Report on Spring Boot Actuator and Metrics Integration

Amo Samuel

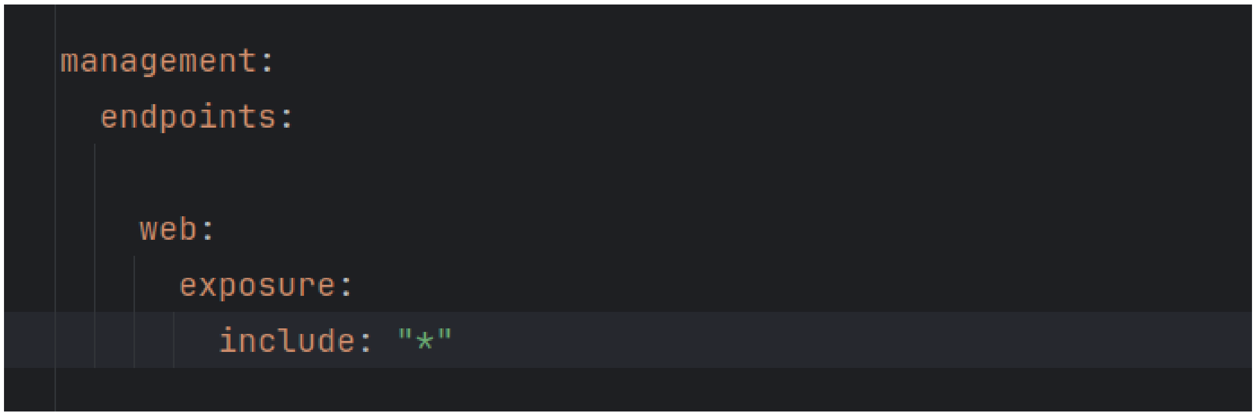
For this project I worked on building a Spring Boot application where I integrated **Spring Boot Actuator** to expose various management and monitoring endpoints. Actuator is a powerful Spring Boot feature that helps monitor and manage application health, metrics, and other operational data.

Key Features Implemented

Spring Boot Actuator Setup:

I began by setting up Spring Boot Actuator within the application by adding the spring-boot-starter-actuator dependency. This allows the app to expose endpoints for gathering essential data such as health status, application metrics, and environment details.
The following configuration was added to the application.yml file to enable Actuator and expose all available endpoints:

This setting ensures that all actuator endpoints are exposed, making the necessary metrics and data available.



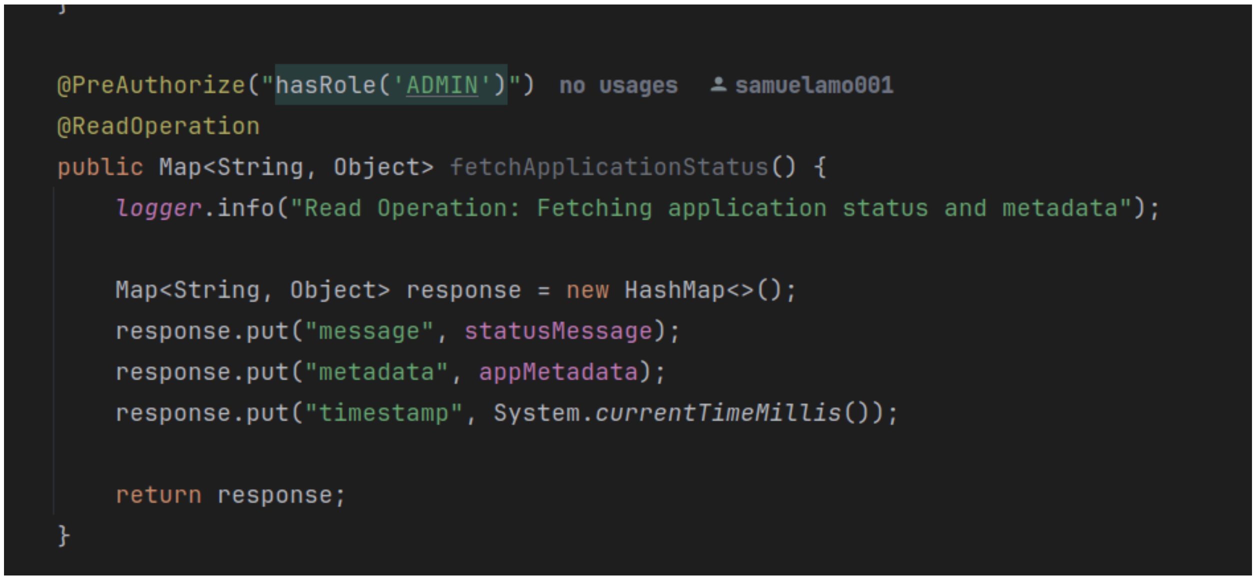
```yaml
management:
  endpoints:

    web:
      exposure:
        include: "*"
```

