The focus of this code challenge is on the backend. You need to implement the initial steps in a transaction processing application. The initial version is basically a CRUD app, but in later stages more complex business logic and validations will be added.

Make sure the code sent runs out of the box. Manage the project with Maven and structure it accordingly. Provide a docker-compose file that can be used to run the application and the database. Keep the tests as self-contained as possible: this means that no external dependencies should be manually installed for the developer to run the tests. Include a README.txt with the project, which should contain step-by-step instructions on how to (1) run the application using docker-compose, (2) configure the client, (3) build the project from Maven, and (4) run the client. Put the project under source control on GitHub and email us the repo link.

Challenge:

We need to implement a backend application that will be used to record Transactions reported by Suppliers.

1) Design a REST API for a company to add/remove its suppliers.

A Supplier is a Resource defined as:
- Supplier name (mandatory)
- Address (mandatory)
- Contact No. (optional)

The API should provide CRUD operations.

2) Create a Spring Boot Java backend that implements this API.

The backend should persist the above data to a database. Decide which DB and framework/library to use to persist and read the data.

3) Create a Spring Boot client Java application that simulates what a real-life Supplier would do, by generating a dozen or so transactions and POSTing each transaction to the API. A transaction should contain:
- Random Transaction ID
- Valid Supplier ID
- Current date and time
- Content (alphanumeric data). Up to 64KB

4) Whenever a transaction is `POSTed`, the application needs to:

a) persist it locally

b) call a microservice (which exposes a REST API). For example `POST http://tracker.local/supplier-transactions` with the transaction as a body

c) return a status code of 202 to the caller

If there is any error while persisting the transaction or when calling the microservice, the application should return a status code of 500 to the caller.
The operation must be idempotent to allow retries from the client, using a client-generated Idempotent Key.
Make sure that the operation handles timeouts and unexpected responses from the microservice.

5) Create unit or integration test cases, or both, wherever logically it makes sense in your opinion.

Questions:

1) Our sales team tells us that some of the Suppliers will be sending around 1 million transactions per day, mostly during business hours, and that they will perform around 10 million queries per day. Which steps will you take to make sure the application can handle this load?

2) How can your application support high availability and handle more concurrent transactions? Which changes do we need to make on the deployment?