure by itself if a JWT token is valid or not, so it intercepts the request's token and forwards it to the Authorization Service for validation. Once the token is validated all the subsequent requests will not require further authentication during that session.



The only whitelist endpoints are the ones required for accessing the gateway from the browser and the ones that lead to token generation via the Authorization Service.

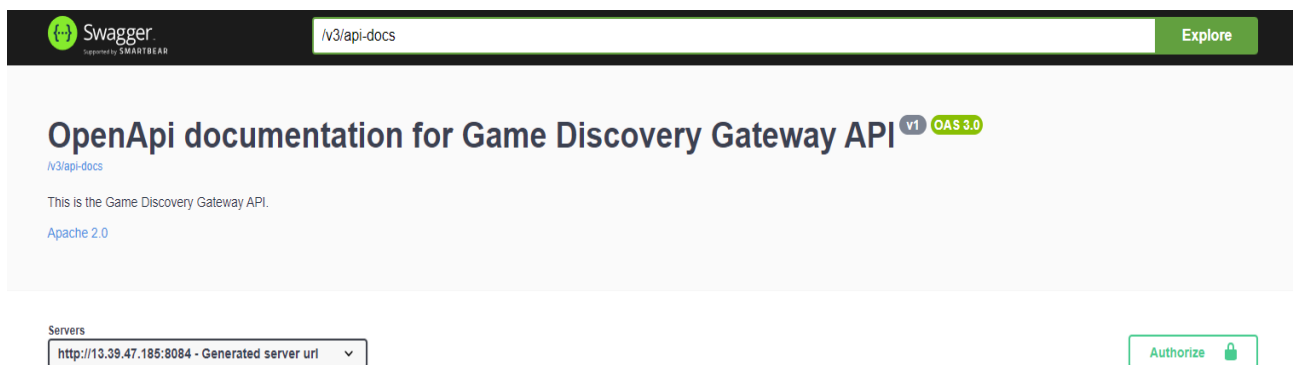


Figure 12: Gateway API via Swagger



## Available authorizations

x

bearerAuth (http, Bearer)

Value:

Authorize

Close

Figure 13: Interactable authorization scheme, where a user can insert a valid JWT token to be able to access secure endpoints.

Admin		^
PUT	/api/v1/by-admin/game/{id}	🔒 ✓
POST	/api/v1/by-admin/publisher	🔒 ✓
POST	/api/v1/by-admin/developer	🔒 ✓
PATCH	/api/v1/by-admin/publisher/{id}	🔒 ✓
PATCH	/api/v1/by-admin/games/synch-with-reviews	🔒 ✓
PATCH	/api/v1/by-admin/games/synch-with-reviews/all	🔒 ✓
PATCH	/api/v1/by-admin/games/status	🔒 ✓
PATCH	/api/v1/by-admin/developer/{id}	🔒 ✓
GET	/api/v1/by-admin/games/with-pending-status	🔒 ✓
DELETE	/api/v1/by-admin/games	🔒 ✓

Figure 14: Administrator privileged section.

The gateway must ensure consistency at lower layers especially in crucial tasks such as game creation and user deletion.

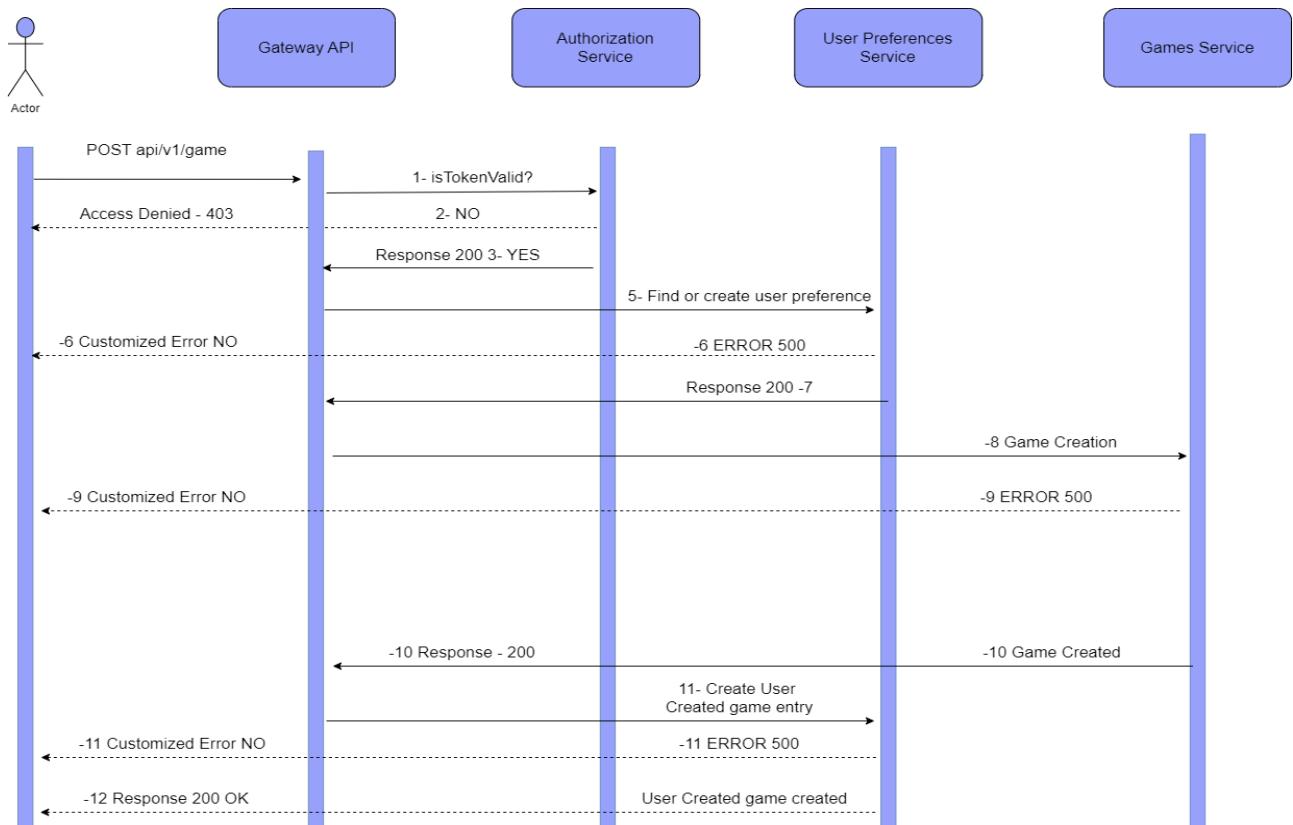


Figure 15: Game creation process from Gateway.

