Hi, I'm Sean.
Dublin.
This is Cover

Implementing A Globally Accessible Distributed Service

World Anti Doping Agency
Athlete Locator
FINAL REPORT

GROUP 5: MEMBERS

Akash Purushottam Patil (21330950)

Akul Rastogi (20331653)

Chaitanya Vivek Joglekar (21348468)

Xiang Mao (21332237)

Guowen Liu (21332735)

CONTENTS

INTRODUCTION

The aim of this system is to provide a common yet globally accessible platform for global Anti-Doping Organisations (ADOs) and athletes to manage test appointments and availability for anti-doping tests respectively. We recognize two types of users for this system. These are ADO Administrators and Athletes.

ADO Administrators
Each of the 194 countries will have a set of ADO administrators who can view the availability of the athletes of their country only. These administrators can then choose time slots for randomly testing the athletes.

Athletes
Each country will have a set of registered athletes who will provide their availability for one hour every day to their respective ADO authority. It may also happen that the athletes travel to other countries for events and hence report their availability in other countries. In such cases, the ADO of the home country will schedule the athlete's appointments in the destination country and the ADO of the destination country will take care of the appointment.

REQUIREMENTS

FUNCTIONAL

Athlete

1. The recorded availability must contain the details of where they can be found (location) and when they can be found (timestamp)
2. Athletes must be able to select where they can be found from a set of predefined locations (countries) and regions (continents)
3. Athlete must be able to only provide future time slots for their availabilities
4. Athletes cannot modify their availabilities within 2 days (48 hours) of the specified availability time

ADO

1. The ADO must be able to view the list of only those athletes who are registered in their country
2. The ADO user must be able to book tests for athletes only within 48 hours of the specified timestamp. Once booked, the test cannot be rescheduled.
3. The ADO user must be able to see all the tests scheduled in their country (including the ones scheduled by ADOs of other countries)
4. Only the ADO should be able to see the list of appointments, as they are random tests and athlete should not be notified about them

User

1. Athlete/ADO must be able to register with relevant details and onboard onto the system
2. Athlete/ADO must be able to login into the system using their credentials
3. An auth token and corresponding ADO/Athlete ID must be issued in the login response which will be used for subsequent requests
4. The system should validate the token for every incoming request after login

NON FUNCTIONAL

- The system should be highly scalable since under one jurisdiction, there can be millions of athletes adding their availability at the same time.
- System should be highly available to allow athletes to add availability at any time.
- The system can allow weak consistency as Athlete's and ADO's interactions with the system are mutually exclusive.
- For Athletes, the number of writes will be more than the number of reads, hence weak consistency models like "read your writes" are sufficient for supporting the athlete functionality.
- As the ADO will have to get the latest availability status of the corresponding athletes, this operation is causally related to athletes putting their availability. Hence the system should be able to handle this causality and should provide read recency.
- Systems should be tolerant to failures of individual services and failures occurring at sharded database cluster layers to provide high data durability and reliability.

SPECIFICATION

Based on the above defined functional and non-functional requirements, we make design choices that best satisfy the requirements. There are 2 types of users and each user has its own set of mutually exclusive operations. Hence, we can imagine 3 main components of the system:

1. ADO Management
2. Athlete Management
3. User management

Hence, on a high-level, we propose a microservice-based architecture in which we arrange the application as a collection of 3 loosely coupled services namely:

1. **Ado-Service**
   Handles following operations/functionality for ADO user sequentially
   a. Get list of athletes for ADO's country
   b. View availability for a particular athlete
   c. Book Test for a particular availability within 48 hours
   d. View list of scheduled appointments

   API specification for Ado-service:
   - *List<Athlete>* ***getAthletes*** *(String adoID)*
   - *List<Availability>* ***getAvailabilitiesForAthlete*** *(String athleteID)*
   - *Void* ***bookTestForAthlete*** *(String availabilityID)*
   - *List<Availability>* ***getAllAppointments*** *(String adoID)*

