Spring Boot Rate Limiter Project Architecture

# ✅ Flow Overview

+--------------------+  
| Client (Browser / |  
| Postman / etc.) |  
+--------+-----------+  
 |  
 | HTTP Request  
 v  
+--------+-------------------+  
| RateLimiterController |  
| - /api/register |  
| - /api/request |  
| - /api/client-config |  
+--------+-------------------+  
 |  
 v  
+--------+-------------------+  
| RateLimiterService |  
| - registerClient() |  
| - isRequestAllowed() |  
| - getClientConfig() |  
+--------+-------------------+  
 | \  
 | \  
 v v  
+--------+-------------------+ +----------------------------+  
| DatabaseStore | | FixedWindowRateLimiter |  
| - saveClientConfig() | | - isRequestAllowed() |  
| - getClientConfig() | +----------------------------+  
+--------+-------------------+  
 |  
 v  
+----------------------------+  
| ClientConfigRepository |  
| - save() |  
| - findById() |  
+----------------------------+

# 📁 Folder Mapping to Flow

|  |  |
| --- | --- |
| Layer / Component | Folder Path |
| Entry Point (main) | XeniRateLimiterApplication.java |
| Controller Layer | controller/RateLimiterController.java |
| Service Layer | service/RateLimiterService.java |
| Rate Limiting Logic | service/strategy/FixedWindowRateLimiter.java |
| Data Access Wrapper | store/DatabaseStore.java |
| Repository Layer | repository/ClientConfigRepository.java |
| Entity / Model | model/ClientConfig.java |
| Configuration | resources/application.properties |
| Testing | test/.../RateLimiterControllerTest.java + Service test |

# Layer-wise Advantages

## Controller Layer (controller/)

* • Separation of Concerns: Keeps HTTP-related logic isolated from business logic.
* • Easy to test: Unit testing of endpoints is straightforward using tools like MockMvc.
* • API Gateway: Can manage authentication, request validation, etc.

## Service Layer (service/)

* • Centralized Business Logic: All business rules are in one place.
* • Testable Logic: Easy to write focused unit tests.
* • Loose Coupling: Controller doesn’t need to know implementation details.

## Strategy Layer (service/strategy/)

* • Flexibility: Easily plug in new algorithms without modifying core service.
* • Separation of Algorithm Logic: Keeps algorithm details away from business logic.

## Store Layer (store/)

* • Decoupling: Decouples persistence logic from service.
* • Simplified Testing: Can mock store interactions independently.
* • Centralized Data Access: Manage transactions, caching, etc., in one place.

## Repository Layer (repository/)

* • Less Boilerplate: Spring Data JPA provides CRUD methods.
* • Standardized Access: Ensures consistent data access patterns.
* • Easy Integration: Works well with databases and JPA.

## Model Layer (model/)

* • Clear Data Contracts: Central definition of data.
* • Validation Ready: Annotated with validation constraints.
* • Reusability: Shared across layers.

## Test Layer (test/)

* • Assures Code Quality: Helps catch bugs early.
* • Isolated Testing: Can independently test layers.
* • Confidence in Refactoring: Safer and easier changes.

# 🔄 Overall Benefits of Layered Architecture

✅ Maintainability – Easier to locate and modify code in a specific layer.

✅ Testability – Each layer can be tested independently.

✅ Scalability – Individual layers can scale as needed.

✅ Team Collaboration – Developers can work on different layers independently.

✅ Reusability – Business logic and strategies can be reused.

# 🔄 Optional Improvements

* 1. Interface for the Store Layer: Define an interface (e.g., ClientStore) to abstract persistence logic.
* 2. Strategy Pattern with Interface: Create a RateLimiterStrategy interface for pluggable algorithms.
* 3. Exception Handling: Use @ControllerAdvice and custom exceptions for clean error management.
* 4. DTO Layer: Use DTOs to decouple internal models from API contracts.