**Design Documentation for Microsoft Employee Task:**

The design document contains an architecture design of the task, and the assumptions I have made while writing the code for the task assigned. It will also contain assumptions and work which remains to be completed, but can be expanded upon, if, and when, the task is expanded/scaled to a higher degree. It also contains chunks of files and code which will help with the scalability and increase in the amount of data and processing that might need to be done.

Since the task is an API endpoint design, I will cover mostly database right till the controller in an MVC structure (just model and controller… no view). The architecture is Microservice based.

**Database Design:**

I have designed a simple schema with one table included in the project. I have also left open codes/files/structures so that we can add multiple other schemas, and corresponding tables if we choose to add the same/scale it to proportion based on the given task of a billion employees.

My simple schema looks like:

|  |  |
| --- | --- |
| **Employee-Microsoft** | **Schema** |
| **Employee** | **Table** |
| ID | PK-Int |
| Name | Varchar |
| Role | Varchar |
| Department | Varchar |
| Reports To | Int |
| Salary | Int |
| IsActive | Boolean |

Here we can add multiple tables in the schema and add multiple schemas to the project.

With there being a billion employees, it is important to fragment and index the database based on different assumptions. I have added few Database assumptions below:

**Database Assumptions:**

1. For the database, we can index/fragment the database based on the use case we have. For example, if more of the operations being performed based on Employee ID’s, indexing, and can be done over ID, and even fragmentation can be approached as per our use case.
   1. Like we can have the first ten million records fragmented into one part, the next 10 million into a second part and so on. And over a billion employees, we can have one hundred such fragments of the data.
   2. If the use-case is over employee names, the same can be applied over the twenty-six letters of the alphabet and indexing and fragmentation should be done as above.
   3. Similarly, we can fragment and index over geological regions, something like East Coast/West Coast or region-based like Chicago/Virginia/Ohio etc. or continent-based like NA/SA/Asia, etc. Here we can also have a combination of multiple of these, like within North America (NA) we have multiple fragmented databases in Chicago, Ohio, Virginia etc.

This assumption is based on our use case and how the business wants to distribute databases. It is also important to note that we need backups of the same as well and that when the database reaches such a high scale, each backup will be cost-intensive as well. So, a combination of the above fragments/indexes can be applied to the database as well. This should be done by a data engineer and/or DevOps.

1. For Performing CRUD operations on the database (Insert, Select, Update, Delete), there are multiple things to keep in mind. In general, read operations are the maximum, followed by update, insert, and lastly delete. This varies depending on different businesses, for example a B2B company which only manages receipts, and only generates quarterly reports has a maximum of Insert operations, then Updates, then Deletes, and only every quarter has Select tasks. Which is why it is necessary to identify what our business goal is. Assuming the generalized idea mentioned above, we can also implement different cache and data brokers like Redis over different fragments. Here, Delete is a write-heavy operation, and we can implement FIFO (First in First out) based windows for the read operations in conjunction to the Redis cache which stores if a record is deleted. Doing this allows us to batch delete all delete operations simultaneously after every set interval (or during RPS downtime), while also maintaining concurrency of the database.
2. While dealing with multiple copies of the databases, it is also better to have master-slave architecture to improve data availability, fault tolerance, and performance (pair containing one master multiple read only slaves, and multiple copies of such pairs). This also helps with all of CPA (concurrency, performance, availability). And additional benefits are easy scalability, easy replication, and better load balancing options.
3. As for load balancing, while dealing with multiple copies of the database, it is also important to properly load balance based on expected load of requests and implement it in FIFO (First in First Out) windows such that no single copy of the database/fragment is overworked while other copies have downtime. The assumption here is that this is setup properly in a pipeline by a data engineer/DevOps team. I am also assuming that we are making sure that proper care is being taken to choose a database implementation which supports proper granularity so that the employee records are properly locked/unlocked and accessible by different users. Older databases get slower along with larger datasets, and for certain processes, it is important to make sure that system timeout does not happen owing to processing speed/delays from the database, etc. which I am assuming can be expanded on later if the task deems so.

**Backend Design:**

When designing the backend, I have chosen to use Java-Spring Boot Microservice structure with MVC based architecture. Here I have tried to use OOP as much as possible and kept the code as modularized as possible. The simple structure starts from connecting the spring boot basic app with the database. This can be done by setting up the database connection in the application.properties file inside the resources folder. Once configured, we can start and test the app with basic database connection. Spring-Boot has a default embedded server with one of the options being Apache Tomcat. It is a powerful resource which allows me to add dependencies on the go, as and how the direction and the requirements of the project changes.