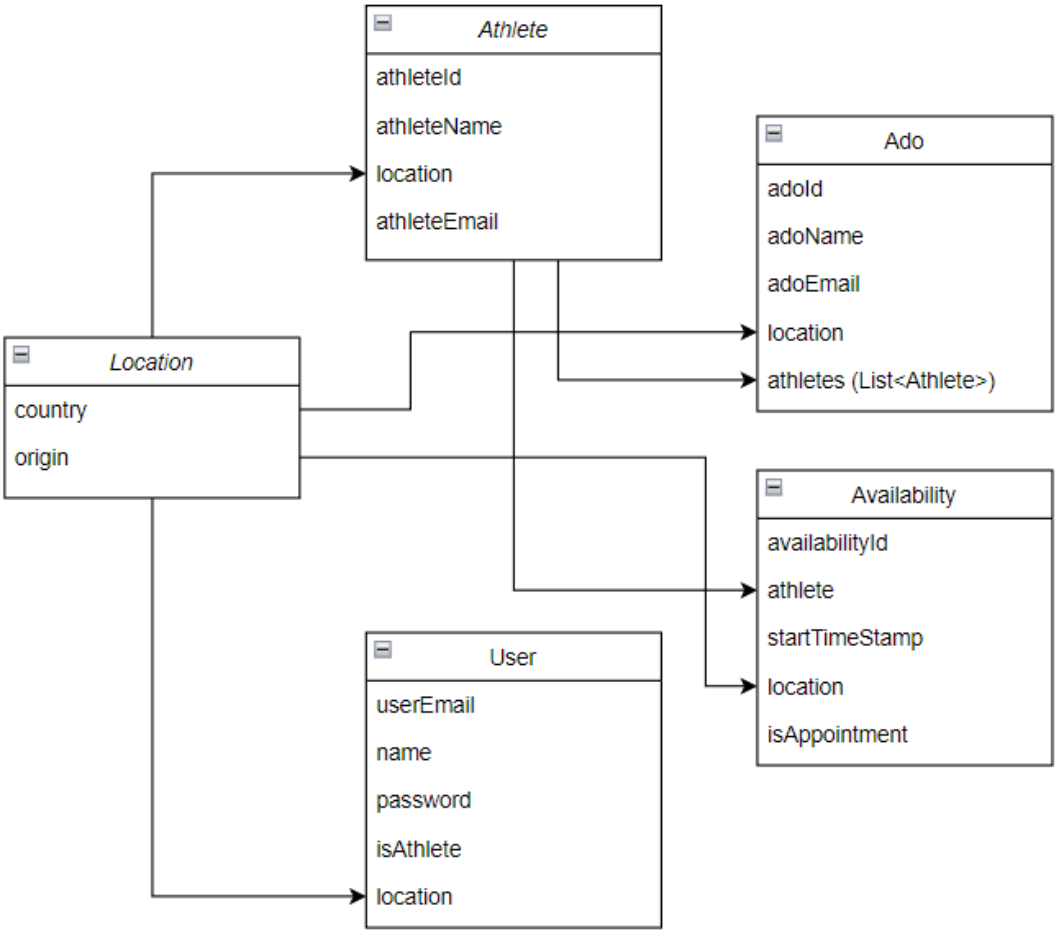


Diagram 1: Different entities in the system

2. **Athlete-Service**
   Handles following operations/functionalities for Athlete user
   a. Add availability for a location and time
   b. Get all existing availabilities
   c. Update availability
   d. Delete Availability

   Possible faults:
   - If add availability fails for an athlete, their corresponding ADOs will not be able to view their availability for anti-doping tests.
   - If an athlete is unable to update their availability as and when desired due to a fault, the ADO may schedule a test when the athlete is unavailable.

   API specification for Athlete-service:
   - *void **addAvailability** (String athleteID, String location, String timeStamp)*
   - *List<Availability> **getAvailabilitiesForAthelte** (String athleteID)*
   - *void **updateAvailability** (String athleteId, Availability availability)*
   - *void **deleteAvailability** (String availabilityID)*

3. **User-Service**
   Handles operations related to administration like
   a. *User login* - Login the user and redirect to Athlete page if isAthlete is true, else redirect to Ado page. Provides user identity details to be used for subsequent requests
   b. *User registration* - Creates user entry as well as Athlete and Ado entry based on isAthlete flag. If isAthlete is false, add entry to ado collection, else add entry to athlete collection
   c. *Get locations* - Fetches all locations stored in the database
   d. *Add locations* - Add new locations if new countries are formed
   e. *Token issuance* - Issue auth token after login for authenticating subsequent requests
   f. *Token validation* - Validate token for each request entering the system

   Possible faults:
   - If an athlete is unable to register into the system, he/she will not be able to add availability and as a result, the ADO will not be able to schedule tests.
   - Failure in token generation service may deprive user from using the system
   - Failure in token validation service may impact on the user login or may induce security vulnerabilities.

   API specification for User-service:
   - *LoginResponse **login**(String email, String password)*
   - *void **register** (UserRegistrationForm form)*
   - *List<Location> **getLocations**()*
   - *void **addLocations** (List<Location> locations)*
   - *void **isValid** (String token)*
   - *String **generateToken** (String username)*

ARCHITECTURE

In this system we have implemented highly distributed, scalable cloud native architecture. The system uses microservice based architecture to modularize the functionality. Following diagram shows the overall architecture of the system implemented with different modules/components involved in it:

Modules/Components within the architecture:

Front End
The dashboard implementation using Ant-Design, React and Express stack. It integrates with the different microservice API in backend developed using Java Spring Boot. It provides different tables for different types of users such as Athlete or ADO.