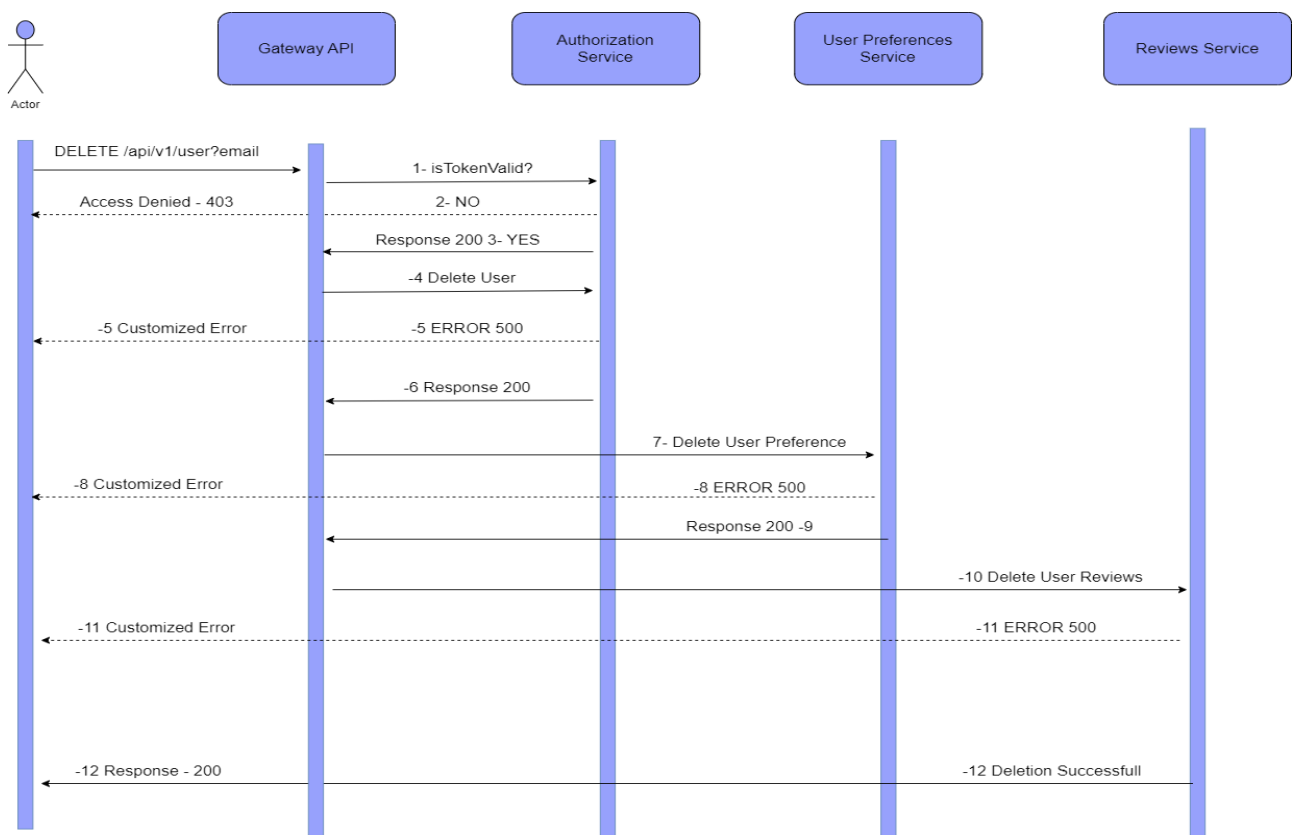


Figure 16: User deletion process by Gateway.

It is also responsible for updating the review count and average rating in the games service when reviews are added, deleted, or updated. The gateway API utilizes asynchronous tasks, coordinated with an internal concurrent map and a Redis cache to limit write operations to the games database.

- Concurrent map
  - Key: game id
  - Value: latest review request made for that game id
  - An asynchronous task runs every 5 minutes updating the oldest key / values.
- Redis
  - Used in the scenario of multiple container nodes.
  - Reduces the amount of duplicated asynchronous calls (eg. duplicated calls to delete rejected game ids within multiple containers)
  - Helps an asynchronous task that runs every 60 minutes to synchronize games database with the actual average rating and reviews quantity for the keys that missed the update until that point.
  - Every 3 hours helps asynchronous task
- As a last resource if everything else fails the synchronization of this type of data can be scheduled by administrators via specific API endpoints.

