Smart Cab Allocation System
MoveInSync Case Study

sankalpacharya@iitbhilai.ac.in
SANKALP ACHARYA

Analysis Report

Objectives

1. **Admin's Cab Allocation Optimization**

   The task is to develop an algorithm to optimize cab allocation for trips, reducing the overall travel distance; i.e. suggesting the best **available** cab based on proximity to the trip start location.

   In the **user-service**, when the user sends a *POST* request to the **/api/v1/user/booktrip** endpoint with the user's location details in the request, the response returned is the details of the best cab based on proximity to the user's location. The *user-service* internally calls the **/api/v1/cab/available** endpoint of the **admin-service** which returns the best cab possible to the *user-service*.

   Internally, all the cab locations: *(longitude, latitude)* are stored in a **MongoDB** collection in the cloud. MongoDB provides **geospatial indexing** which enables *storage & retrieval of geospatial data* in an optimized manner. ***2dsphere*** is one such type of indexing, which we create here on our cabs-location field. When **api/v1/cab/available** endpoint of the **admin-service** is called, a **geoNear query** using the **trip's start location** is performed on this index to find documents: *(in our case, cabs)* within a specified distance of a given geographical point: *(in our case, trip's start location)*.

This method of cab allocation is *extremely efficient* as *MongoDB* uses **geospatial indexing** along with **query optimization**. Also, utilizing **spherical geometry** gives highly accurate results.

2. Employee's Cab Search Optimization

The task is to enhance the user experience for employees searching for cabs by suggesting nearby cabs that are currently in use; i.e. suggesting/displaying **top 5** nearby cabs that are **engaged**.

In the **user-service**, when the user sends a *POST* request to the **/api/v1/user/displaynearbycabs** endpoint with the user's location details in the request, the response returned is the details of the top 5 best cabs based on proximity to the user's location: *(in our case, within 5 kms of the user)*. The *user-service* internally calls the **/api/v1/cab/busy** endpoint of the **admin-service** which returns the best engaged cabs possible to the *user-service*.

Internally, all the cab locations: *(longitude, latitude)* are stored in a **MongoDB** collection in the cloud. MongoDB provides **geospatial indexing** which enables *storage & retrieval of geospatial data* in an optimized manner. ***2dsphere*** is one such type of indexing, which we create here on our cabs-location field. When **api/v1/cab/available** endpoint of the **admin-service** is called, a **geoNear query** using the **trip's start location** is performed on this index to find documents: *(in our case, cabs)* within a specified distance of a given geographical point: *(in our case, trip's start location)*. The only difference between this and the above is: here we only search for cabs that are **"busy"**: *(apply a filter on the query)* & we apply a **limit** on the *maximum distance* with another filter: *(5 kilometers from trip start location)*.

This method of cab allocation is *extremely efficient* as *MongoDB* uses **geospatial indexing** along with **query optimization**. Also, utilizing **spherical geometry** gives highly accurate results.

3. Real-Time Location Data Integration

The task is to ensure seamless integration of real-time location data for cabs and trip start locations to enhance the accuracy of suggestions.

For the integration of trip start locations in real-time, this is already achieved through the user requests to the endpoints: **/api/v1/user/booktrip** & **/api/v1/user/displaynearbycabs**. Whenever one of these endpoints is hit, the current trip start location data is updated in the database.

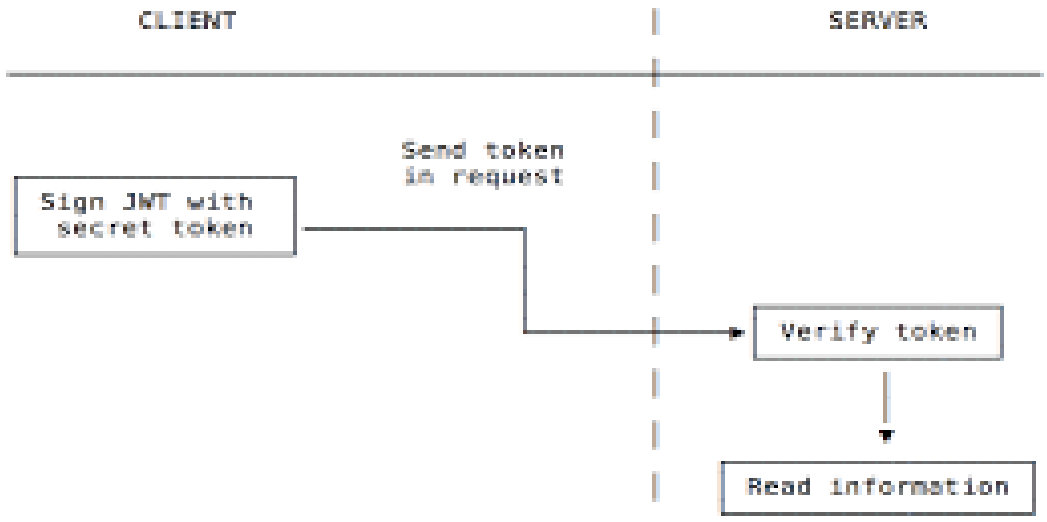
However, for the integration of cab locations in real-time, a **WS:** *(web-socket) server* called the **cab-data-service** is implemented. This service keeps listening to clients: *(in our case, cabs)* to send their real-time location data to the server. The server then reads the requests, parses it and updates the location of the respective cabs in the database.