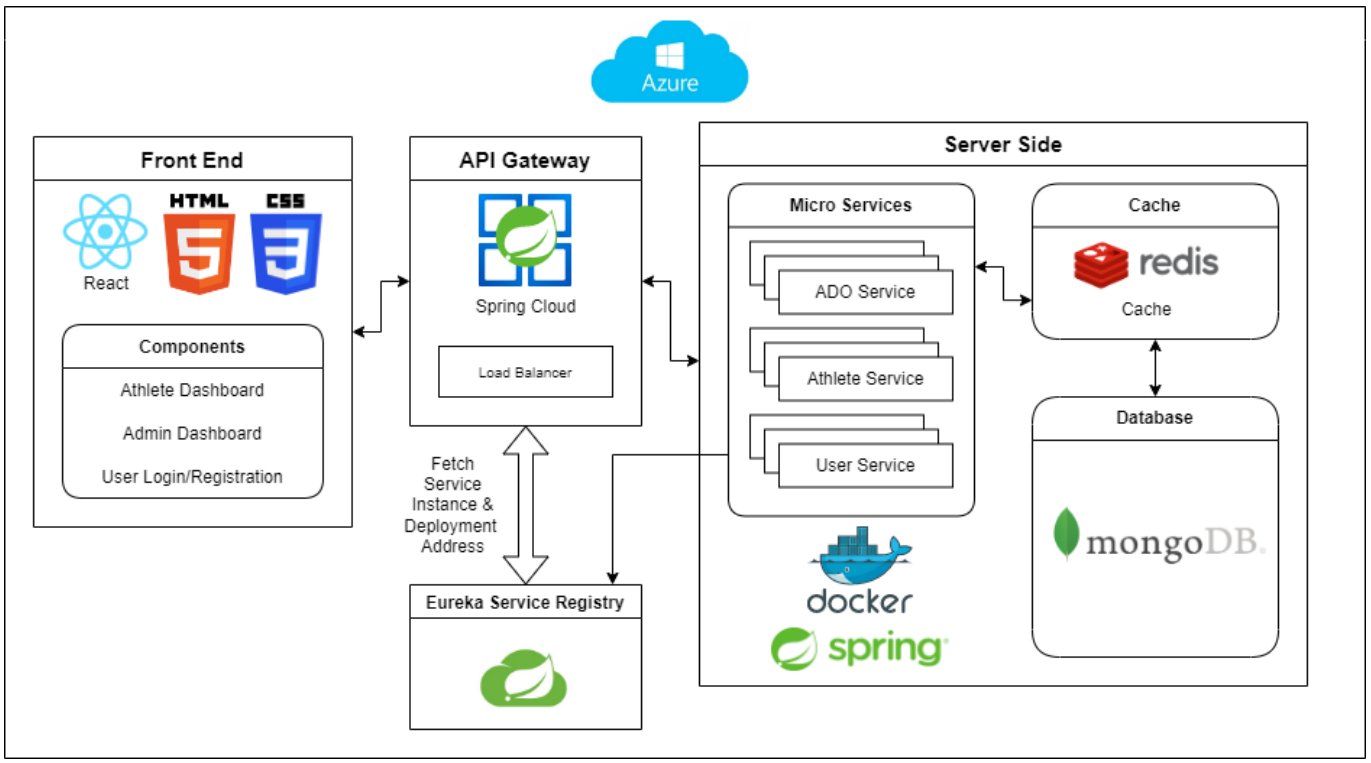


Figure: Technical Architecture

API Gateway

It's an entry point of the system. It acts as an edge server which exposes all the backend services to the outside world. It is implemented using Spring Boot Cloud. It hides all microservices from direct access and also provides security by implementing token based authentication and authorization. Any request which fails to achieve the security specification will be filtered from the gateway itself. Once the request is executed against all the security filters gateway will route the request to a corresponding service using a custom load balancer. Load balancer is an important sub-component within the gateway (edge-server) which tries to load balance the request against the multiple running instances of the services. It creates sticky sessions based on the locality (region) reference. As a result the request referring to a particular region will always be forwarded (loan-balanced) to a set of services within that region. It dynamically resolves the availability of the set services by communicating with Eureka Discovery Service which acts as a service registry. It then routes the request to the appropriate microservice based on the service information obtained from the Eureka service registry.

Eureka Service Registry

We have used Netflix Eureka which is a client side discovery service which stores all necessary information about available microservices. All the client micro services register themselves in this service registry on startup. Whenever any service wants to connect with any other service (like gateway service will route all the requests to specific service) it communicates with this service registry. Hence the inclusion of eureka service registry offers a highly reliable inter-service communication.

Service Implementation (Microservices)

We have used microservice based architecture to implement different services. These

include individual services for Athlete, ADO and User(Admin) which provides the required functionality as described in Requirements and Specification sections. All these services are deployed using docker containers. Using docker services along with docker-compose facilitates easy and on-demand scaling of the service instances. Multiple instances of these services are used to serve the user requests. Individual microservice communicates with the MongoDB cluster using Spring Data. It also communicates with the redis cache cluster to read/write most recently used data. The service will first do a cache lookup before reading the data required from the MongoDB cluster.

Redis Cache

Redis Cache cluster has been implemented to speed up overall read performance of the system. All the microservices will first fetch/look at the data in the cache cluster before reading from the MongoDB database. We have used caching in few of the read heavy functionality in which locally caching the data significantly improves the performance.

MongoDB Database

MongoDB cluster has been used as a backend persistent store. We have configured MongoDB as a sharded cluster. MongoDB sharded cluster which horizontally partitions the data and then stored it across different shared servers.