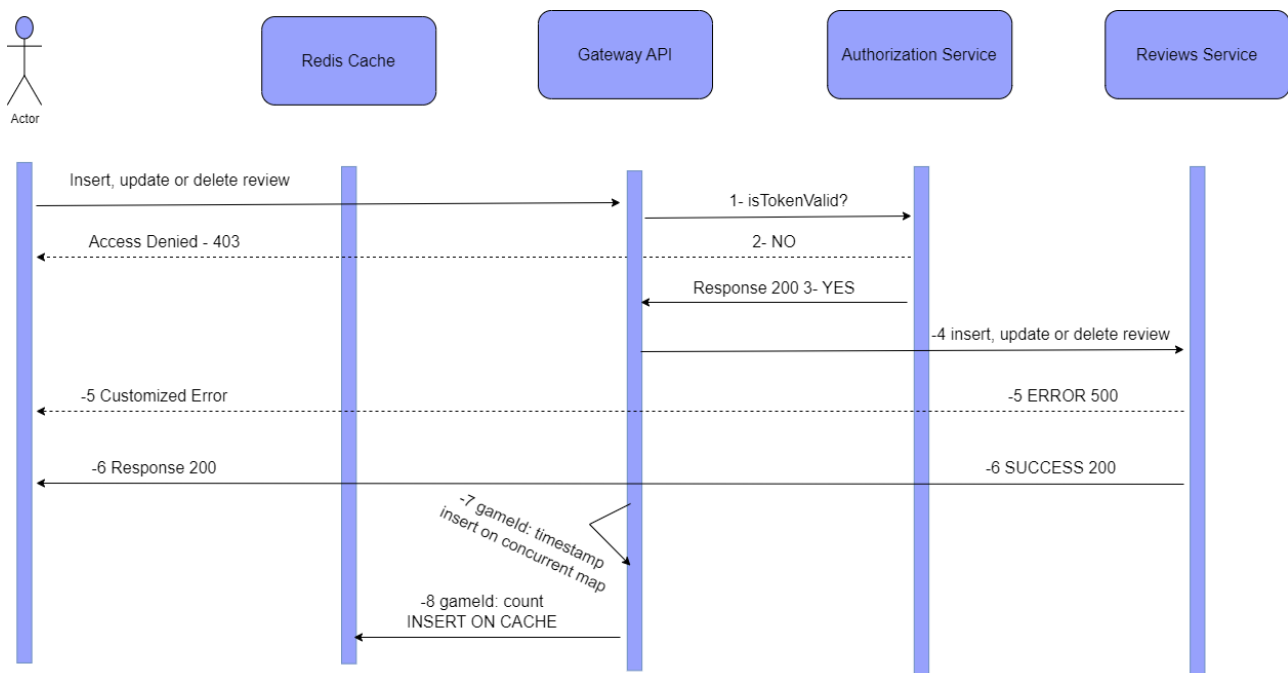


Figure 17: Review request to Gateway updating the internal Concurrent map and the Redis cache.

Redis cache is a powerful addition that reduces the number of read operations at DB level, ensures asynchronous tasks run only once per time even when multiple containers are active and adds the possibility of maximising the consistency between microservices.

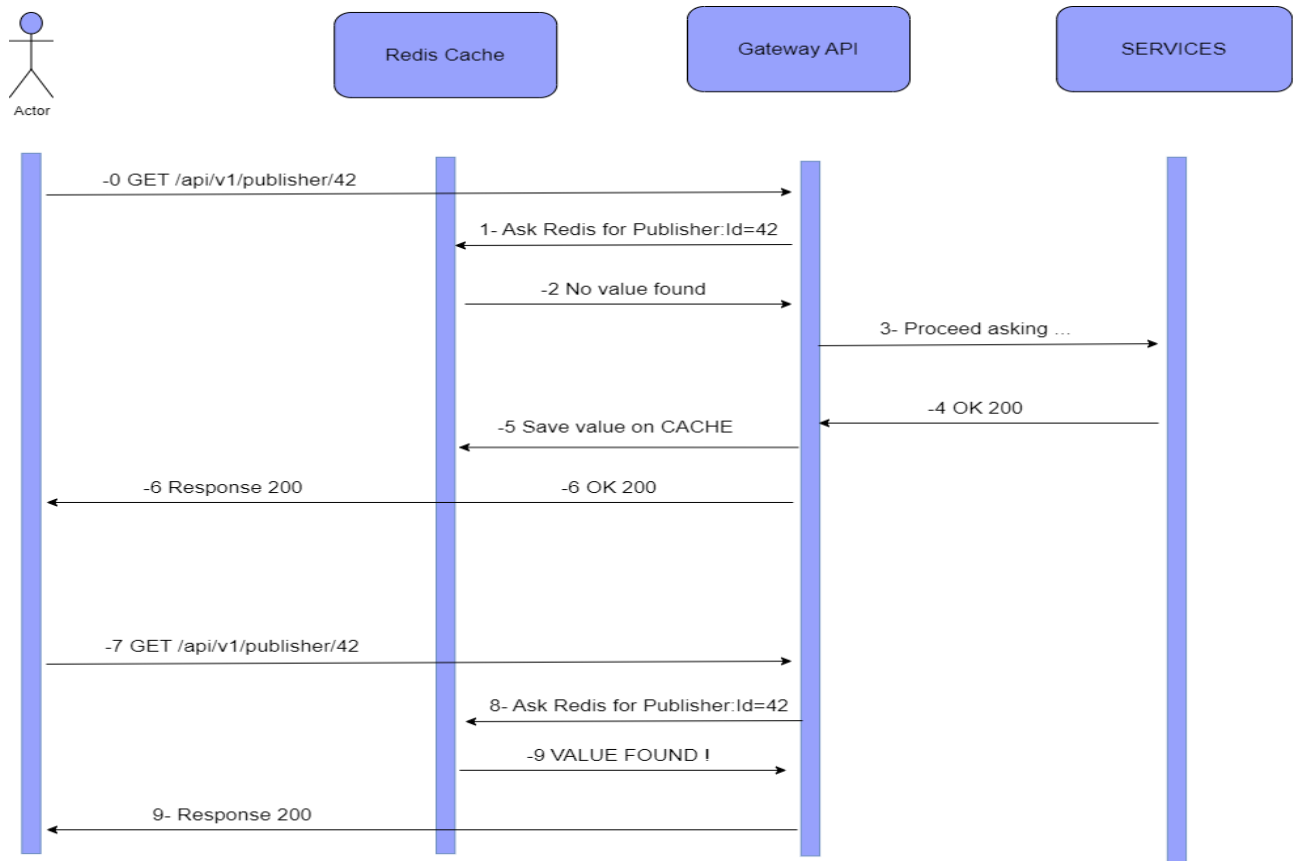


Figure 18: GET request of a publisher. Cache first doesn't have the value, so it is found by asking the backend layer and the key|value is inserted into the cache. The second time the request is performed it is found in the cache and immediately retrieved.