**Employee Entity:**

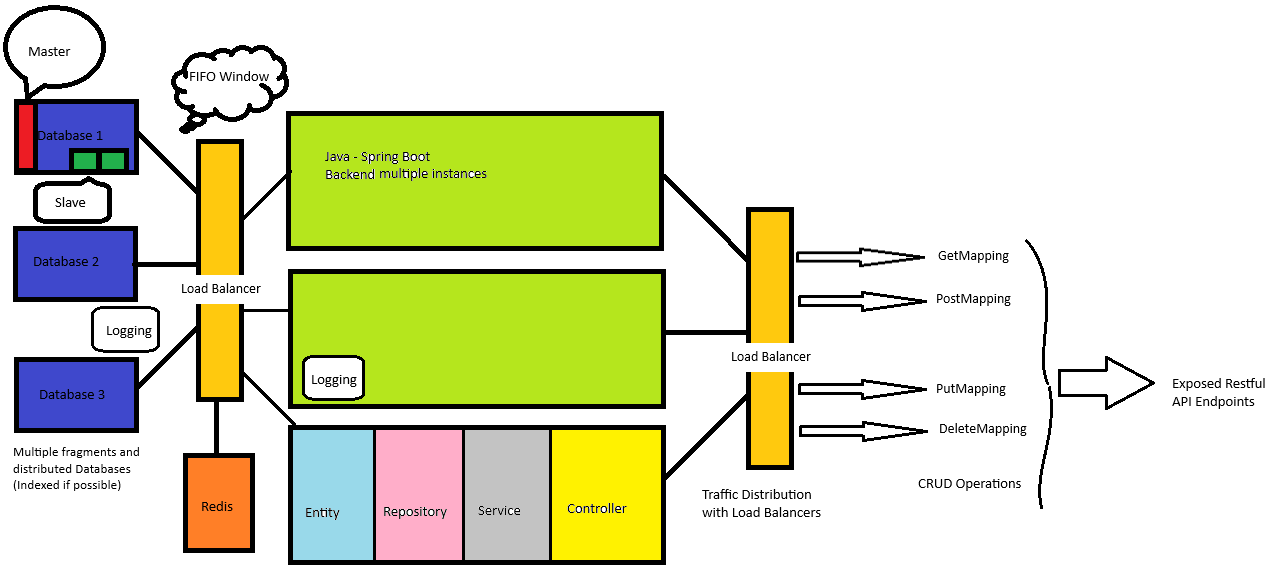
I started with the employee entity with the help of the ORM Java Persistence API (JPA). We can use different service-mesh here to connect different microservices as well. Here in Java, when referring to entities in the context of database tables, these are objects that represent records in a database table. These objects are part of an Object-Relational Mapping (ORM) framework, which allows developers to interact with databases using Java objects. An entity class is a Java class that represents a database table. Each instance of the class corresponds to a row in the table. An entity object is an instance of an entity class. It represents a specific record in the associated database table.

**Backend Code with connections on Database/Service Layer:**

1. Each entity class created is equivalent to a table in the database. Every entity within the class represents a column in the tables and every entity object is equivalent to a record. Here my assumption is that for every table that we create/will create, we will have equivalent entities created in Java as well.
2. With JPA, for each entity that we create, we can have an interface for basic database operations including specific ones, for individual tables. These interfaces are quick and are easy to write and understand. However, for complex queries involving multiple tables, it is better to write the same business logic in services.
3. Being OOP, Java is heavily modularized and helps with abstraction and inheritance. I can reuse every public class and methods associated with specific interfaces.
4. The service layer contains all the business logic associated with the project including all methods for backend we are to use. This in turn is connected to the controller which helps expose endpoints of the application for external use.
5. Controllers are the crux of the CRUD operations. In my code, I am using REST API service. Java in spring boot has useful annotations and beans which help with creation of these processes. In the code, I have different Mappings like Get Mapping, Post Mapping, etc. which provide the endpoints with different HTML functionality of Get, Put, Post, Delete etc.

**The System:**

Finally, all these components come together to form the microservices with Restful API endpoints which are publicly available. Below is an image which represents the cumulation of all the above points into one fine refined microservice system which has publicly exposed REST API endpoints:



Every element in the diagram is replicated in every object. For simplicity and coherence, I have shown the inner components in only one object. But they are technically replicated in all adjacent objects.

The diagram explains the way I have designed the Microservice which exposes CRUD operations based Restful Endpoints publicly.

We may have multiple sets and copies of (if possible fragmented) databases which can be indexed, and each database having a pair of a master, and multiple slaves, where the function of the master is any and all database updates (insert, update, delete) while the slaves only have access to read (select queries).