Following diagram depicts the overall architecture of mongo db sharded cluster and different components within it:

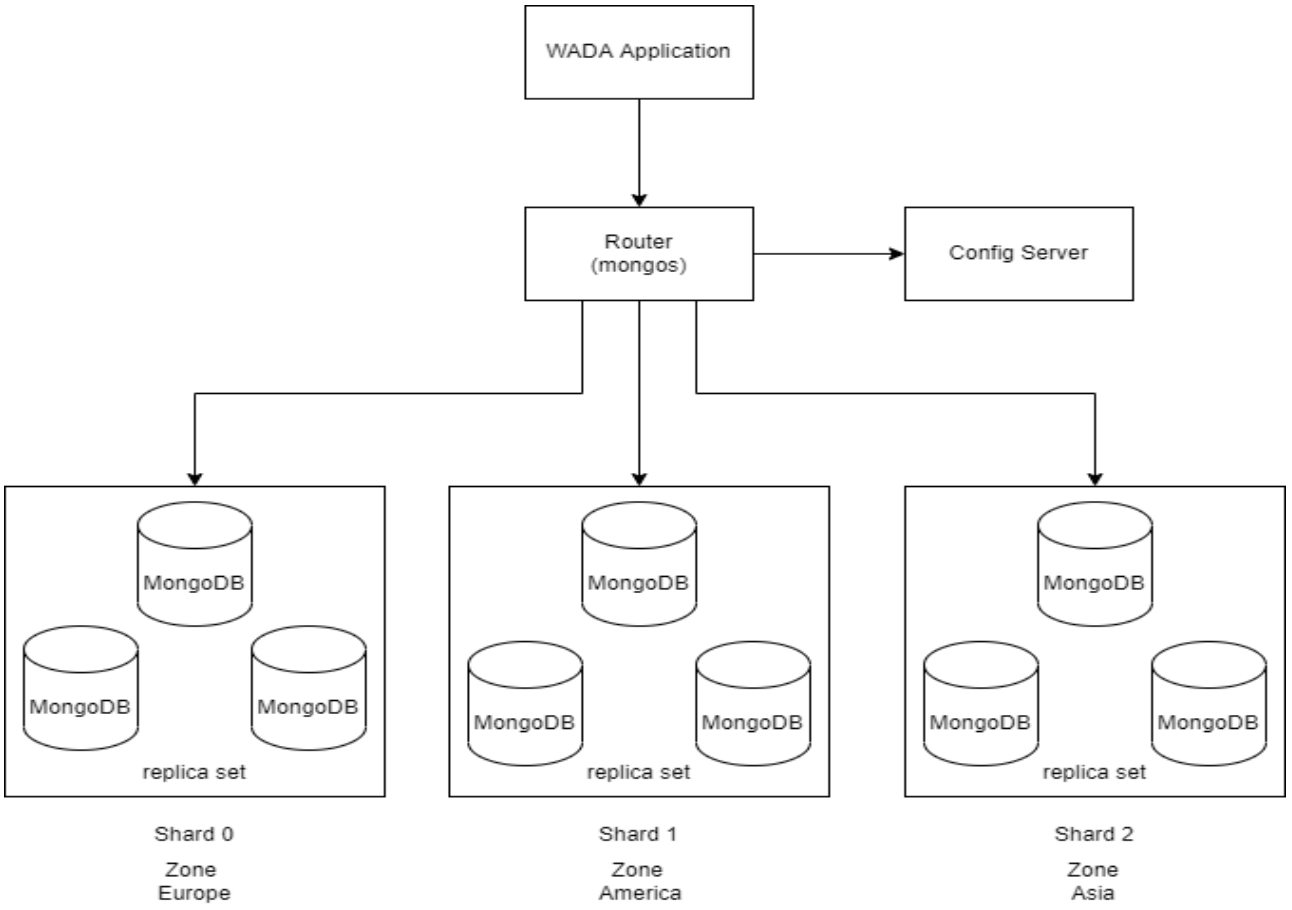


Figure: MongoDB Sharded Cluster

Components within MongoDB sharded cluster:

1. **Router (mongos)**: It acts as a routing interface of the MongoDB cluster. All the backend microservices will communicate with the database through this. In short the individual services will abstract off the sharding nodes of the database and all their requests will be handled through a mongos router. The mongos router communicates with each shard cluster by fetching the metadata from the config server.
2. **Config Server**: Stores the metadata and other configuration information about shard servers in a cluster.
3. **Shard:** Each shard server consists of the subset of the sharded/partitioned data. We have deployed each shard server as a replica set which will have initial configuration of 1 Primary node and 2 Secondary nodes. The sharding strategy used is based on the geographic regions/zones. So each shard cluster represents a database server in one particular region/zone as depicted above. This enables partitioning the data about Athlete , Athlete availability based on geographic region managed by their corresponding ADO's. To implement this we have defined a custom shard key for each type of collection which will include the region/location column allowing us to partition data in that collection in distinct zones. We have considered 3 regions namely Europe , America, Asia for this application and each shard server is dedicated to the respective zone.

IMPLEMENTATION

System uses different techniques to handle the failure scenarios. Some of the important techniques are as follows:

Service Failure Detection using Health Checks

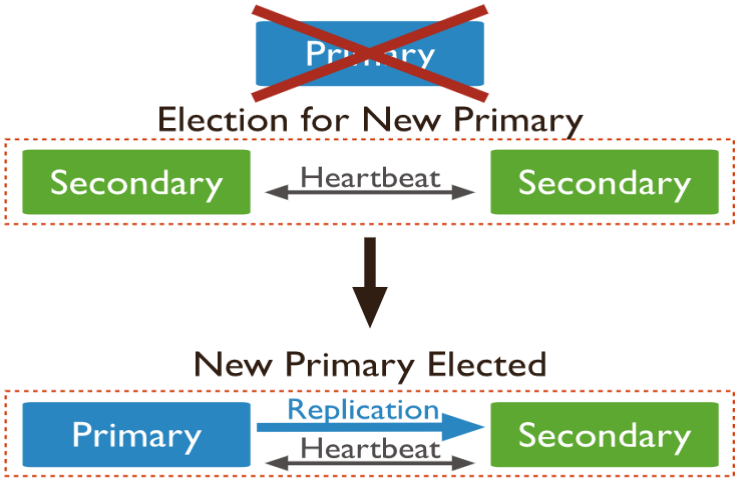
The Eureka Service Registry makes use of passive monitoring to monitor the status of all the registered clients. Eureka server exposes the API to which clients (individual microservices) send their heartbeat signals. The Eureka server then updates the status of the client as either "UP" , "DOWN". So Eureka server will renew the lease of the client instance. We have used this property in a custom load balancer implemented in gateway service. So Load balancer always retrieves the latest status of the available microservices. Hence load balancer will only consider the instances which are healthy and have "UP" status.

Due to network partition or the heart beat frequency the service registry status might not be consistent hence we have used some default fallbacks using circuit breaker patterns to handle such scenarios.

High Availability & Fault Tolerance using MongoDB replica set

We have created replica sets within each shard which provides high data availability. Each replica set has 3 members . This addresses the following scenarios -

a. When the primary replica goes down the election takes place within the secondary members to elect the new Primary. From the client side we have set the read preference to "**preferPrimary**" option so that even when the primary is not available (either it's down or the election process is in progress) it will read the data from secondary. Also we have configured the client to such that it retry the writes after some time if it detects the failure of primary while writing.



b. When the secondary nodes the cluster will still continue to serve the data as two other nodes (one primary and secondary) includes the full copy of the data.

Read & Write Preferences for Read/Write Reliability

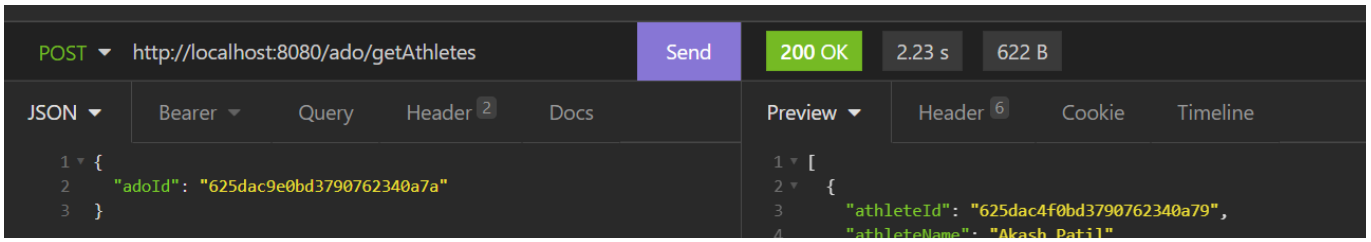
We have used read , write preference settings provided by MongoDB to make read and write operations more reliable. For example, write concern refers to the level of acknowledgement required to confirm write operation. We have used write preference level “majority” which implies that before acknowledging the write operation to the client it takes the acknowledgment from at least a pre-defined number of data bearing voting members. This ensures that write operations have been successfully replicated across the secondaries further ameliorating the data durability.

TESTING

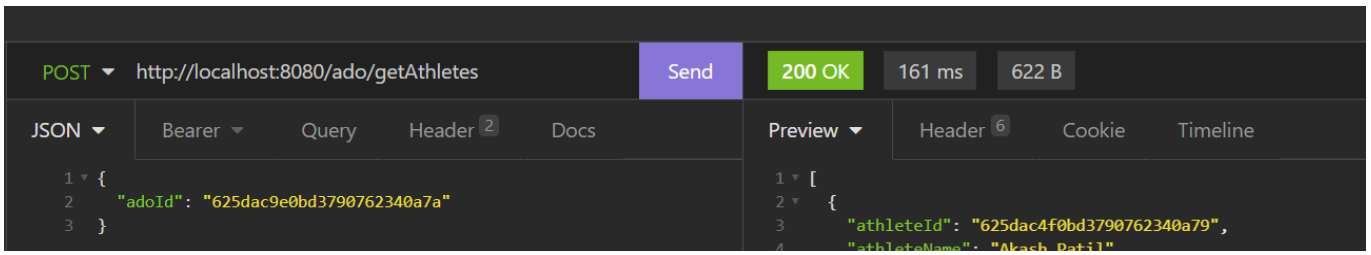
API Testing

We performed manual API tests for validating all the API responses. This testing was performed through the user interface as well as through API testing tools like Postman. User Interface allowed us to simulate User Acceptance Testing while API testing tools were crucial in validating correctness of API responses.