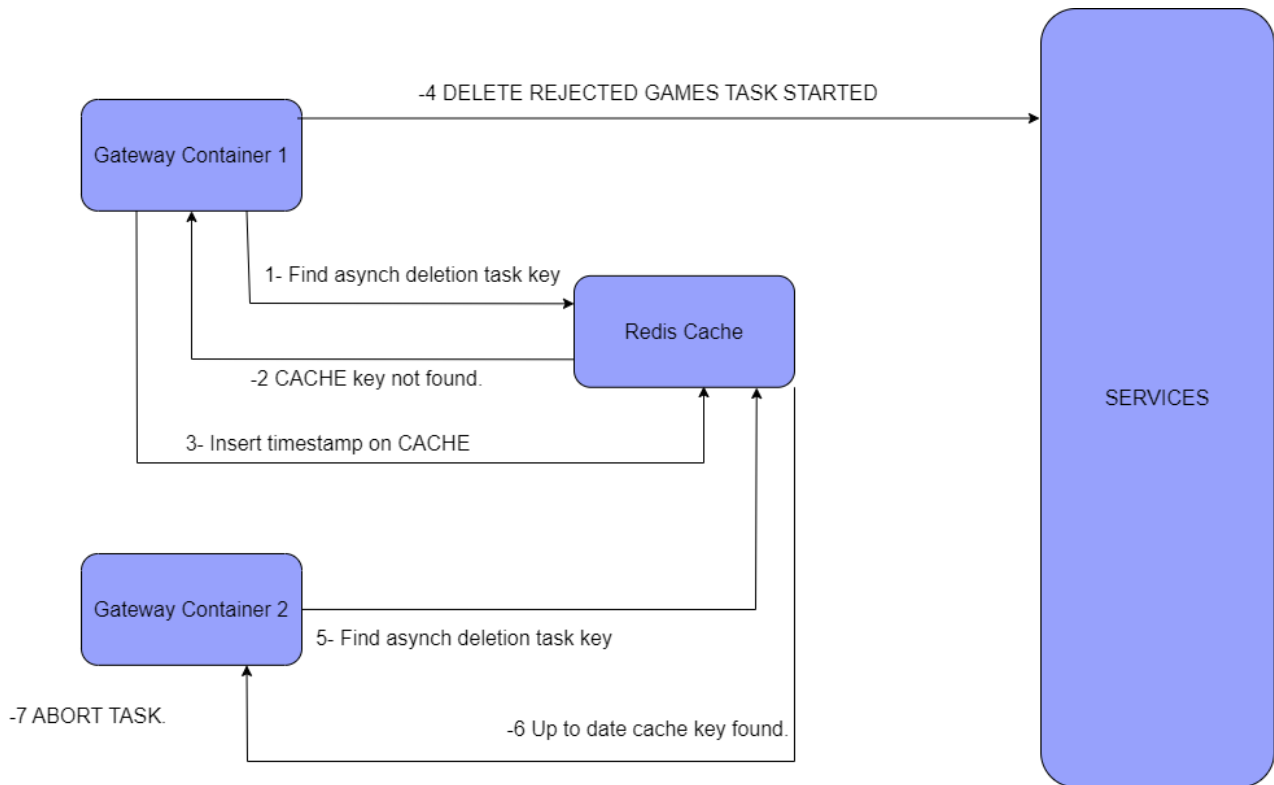


Figure 19: Scenario where two Gateway container want to start an asynchronous task for deleting rejected games. The Redis Cache is active so the first container will start the task and update Redis cache. The second one later will abort the task since it finds an up to date task key already present on the cache.

Below I show some of the APIs exposed by the Gateway API:

HTTP METHOD + API	NOTES
<h3>Authorization</h3> <div> <div>POST /api/v1/auth/sign-up</div> <div>POST /api/v1/auth/authenticate</div> </div>	Authorization Controller with endpoints to authenticate via the Authorization Service.
<h3>Games Discovery</h3> <div>GET /api/v1/discovery</div>	API to find games via user owned preferences such as owned games, user preferred genres and so on... and tuneable randomness that ranges from [0 to 1].

<div>PUT</div> <div>/api/v1/by-admin/game/{id}</div>	API to update a game. Only admins can update games that they approved.
<div>PATCH</div> <div>/api/v1/by-admin/games/synch-with-reviews/all</div>	API to synchronize games service average rating and review quantity via reviews service. This is a massive operation that can be used only by administrators.
<div>GET</div> <div>/api/v1/by-admin/games/with-pending-status</div>	API to get games 10 by 10 via pagination that are on pending status
<div>POST</div> <div>/api/v1/review/game</div>	API to create a review via the Reviews Service
<div>GET</div> <div>/api/v1/review/game/user/{userId}</div>	API to get all the reviews for a certain user by its id.

### About Game Discovery

Game discovery was initially the primary feature of the project, but it was the last to be developed. It relies on a tuneable parameter ranging from 0 to 1, where a value closer to 1 bases the search entirely on user preferences. The search aims to find at least 10 games for the user, making multiple requests if necessary to meet this number. The search mechanism is based on a point system, where higher points for certain user preferences increase their likelihood of being included as request parameters in the 'GET api/v1/games/by-discovery' endpoint. The more games a user has in their created games table, library table, and wish list table, the more parameters are available for the search.