Plus Points in Implementation

1. Authentication

JWT (JSON Web Token) authentication has been implemented in this case study. It is a method of securely transmitting information between parties as a JSON object. The **admin-service** and **user-service** have their *separate JWT signing keys*, to ensure no user can access admin paths and vice-versa. You can create a user/admin account using the **api/v1/user/create** & **api/v1/admin/create**

respectively. Once accounts are created, login through the endpoints: **api/v1/user/login** & **api/v1/admin/login** to get your *auth token*. This auth token shall be used to access protected paths for the respective service. For example, the paths: **/api/v1/user/booktrip** & **/api/v1/user/displaynearbycabs** require a *user-service signed JWT auth token*. Similarly, the endpoint: **api/v1/admin/addcabs** require a *admin-service signed JWT auth token*.



2. Cost Estimation - Time & Space

As discussed above, efficient & accurate searching algorithms have been applied using *geospatial indexing & querying* in **MongoDB**. The **time & space complexity** of a **geoNear query** in *MongoDB* can vary depending on several factors..Here's a breakdown of the factors that can influence the time & space complexity:

- **Indexing**: MongoDB uses geospatial indexes to efficiently perform geonear queries. The time complexity of querying with an index is typically **O(log n)**, where '**n**' is the number of indexed documents. This assumes that the index lookup is efficient. Also, geospatial indexes typically use a specialized data structure optimized for spatial queries. The space

complexity of the index itself is usually proportional to the number of indexed documents and the complexity of the index structure, which is often logarithmic **O(log n)**.

- **Document Filtering**: After using the geospatial index to identify potential candidate documents near the specified point, MongoDB may need to filter these documents further to ensure they meet additional criteria specified in the query (e.g., *maximum distance*). The time complexity of this filtering step depends on the number of candidate documents and the complexity of the filtering conditions. In the worst case, it could be **O(n)**, where **'n'** is the number of candidate documents.
- **Distance Calculation**: MongoDB calculates the distance between each candidate document's coordinates and the specified point. This calculation involves spherical geometry calculations, which can be computationally intensive but are typically efficient for small to moderate-sized result sets. The time complexity of distance calculations depends on the number of candidate documents and the complexity of the distance calculation algorithm.
- **Result Set Size**: The size of the result set returned by the **geoNear query** can also impact the overall time complexity. If the result set is large, it may take longer to transmit the data from the server to the client. However, in our case, the result set varies from *1 - 5 documents*, so it should not impact the complexity significantly.

In practice, geonear queries can often be performed efficiently for small to moderate-sized collections with appropriate indexing and query optimization. However, for very large collections or complex queries, performance testing and optimization may be necessary to ensure acceptable query response times.

3. Handling System Failure Cases

In the case of database failure, our *MongoDB* instance is hosted on cloud: on **Mongo Atlas**. It offers **99.995% uptime** according to its SLA. Hence, if in any case, our *3-replica-set* database goes down, it should not have much downtime.

In the case of server-side failure, **monitoring tools** have been deployed to watch out for any faults. In any case, due to comprehensive alerting: downtimes would not last for long.

4. Object Oriented Programming Language (OOPS)

Concepts of OOPS have been implicitly applied in the implementation of this case study. Some of them are *encapsulation*, *abstraction* & *composition*. These can be seen being leveraged when **Gin handlers** are used throughout the codebase.

- **Structs for Route Handlers**: In Gin, route handlers are typically defined as functions or methods attached to a router instance. Internally, Gin uses structs to represent route groups and individual routes. These structs often contain fields and methods for handling HTTP requests and responses, mimicking the **encapsulation** aspect of OOP.
- **Method Receivers**: Gin uses method receivers to define methods on its structs. These methods are responsible for handling different HTTP methods (*GET, POST, PUT, DELETE, etc.*), parsing request parameters, and generating responses. By attaching methods to structs, Gin achieves **encapsulation** and **abstraction** similar to OOP.
- **Middleware Composition**: Middleware in Gin is implemented using functions or methods that wrap around route handlers. Middleware functions can modify the request or response objects, perform

authentication or validation, and execute additional logic before or after the main route handler. This approach resembles the **composition** aspect of OOP, where behaviors are composed together to achieve the desired functionality.