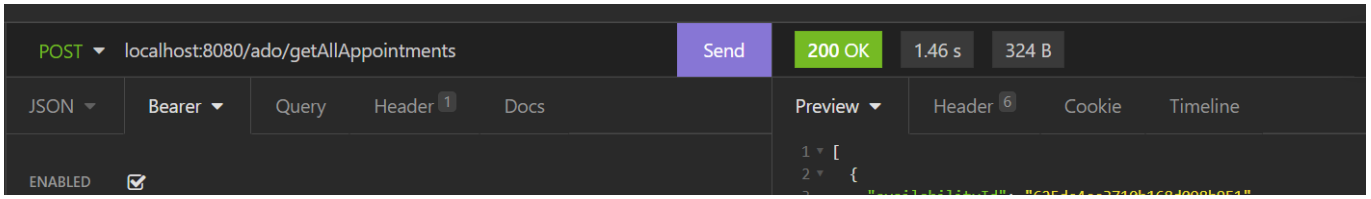
Further, we employed caching using multiple API endpoints to achieve faster response times. Below are some scenarios for getAthletes and getAllAppointments API endpoints that depict the quicker response time. The first time an API call is made, the results are fetched from the database and stored in the cache. For subsequent calls, the cached results are fetched reducing the number of database calls.



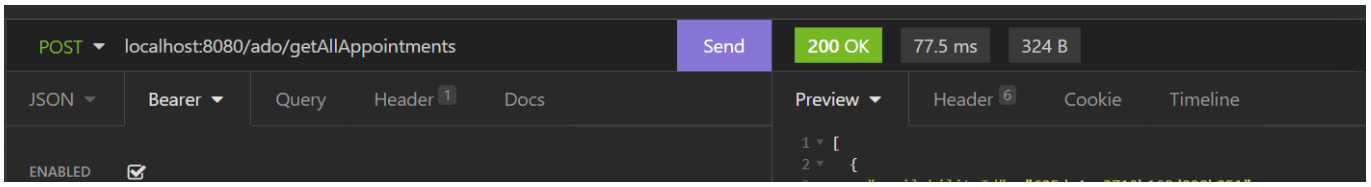
Response time for first call for /getAthletes API = **2.23s**



Response time for subsequent calls for /getAthletes API = **161ms**



Response time for first call for /getAllAppointments API = **1.46s**



Response time for subsequent calls for /getAllAppointments API = **77.5ms**

Database Testing

We performed manual testing of database sharding by simulating failure of primary shards. Primary replica of each shard was brought down by stopping the respective docker container. This resulted in automatic shift of primary status to secondary replicas of the shards. With this, we were able to test the fault tolerance and availability of our database.

Scalability Testing

To test if our application can handle heavy load, we performed scalability testing on our server and used the Locust Testing Framework. Locust is an open source load testing tool which enables creation of user behaviour and subsequent swarming of systems with multiple users.

We tested our server in two different configurations:

1. Load Balancing Disabled
2. Load Balancing Enabled

We generated an incremental load of 10 users per second using the Locust framework and performed tests for 1000 maximum users.