## 6. Aws Deployment

The architecture used for the AWS deployment is the following:

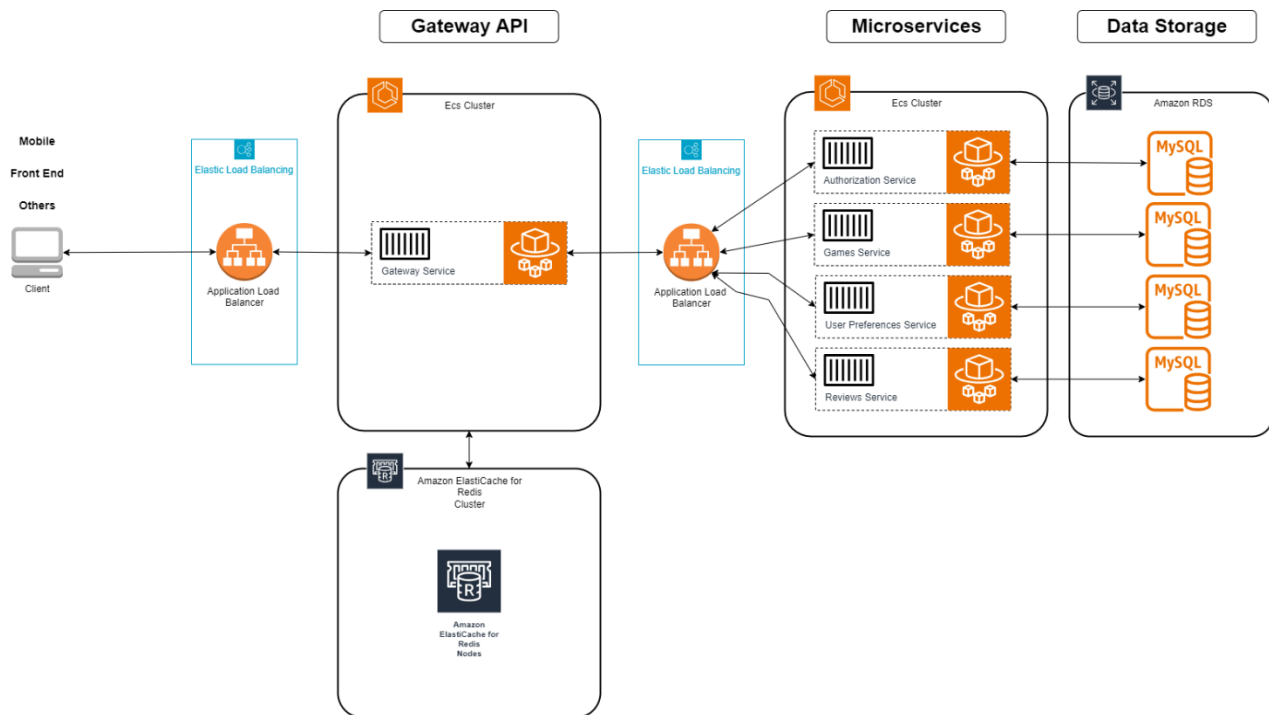


Figure 20: AWS deployment architecture diagram.

For each microservice one docker image has been created and all of them were pushed on AWS:ECR. This makes the images easily available for ECS clusters.

On ECS has been defined one cluster with 4 services, one per each microservice and then another cluster for the GatewayAPI. All the services use Fargate as launch type. Fargate is a technology that avoids configuring and selecting EC2 instances to run containers. Furthermore, Fargate, gives complete isolation for what concerns CPU, Memory, and resources usage.

For each service (Gateway included) was created a target group and listeners on respective ports.

- Gateway: 8084
- Reviews: 8083
- Preferences: 8082
- Games: 8081
- Users: 8080

There are two application load balancers (ALB). The first one accessible publicly is placed in front of the Gateway and listens incoming requests on standard HTTP port 80 redirecting them to the gateway service tasks.

The second load balancer is an internal one, not accessible from the outside and can only accept incoming requests from the gateway security group and forward them to the microservices behind it via target groups and listeners mentioned before.

Notice that all the microservices are not accessible via swagger because they should only communicate with the load balancer in front of them. Meanwhile the Gateway is still accessible via browser publicly.

Redis, that was an important part of the local environment is now used in AWS (small type instance) but this time via a Redis Cluster (Amazon ElastiCache) that has a maximum of 3 replicas running. This makes Redis Cache highly available in case of failures. For enhanced security Redis Cluster will communicate exclusively with Gateway cluster and will not be accessible from the outside.

Scale-in/out

For each ECS service (gateway included) has been defined an autoscaling target tracking policy as in the following picture:

Parametro del servizio ECS

ECSServiceAverageCPUUtilization

Valore di destinazione

15

Tempo di raffreddamento aumento orizzontale

300

Tempo di raffreddamento riduzione orizzontale

300

☐ Disattiva la riduzione orizzontale

Figure 21: the ECS service metric is the CPU utilization.

All the services have an established 15% target for CPU Utilization metric:

- 1. When below 13.5% threshold for 15 datapoints within 15 minutes a scale-in policy will be triggered decreasing the number of running tasks.
- 2. When higher than 15% threshold for 3 datapoints within 3 minutes a scale-out policy will be triggered increasing the number of running tasks.

The target was chosen to be as small as possible to ensure that scale in-out policies would be triggered during the final tests. Higher values would require a large number of concurrent users, which could lead to frequent test failures and increased costs for running the tests.

Here I show the JSON alarm tracking rules generated by CloudWatch common for all the services (gateway included).