5. Trade-offs in the System

Docker & the **microservices** architecture was followed in this case study. The pros and cons of this approach are listed below. **Advantages** of this approach:

- **Isolation**: Docker provides lightweight, containerized environments that isolate applications and their dependencies from the underlying host system. This isolation ensures consistency and reproducibility across different environments, reducing compatibility issues and dependency conflicts.
- **Portability**: Docker containers encapsulate an application, its dependencies, and runtime environment into a single package. This portability allows developers to build, test, and deploy applications consistently across different platforms, including development laptops, staging servers, and cloud environments.
- **Efficiency**: Docker containers share the host operating system's kernel, resulting in lower resource overhead compared to virtual machines. Containers start and stop quickly, consume fewer system resources, and enable higher resource utilization density, making them efficient for deploying and scaling applications.
- **Scalability**: Docker's lightweight architecture and container orchestration platforms like Kubernetes enable horizontal scaling of applications. By deploying multiple instances of containerized services across clusters of

nodes, organizations can easily scale their applications to meet varying demand levels.

- **Flexibility**: Docker supports a wide range of operating systems and programming languages, making it suitable for diverse application stacks. Developers can package applications written in different languages and frameworks into Docker containers without worrying about compatibility issues.
- **DevOps Practices**: Docker facilitates the adoption of DevOps practices such as continuous integration (CI) and continuous delivery (CD). Developers can use Docker containers to build reproducible build environments, automate testing, and streamline deployment pipelines, improving collaboration and deployment velocity.
- **Version Control**: Docker images are version-controlled artifacts that capture the application's state and dependencies at a specific point in time. By versioning Docker images and using container registries like Docker Hub or private repositories, teams can track changes, roll back to previous versions, and ensure consistency across environments.

However, some of the **disadvantages** are:

- **Complexity**: Docker adds complexity to the system architecture. Managing Docker containers, orchestrating them, and ensuring compatibility between containers and the host environment requires additional expertise and resources.
- **Security Concerns**: While Docker provides isolation between containers and the host system, misconfigurations or vulnerabilities in container images can pose security risks. Managing security updates, implementing access controls, and securing containerized applications require careful attention.

- **Networking Complexity**: Docker containers communicate with each other and external services through networking. Managing networking configurations, service discovery, load balancing, and security policies in containerized environments can be challenging, especially in distributed systems.
- **Persistent Storage**: Docker containers are ephemeral by default, meaning that any data stored within a container is lost when the container stops. Implementing persistent storage solutions for containers, such as Docker volumes or external storage systems, adds complexity and overhead.
- **Performance Overhead**: Running applications in Docker containers may incur a performance overhead compared to running them directly on the host operating system. The additional layers of abstraction introduced by Docker can impact application performance, especially in high-throughput or low-latency scenarios.
- **Vendor Lock-in**: Depending heavily on Docker and associated tools may lead to vendor lock-in. Switching to alternative containerization solutions or migrating away from Docker can be challenging and may require significant effort.

6. System Monitoring

System monitoring has been implemented in this case study using **Prometheus**: *a TSDB* & **Grafana**: a visualization dashboard. Prometheus is being used to *scrape metrics* such as **CPU usage, uptime, byte allocations** from the *user-, admin- and cab-data-services*. These metrics are then fed to Grafana where *real-time dashboards* have been deployed to monitor our application extensively.

7. Caching

Caching has not been integrated into this case study yet, however it will significantly reduce the load and response times of the application if we cache the JWT tokens using **Redis** or **DragonflyDB**. Also, various cache eviction policies such as ***DLIRS, LFRU***, etc. can be implemented for optimal resource utilization.

8. Error & Exception Handling

Extensive error handling & structured logging has been integrated in the application. **Structured logging**, such as that provided by **zerolog**, is important for several reasons:

- **Consistency**: Structured logging enforces a consistent format for log messages across an application. Each log entry consists of predefined fields with well-defined data types, making it easier for developers and operators to parse, filter, and analyze log data consistently.
- **Machine Readability**: Structured logs are designed to be easily parsed and processed by machines. By encoding log data in a structured format (e.g., JSON), applications can generate logs that are machine-readable and compatible with log aggregation and monitoring tools.
- **Contextual Information**: Structured logs can include contextual information such as timestamps, log levels, source code file paths, line numbers, and additional metadata. This contextual information provides valuable context for understanding log events and diagnosing issues in distributed systems.
- **Search and Filtering**: Structured logging enables advanced search and filtering capabilities based on specific log fields or criteria. Operators can filter logs based on attributes like severity level, timestamp range, error

codes, or custom metadata fields, making it easier to identify and troubleshoot issues.

- **Aggregation and Analysis**: Structured logs facilitate log aggregation and analysis by providing a standardized format for log data. Log aggregation tools can ingest structured logs from multiple sources, correlate related log events, and perform advanced analytics to identify patterns, trends, and anomalies.
- **Performance**: Structured logging libraries like zerolog are designed for performance and efficiency. They minimize overhead by encoding log entries directly into the desired output format (e.g., JSON) without unnecessary string formatting or concatenation, resulting in faster log generation and lower CPU usage.
- **Scalability**: Structured logging scales well in distributed and microservices architectures. By emitting structured log data with contextual information, applications can provide insights into system behavior, performance metrics, and application health across distributed environments.
- **Interoperability**: Structured logging promotes interoperability between different components of an application stack. Log messages emitted by one component can be consumed and correlated with logs from other components, enabling end-to-end visibility and troubleshooting in complex distributed systems.