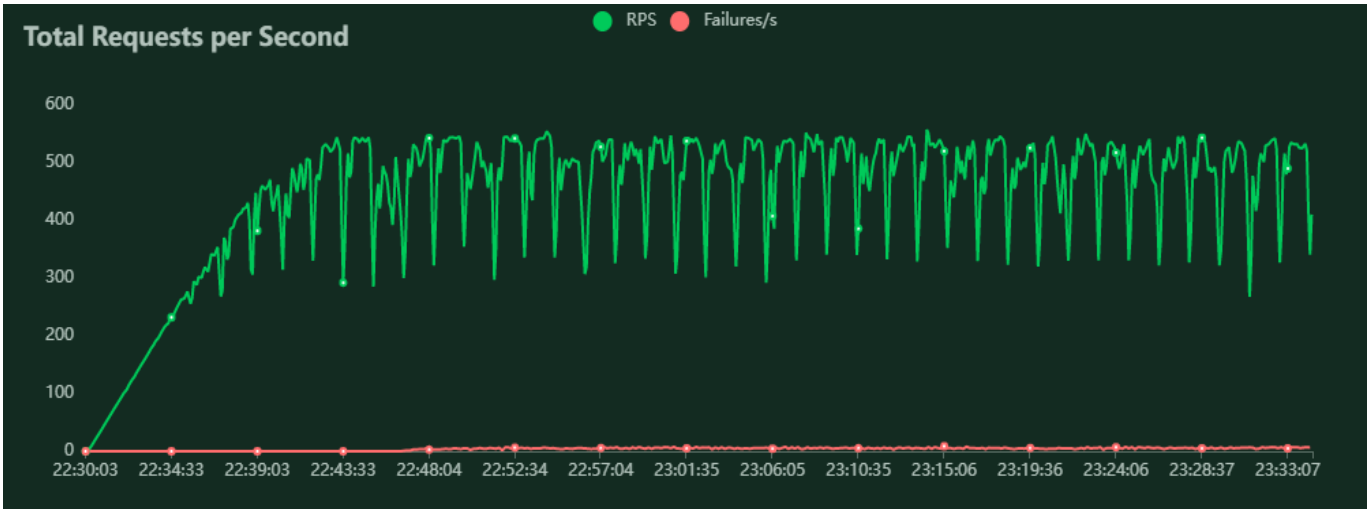


Figure: Total Requests per Second
Load Balancing Disabled

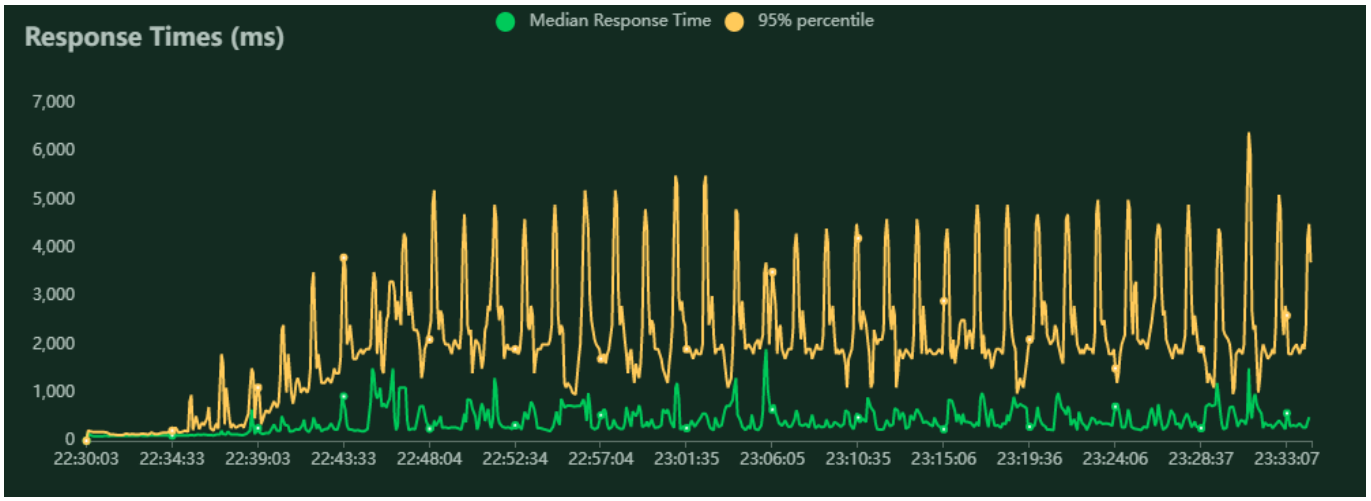


Figure: Response Times (ms)
Load Balancing Disabled

Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/ado/getAllAppointments	347422	2885	578	87	21458	597	90.0	0.7
POST	/ado/getAthletes	346983	2834	576	84	21538	742	89.9	0.7
GET	/availability	347010	3753	529	31	21558	2524	89.9	1.0
GET	/location	347214	2838	1774	85	21531	7954	89.9	0.7
POST	/user/login	346429	2822	1162	55	21474	223	89.7	0.7
	Aggregated	**1735058**	**15132**	**924**	**31**	**21558**	**2409**	**449.3**	**3.9**