<pre>{   ...   "MetricName": "CPUUtilization",   "Namespace": "AWS/ECS",   "Statistic": "Average",   "Dimensions": [     ...   ] }</pre>	<p>This part corresponds to the scale-out policy alarm when the CPU Utilization goes below threshold.</p> <p>Under “Dimensions” field there are the ClusterName and ServiceName properties that not shown here because are different for each service.</p>
--	--



<pre>     ],     "Period": 60,     "Unit": "Percent",     "EvaluationPeriods": 15,     "DatapointsToAlarm": 15,     "Threshold": 13.5,     "ComparisonOperator":     "LessThanThreshold"   } } </pre>	
<pre> {   ...   "MetricName": "CPUUtilization",   "Namespace": "AWS/ECS",   "Statistic": "Average",   "Dimensions": [     ...   ],   "Period": 60,   "Unit": "Percent",   "EvaluationPeriods": 3,   "DatapointsToAlarm": 3,   "Threshold": 15,   "ComparisonOperator":   "GreaterThanThreshold" } } </pre>	<p>This part corresponds to the scale-out policy alarm when the CPU Utilization goes higher than threshold.</p> <p>Under “Dimensions” field there are the ClusterName and ServiceName properties that not shown here because are different for each service.</p>

Amazon RDS has been used to create four MySQL databases that can only communicate with the respective services.

To access the DBs I m using an EC2 instance where I saved initial scripts to populate them. This is crucial because AWS deletes the data every 7 days without backups... and backups cost money while an EC2 instance turned off doesn't.

AWS Cloudwatch has been used to extract metrics and monitor the application under stress tests conditions.

## 7. Tests

To ensure the microservices architecture is robust and scalable, it is crucial that each service can independently scale based on its load.

For effective testing Locust was utilized. It is a powerful Python testing tool that can simulate concurrent users performing various requests (tasks) at different intervals.

To monitor the system, AWS CloudWatch was employed, focusing on CPU Utilization and the number of running tasks. Alarms generated for each service were also monitored in CloudWatch and were instrumental in testing the independent scaling capabilities of the microservices, both scaling in and scaling out.

All the tests made wanted to stress out the whole system, in particular the generated users will be divided into normal users and admin users and will call in order the REST APIs to:

1. Authenticate themselves (ONLY once) to get a JWT token (USER or ADMIN)
2. Submit new games. (USER or ADMIN)
  - a. This also will make the user find and sometimes create a preference for the current user. It will also make a new entry on 'created\_games' table.
3. Get 10 pending games and randomly approve or reject them (ADMIN).
  - a. This is super important in this application since users can't submit more than 10 games every 24 hours if nobody approves or rejects their games.
  - b. Games that are rejected are deleted every 3 hours so won't affect the tests.
4. Create or update a review for one of the first 20 games that are on Database. (USER or ADMIN)
  - a. First get the user id then choose a game randomly and find out if a review exists for it. Finally update or create it if doesn't exist yet.
5. Update a preference or create a new one. (USER or ADMIN)
  - a. This consists in finding the user id and then creating or updating the preferences if it already exists.
6. Game Discovery (USER or ADMIN)
  - a. This is a more sophisticated search that surely can consume a lot of CPU for game discovery.

Notice that once arrived at point 6 users will start again from point 2 in a perpetual loop until time ends.

Two tests were conducted, both lasting 30 minutes.

## Test 1

The first round of tests was conducted by simulating 30 users for the initial 10 minutes, then increasing the number to 60 users for the next 10 minutes, and finally decreasing it to 20 users for the last 10.



Figure 22: Locust - TEST 1

During the test, three alarms were triggered. The first services that had to scale up by increasing the number of running tasks to stay near the target were the games service and the gateway. The games service is the most stressed in this scenario, as not only it is frequently called by users (points 2 and 6) but needs to perform many heavy READ operations and has to get updates for any addition or update to the reviews service. The gateway is also significantly affected since it is the primary access point and handles various asynchronous tasks in the background, so it is expected to get this result. The other service that is always called is the authorization service with many READ operations made to get the users IDs. Has shown in the following diagram it was most stressed during the addition of users to the test meaning that generating tokens for all of them is probably the cause of this increase in CPU Utilization.

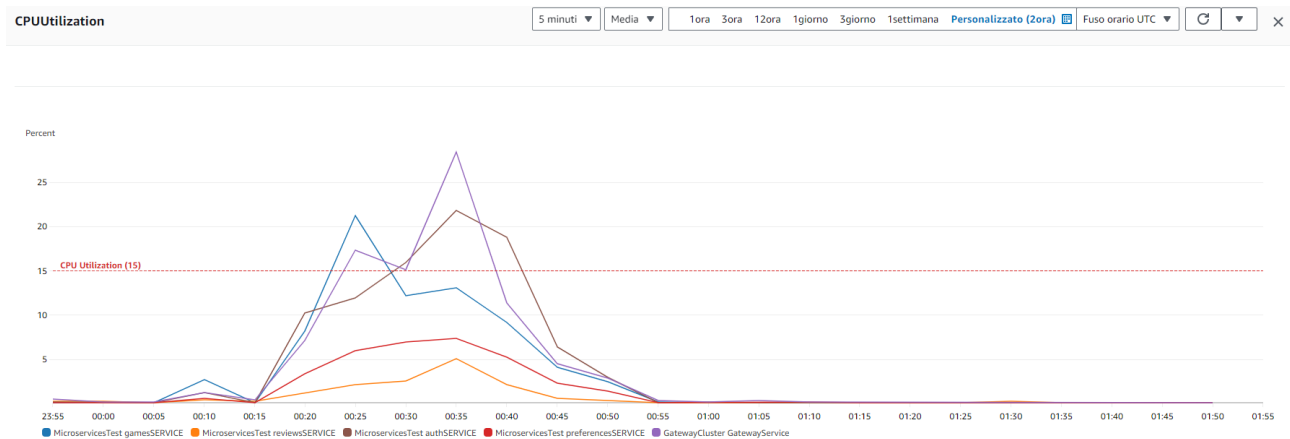


Figure 23: CloudWatch CPU Utilization - TEST 1. Time start around 00:15, time end around 00:45

On the other hand, reviews and preferences didn't get a CPU Utilization value higher than the threshold meaning that the requests performed weren't as impacting, although the number of requests these services got was on pair with the other ones. As expected no new running tasks were then created for these services.