Here whenever the C/U/D of CRUD is called, they go through the master database of each copy of the database (if fragmented, then all the required copies which may contain that part) and load it in all of them concurrently. Alternatively, we can also have a caching system like Redis. With this project of billions of employees, caching is not useful, however, fragmentation can prove to be useful. Delete being a write heavy request, I also want to make sure that whenever C-R-U requests come in, we check with caches like Redis to make sure that these operations are not being performed on something that is already to be deleted, and then batch delete everything with a daemon. This will also reduce the required database indexing and the intervals at which we need to continue indexing it. Here we can also have horizontal partitioning or Sharding depending on our use case, so that we have common records across different databases, so that in case all copies of one of the fragments go down, most users are still not affected.

Next up is the load balancer which makes sure that each database fragment containing similar pieces gets targeted equally, and not that only one fragment gets all traffic whereas other fragments sit by idly. This also applies to multiple load balancers as well. Since having more than one such instance of backend/database running together might mean we have added more than one load balancer between the backend and the database.

The backend as explained above consists of a layer of entity, which is equivalent to the schema/tables/columns/records of the database, which connects with the JPA based interface for basic database operations, and the service layer which executes complex database operations. Both of which connect with the controller who connects the View portion of the MVC to the backend, giving us endpoints, which can be exposed externally. There again, based on the task of managing a billion employees, we have load balancing required for each instance of backend to get similar amounts of RPS.

However, on the other end, whoever is using these exposed endpoints, they will not know which instance of the backend/database is managing their traffic. To them, we only expose the endpoint.

We also need to have logging at every level. In the figure, I have added database logs, Backend based logs and API logs. At every connection, we should ideally have logging. This helps identify where faults might have occurred for developers who are not familiar with the code itself. For example, if User reported break, which we cannot replicate locally, it may be a temporary service downtime, or it may be a master/slave-based issue where master properly updated the database, but the slave is running on an older version, without connecting with Redis. Logging helps with all these types of debugging.

Overall, this is the design of the REST API endpoints in a Microservice structure, which is scalable to a billion employees, and because of the modular design, we can increase/decrease each different component based on our necessity/requirements.

**Security Aspect:**

1. Spring-Boot with Java-Gradle provides a set of features/dependencies for securing applications, and it integrates seamlessly with Spring Security. Some key features:
2. Authentication: Spring Security handles user authentication through various mechanisms like form-based, HTTP Basic, or OAuth.
3. Authorization: One should always configure fine-grained access control using annotations or configuration, specifying which roles or authorities are required to access specific resources.
4. Token-Based Authentication: For stateless authentication, Spring Boot supports token-based authentication, commonly used with RESTful APIs. JSON Web Tokens (JWT) are a popular choice for token-based authentication with which I am familiar.
5. OAuth 2.0: Spring Boot supports OAuth 2.0, an open standard for access delegation. It enables secure authorization workflows, commonly used for third-party application integration. Spring Security provides convenient annotations and configurations for OAuth 2.0.
6. Role-Based Access Control (RBAC): Leverage role-based access control to grant different permissions to different user roles. Spring Security supports easy configuration of roles and authorities.

**Other Features/Assumptions:**

1. Gradle has functionality for adding Log4J directly as a dependency of the project which can be directly added to the build.gradle for logging errors etc. Other than that, we can use other external tools like SumoLogic, Kiali, Kinesis Firehose, etc. for keeping logs at multiple levels of the code. I have created opening files for the same which can be edited/completed later to have a total log of the project. This is important because anything that can go wrong can get caught with the proper server, API, database, etc. logs. Multilevel logs also help ascertain where exactly the break happens and shorten our debugging time if we do not know the code but know the log.
2. As for Code Coverage, Spring Boot comes with its coverage option, or we can add the 'JaCoCo' code coverage locally to our project. I prefer using SonarQube (and its local plugin for testing code locally) for code coverage because integrating code with SonarQube also helps with better coding practices, it identifies code smells and vulnerabilities.
3. While designing code, it is also imperative to evaluate the connectivity with every Microservice component we have, along with their RTT for connection. Older databases get slower along with larger datasets, and for certain processes, it is important to make sure that system timeout does not happen owing to processing speed/delays from the database, etc. which I am assuming can be expanded on later if the task deems so.
4. I am assuming that we are making sure proper care is being taken to choose a database implementation which supports proper granularity so that the employee records are properly locked/unlocked and accessible by different users.
5. To have consistency and availability of data, I assume that fragmenting data into separate databases will allow for the microservice to keep servicing at high RPS, in cases where a database or a microservice server fails, only users of that fragment will be affected. Even this issue can be partially mitigated by having multiple copies of each fragment.