Figure: API Requests Aggregated
Load Balancing Disabled

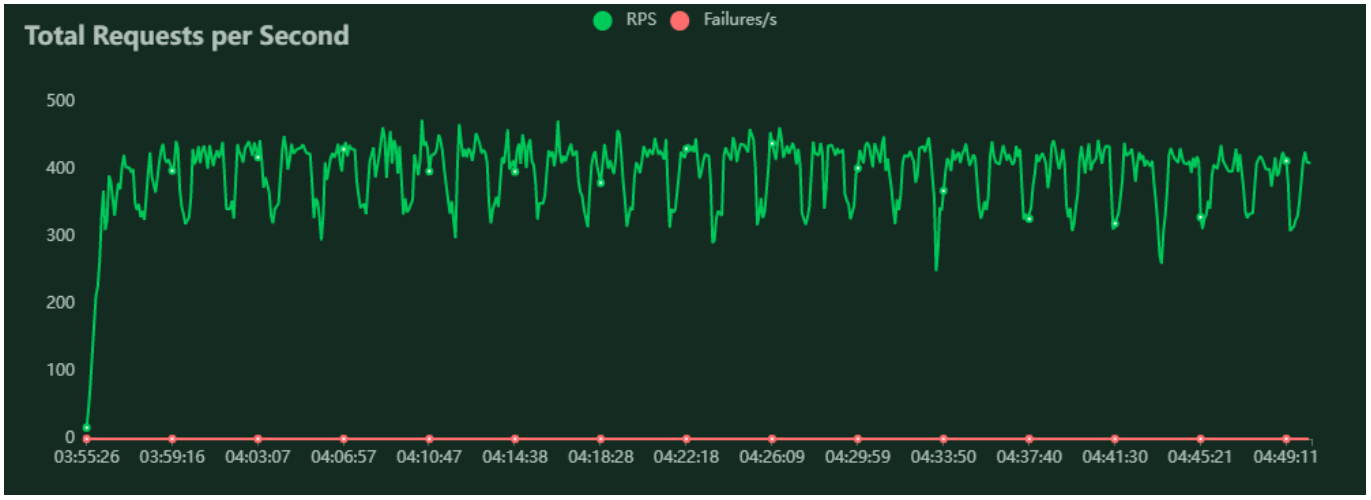


Figure: Total Requests per Second
Load Balancing Enabled

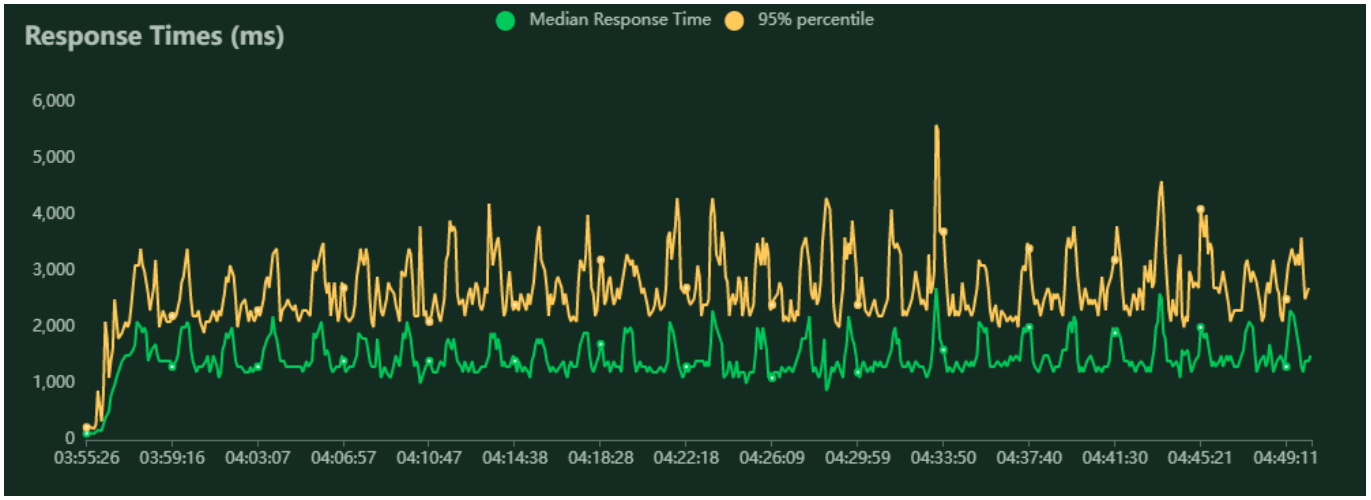


Figure: Response Times (ms)
Load Balancing Enabled

Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/ado/getAllAppointments	325052	0	881	93	12136	2	98.6	0.0
POST	/ado/getAthletes	324660	0	1030	96	7555	6108	98.5	0.0
GET	/availability/athlete/625e23e255a21d77983dcf50	325100	0	873	91	6831	314	98.6	0.0
GET	/location	325390	0	1058	92	8018	7973	98.7	0.0
POST	/user/login	1000	0	283	184	2344	234	0.3	0.0
	Aggregated	**1301202**	**0**	**825**	**91**	**12136**	**3596**	**394.7**	**0.0**

Figure: API Requests Aggregated
Load Balancing Enabled

SUMMARY

1. We have implemented highly distributed, scalable cloud native architecture for the system which can be used by World Anti Doping.
2. Given the further computational resources this system can be extended and scaled to handle many more world-wide regions. (We were only able to demonstrate for the limited number of regions due to constraints on computing resources).
3. From the myriad of possible technologies available for implementing such distributed design, After analysing pros and cons of it we finalised the technological stack most appropriate for the task in hand. With this we have tried to handle the distributed aspects of the system like high availability, reliability,scalability.
4. We observed a significant improvement in response times for Availability Lists and Athlete Lists after integrating redis caching into the system. This shows the importance of caching for high performance systems. Given the limited number of records for athletes and their availability in the database, it only made sense to use a single cache in the redis-server. However, It would be interesting to observe the impact of cache replicas in the redis-server if the system is exposed to millions of athletes.
5. We successfully managed to deploy and test the scalability of a database partitioned based on regions, as well as understand its impact and usefulness despite having a limited number of records.
6. Throughout the implementation of the project, we learned to implement the following:
    - Use and configure Redis caching on APIs
    - Configure read-write concerns for MongoDB client
    - Spin up multiple instances of microservices using docker
    - Integrate load balancing using Spring Cloud load balancer
    - Load testing on Locust
    - Sharding NoSQL database using shard keys
    - Integrating all components and deploying it on a server

CONTRIBUTION

Chaitanya Joglekar (21348468)

- Implemented And Configured MongoDB Sharded cluster, implemented data sharding & replication.
- Implemented custom load balancer.
- Worked on designing & configuring the deployment pipeline (creation of docker scripts) required for the project.
- Collaborated with other group members to define functional requirements.
- Collaborated with other group members to design the overall architecture of the proposed system.

Akash Purushottam Patil (21330950)

- Designed and created microservices with Akul, particularly focussed on the implementation of ado-service
- Integrated Spring Gateway (Token validation and token issuance) and Eureka server with the 3 microservices
- Implemented, integrated and tested Redis caching for ado-service and athlete-service

- Collaborated with other group members to define functional and non-functional requirements.
- Collaborated with other group members to design the overall architecture of the proposed system.

Akul Rastogi (20331653)

- Designed and created microservices with Akash, particularly focussed on the implementation of athlete-service and user-service
- Performed scalability testing of application using Locust Testing framework with Guowen
- Worked on designing & configuring the deployment pipeline (creation of docker scripts) required for the project.
- Collaborated with other group members to define functional requirements.
- Collaborated with other group members to design the overall architecture of the proposed system.

Guowen Liu (21332735)

- React front-end page framework building and Ajax request Encapsulation.
- Scalability testing and availability testing of servers using Locust.
- Collaboration with Xiang on front-end interface development.
- Collaboration with other group members.

Xiang Mao (21332237)

- React implementation of user interfaces.
- User interface interaction design.
- Calling the interfaces of distributed applications for front-end project implementation.
- Collaboration with other group members.
- Participate and Assist in the test working with Guowen.