Figure 24: CloudWatch RunningTaskCount - TEST 1.



Figure 25: CloudWatch RequestCountPerTarget - TEST 1

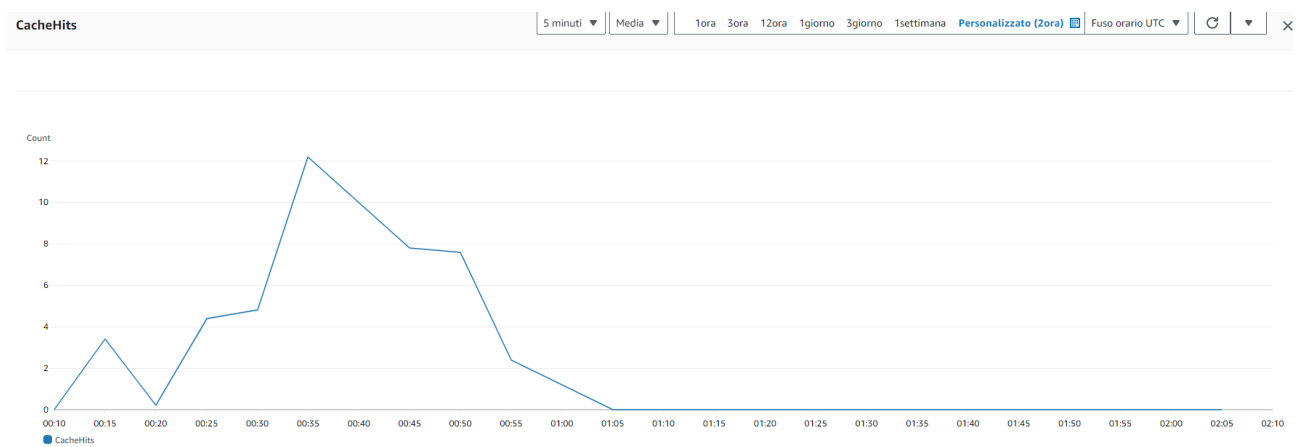


Figure 26: CloudWatch Cache Hits - TEST 1

## Test 2

The second round of tests was conducted by simulating 30 users for the initial 10 minutes, then increasing the number to 100 users for the next 10 minutes, and finally decreasing it to 10 users for the last 10.



Figure 27: Locust - TEST 2

During this test, three alarms were triggered again. The CPU utilization for the Games and Gateway services was significantly higher. As a result, the autoscaling policy needed to increase the number of running tasks by more than just one to return to standard service levels. In fact, the Games service scaled up to 4 running tasks at once in this case.

Regarding the Preferences and Reviews services, there is not much new to report. Although the Reviews service experienced slightly higher stress than before, it still did not exceed the established target. As expected, no additional running tasks were created for these services.

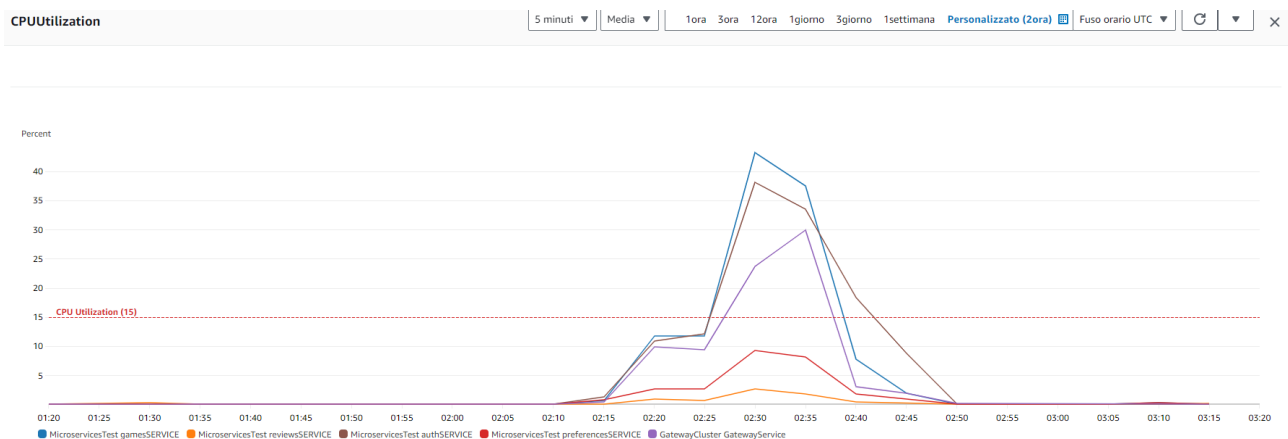


Figure 28: CloudWatch CPU Utilization - TEST 2 Start time around 02:15, end time around 02:45

