Day23 Spring Boot Actuators and Metrics, Logging.

Day22 RBAC – Role Based Access Control. JWT based Authentication & Authorization.

1. Spring Security always check for the Session.
2. If no session is associated with the request, it will redirect to login page.
3. After completing the credentials validity check, it creates a session with unique session-id.
4. Adding JWT Dependency. [External Dependency for jjwt-api, jjwt-impl, jjwt-jackson]
5. Create SecurityConfig [@Configuration, @EnableWebSecurity ] – requestMatchers , JwtAuthFilter, JwtUtil
6. DTO – LoginRequest, RegisterRequest (Used for transmitting the data between client and server – It’s not saved any where else) [Data Transfer Object]
7. UserDetailsService (loadByUsername())
8. Registration endpoint created a record in the Database table. (It won’t create JWT)
9. Login/Signin endpoint created JWT after validating the credentials.
10. Subsequent request after JWT creation, needs to have a header named AuthHeader. With a content Bearer “actual-jwt”

CSRF – Cross Site Request Forgery

DAO – Data Access Object

DTO – Data Transfer Object

SpringBoot Actuators –

Spring Boot Actuator is a sub-project within the Spring Boot framework that provides production-ready features to monitor and manage Spring Boot applications. It exposes various operational data about a running application through HTTP endpoints or Java Management Extensions (JMX).

Key Features and Endpoints:

* **Monitoring and Metrics:**

Actuator provides endpoints to gather real-time metrics about the application's health, performance, and resource usage. This includes:

* + /health: Shows application health information (e.g., database status, disk space).
  + /metrics: Provides various metrics like CPU usage, heap memory, threads, loaded classes, and HTTP request/response details.
  + /info: Displays arbitrary application information.
* **Application Management:**

Actuator offers endpoints to interact with and manage the application at runtime:

* + /beans: Lists all Spring beans in the application.
  + /env: Exposes properties from Spring's ConfigurableEnvironment.
  + /loggers: Allows viewing and changing the logging levels of loggers at runtime.
  + /shutdown: Enables graceful shutdown of the application (not enabled by default).
  + /threaddump: Performs a thread dump for debugging.
* **Configuration and Customization:**
  + Endpoints can be enabled or disabled and exposed or hidden through configuration properties in application.properties or application.yml.
  + Custom Actuator endpoints can be created to expose specific application-level information or functionality.

Enabling and Configuration:

To enable Spring Boot Actuator, add the spring-boot-starter-actuator dependency to your project's build file (e.g., pom.xml for Maven or build.gradle for Gradle).

Code

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

Endpoints are typically accessed via /actuator/{endpoint-id} (e.g., /actuator/health). Exposure of endpoints can be controlled using properties like management.endpoints.web.exposure.include and management.endpoints.web.exposure.exclude.

Security Considerations:

Some Actuator endpoints can expose sensitive information. It is crucial to secure these endpoints, especially in production environments, by placing them behind a firewall or integrating with security mechanisms like Spring Security. Careful consideration of which endpoints to expose and under what security constraints is essential.

Prometheus & Grafana (Helps to Visualize application health metrics) – Loki

Observability – Monitoring the various status of the Running Spring Boot Application

Prometheus & Grafana. – Prometheus converts the Application metrics to Time-series data. Grafana helps to visualize the Prometheus Data. (Dashboard)

Centralized Monitoring. (Micro-Service Environment) [Monitoring the Health info of running services] – Prometheus, Grafana & Loki.

Logging – Providing running details of currently executing code. It contains DateTimeStamp

Logging in Java involves recording events and messages generated by an application during its execution. This is crucial for debugging, monitoring, and understanding application behavior, especially in production environments.

Log Level = INFO, WARN, ERROR, DEBUG

Key Concepts:

* **Logging Frameworks:**

Java provides a built-in logging framework (java.util.logging), but widely used third-party alternatives like Log4j, Logback, and tinylog offer more features and flexibility.

* **Logging Abstractions:**

Frameworks like SLF4J (Simple Logging Facade for Java) and Apache Commons Logging provide an abstraction layer, allowing you to switch between different logging implementations without modifying your application code.

* **Loggers:**

These are the objects responsible for generating log events. They are typically associated with a class or package and are configured with a specific logging level.

* **Logging Levels:**

Log messages are categorized by severity using levels such as SEVERE, WARNING, INFO, CONFIG, FINE, FINER, and FINEST (in java.util.logging), or ERROR, WARN, INFO, DEBUG, and TRACE (in frameworks like Log4j).

* **Appenders/Handlers:**

These components determine where log messages are outputted. Common appenders include ConsoleHandler (for console output) and FileHandler (for writing to files). Custom appenders can also be created.