Mortgage

Here is a high-level design of the system:

A diagram of a software application

Description automatically generated

As you can see, the design system is capable of running in a cloud environment and can be horizontally scaled due to the stateless nature of the application. Since this is an assignment, I focused on implementing only the minimum mandatory components of the above diagram.

The diagram consists of various elements relevant to a microservices architecture. For instance, load balancers are used to distribute incoming traffic across API gateway instances and application instances, which represent the product I was asked to develop. Additionally, the diagram includes components like Throttling and Rate Limiting, which are essential for preventing denial-of-service (DoS) attacks and ensuring resources are not wasted by users who might intentionally or unintentionally overload the system.

Authentication and Authorization are integral parts of any ecosystem but are not implemented here because this API is designed to fetch and provide public information. However, for APIs that include admin endpoints, implementing proper authentication and authorization mechanisms would be essential.

Design decisions:

* **Microservice Architecture:** To enable scalability, flexibility, and independent development and deployment of loosely coupled services tailored to specific business functionalities.
* **Identity and Access Management:** every critical system which is open to users should be under control of this system to avoid any unauthorized access. For example here if we want to implemented admin APIs so we need to enable Resource Server capability of Oauth2 protocol on any type of Auth system such as Keycloak or Auth0 or Azure ID Auth.
* **Two category of users**, public users and admin users. The admin API is not implemented in this assignement.
* API Gates:
* **SQL Database**: To have a permanent storage for feature usages such auditing (as ING a bank we need to have history for everything for auditing and regulatory purposes)
* **Redis Cache**: To speed up all fetch operations better to move data results for requested interest rate into a distributed and decentralized cache such as Redis, Hazelcast, or etc.
* **Log Events**: all log streams should be redirected to a log processing applications such as Splunk to have an insight about them and use them for monitoring and visibility purposes
* **Monitoring System**: This part is crucial to have visibility on our APIs specially the data which is gathered from API metrics data by Prometheus. These two facilities are implemented already in the project but still need have a work around to implement Grafana and Prometheus server to visualize the metrics.
* **Application Servers**: Developed application is running in multiple instances to have high available and resilient Mortgage service in our design solution.
* **API Gateway**: An API Gateway acts as a single entry point for managing, routing, and securing requests to microservices, while handling concerns like authentication, rate limiting, and load balancing.
* **Throttling and Rate-Limiting**: This component controls the flow of requests to ensure fair usage, protect resources from overuse, and maintain system stability.

**Low level Software design:**

For this API, we are using Domain-Driven Design (DDD) with a layered architecture:

|  |  |  |  |
| --- | --- | --- | --- |
| Domain | Validation | Rest Controller | Config |
| Service Façade |
| Cache |
| Service |
| Repository |
| SQL Database |

**SQL Database**

Starting from the bottom, I used an H2 database to simplify implementation, but I also incorporated the database versioning tool Liquibase to manage schema changes efficiently, ensuring consistency and enabling easy rollbacks across different environments. The project also includes a profile for generating fake data, which is enabled by default in the Spring "dev" profile.

**Repository Layer**

In the repository layer, I used Spring Data and JpaRepository. JpaRepository is a Spring Data interface that provides built-in CRUD operations, pagination, and query capabilities for managing relational database entities with minimal boilerplate code.

**Service Layer**

The service layer contains the core business logic of the application. A local caching mechanism, Caffeine, is applied to business methods to enhance performance. While the application is ready for production, it is recommended to replace the local cache with a global caching solution such as Redis or Hazelcast. However, due to time constraints, I opted for a local cache implementation to demonstrate my knowledge of how to serve users in real time without additional efforts.

**Service Façade**

A Service Façade is a design pattern that provides a unified and simplified interface to orchestrate underlying services and business logic, decoupling clients from implementation details and improving maintainability by consolidating service interactions in one layer.

**Rest Controller**

The Rest Controller is the outermost layer and serves as the entry point of the system, where users interact with the application. All input validations and request-handling logic are applied in this layer.

**Validation Layer**

The Validation API is not only used for simple validations, such as enforcing minimum field values, but also for handling complex business logic checks. For instance, this project implements two key business constraints:

1. The amount of the mortgage.
2. The relationship between income and the requested mortgage.

Since these types of business rules are cross-cutting concerns, they are implemented using the Validation API, which is integrated as an aspect (AOP). Two interfaces, IncomeConstraint and MortgageAmountConstraint, are designed to enforce these rules. Their corresponding implementations are IncomeValidator and MortgageAmountValidator.

**Config**

all configurations are gathered here

**Detailed information about the tools and techniques used in this project:**

* Unit Testing: Implemented with JUnit.
* Integration Testing: Achieved using MockMVC.
* Mockito: Used for mocking in unit tests.
* Liquibase: Database versioning and schema management.
* Maven: Dependency management and build automation.
* GitHub CI: Continuous integration configured via .github/workflows/ci.yml.
* Swagger UI: For API documentation.
* Docker: Dockerfile and Maven's Fabric8 plugin are used to build Docker images.
* Docker-Compose: Simplifies running and managing Docker containers.
* Spring Profiles: Separate profiles for "dev" and "prod" environments.
* Testcontainers: Added to the project but not used due to time constraints.
* Exception Handling: Implemented using @ControllerAdvice.
* Logback: Used for logging.
* In memory and simple user management

**Future Works:**

* It would be better to implement a workaround by adding save methods along with a cache eviction mechanism.
* Integrating the application with an authentication and authorization server.
* Using Redis or Hazelcast as a distributed cache layer instead of a local cache.
* Including performance testing in the project using Gatling.
* Following an API-first development approach.