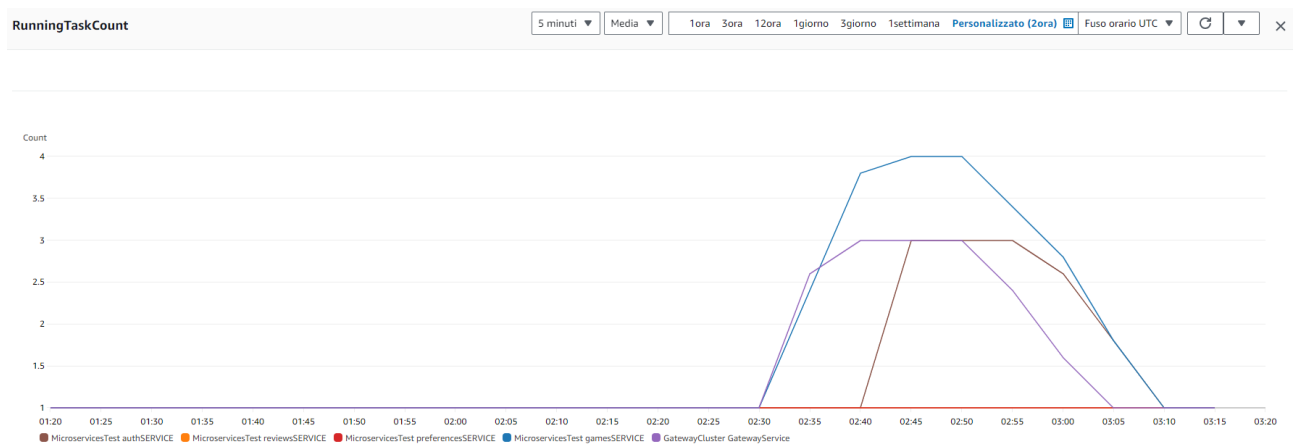


Figure 29: CloudWatch RunningTaskCount - TEST 2

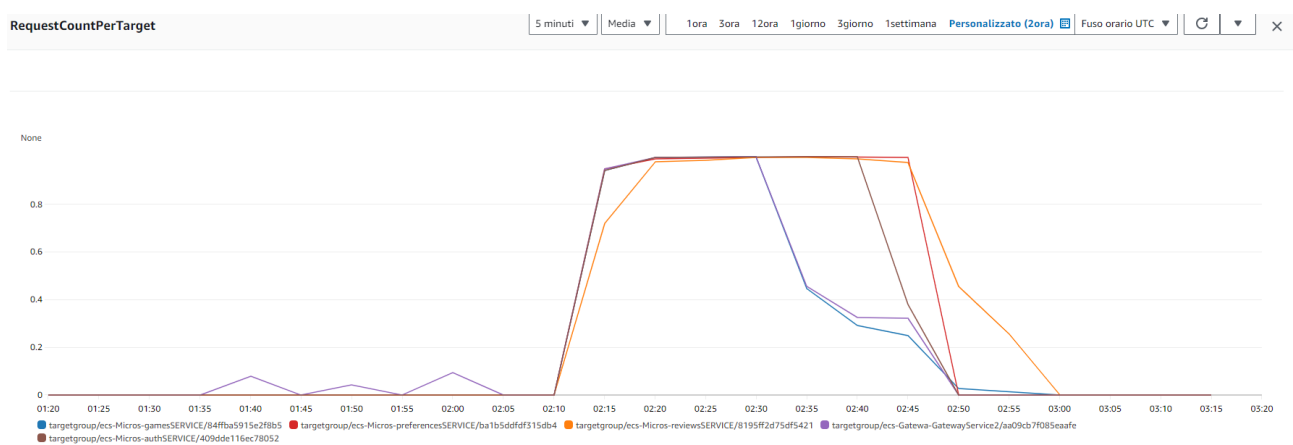


Figure 30: CloudWatch RequestCountPerTarget - TEST 2

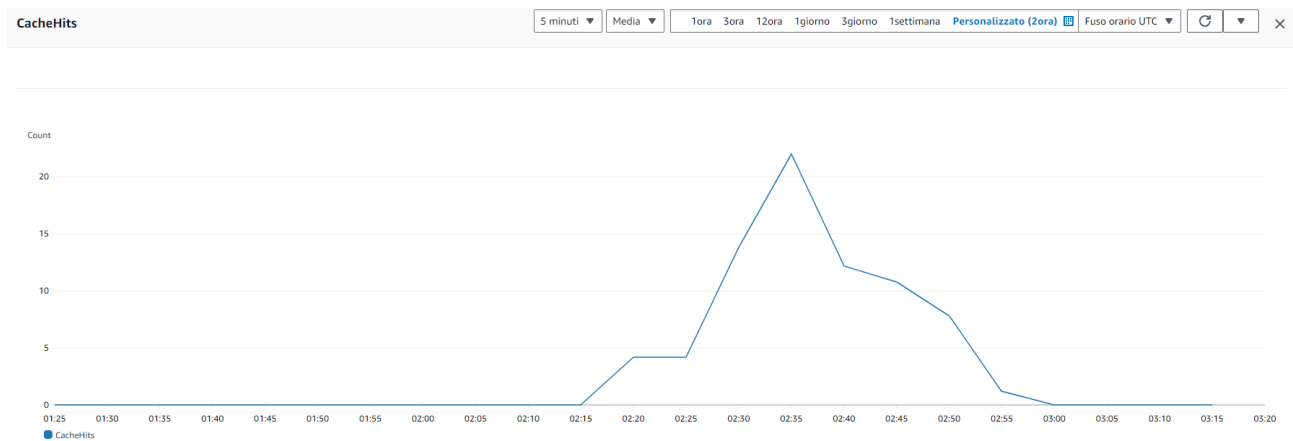


Figure 31: CloudWatch CacheHits - TEST 2

## 8. Conclusions

The experimental results demonstrate that the autoscaling system for the microservices functions as expected: with a low number of concurrent users, the CPU utilization remains low, and no additional tasks are created. On the other hand, as the number of concurrent users increases and more requests are made, the thresholds are exceeded, triggering the system to scale out. This mechanism ensures that each microservice maintains a balanced number of tasks, keeping the services highly available at any time.

This project was really challenging to execute but I'm really satisfied with the results. It increased my awareness on the power of cloud technology in enhancing deployment of software. It also showed me that the scalability and efficiency of microservices can bring significant benefits for web-based applications. Overall, it was a very valuable experience.