Custom Application Status Endpoint:

To provide a more personalized view of the application's status, I created a custom endpoint called application-status by leveraging Actuator's @Endpoint annotation. This endpoint can be accessed only by users with an "ADMIN" role, ensuring that sensitive information remains protected.

Here's an excerpt of the ApplicationStatusEndpoint class:

The custom endpoint provides metadata like application version, status message, and a timestamp.

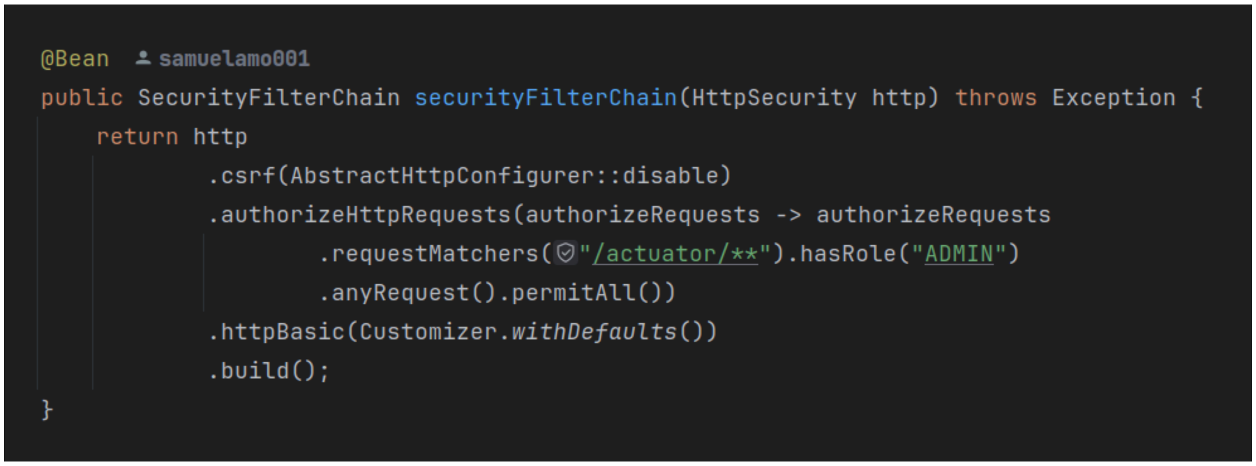


```
@PreAuthorize("hasRole('ADMIN')")  no usages   samuelamo001
@ReadOperation
public Map<String, Object> fetchApplicationStatus() {
    logger.info("Read Operation: Fetching application status and metadata");

    Map<String, Object> response = new HashMap<>();
    response.put("message", statusMessage);
    response.put("metadata", appMetadata);
    response.put("timestamp", System.currentTimeMillis());

    return response;
}
```

Security Configuration:
To secure the Actuator endpoints, I implemented **Spring Security**. Specifically, I restricted access to /actuator/** so that only users with an ADMIN role can view or modify application statuses. I used an in-memory user details manager for simplicity in this setup.

Here is the security configuration:

This setup ensures that monitoring and management endpoints are protected, which is a best practice when exposing such sensitive operational data.



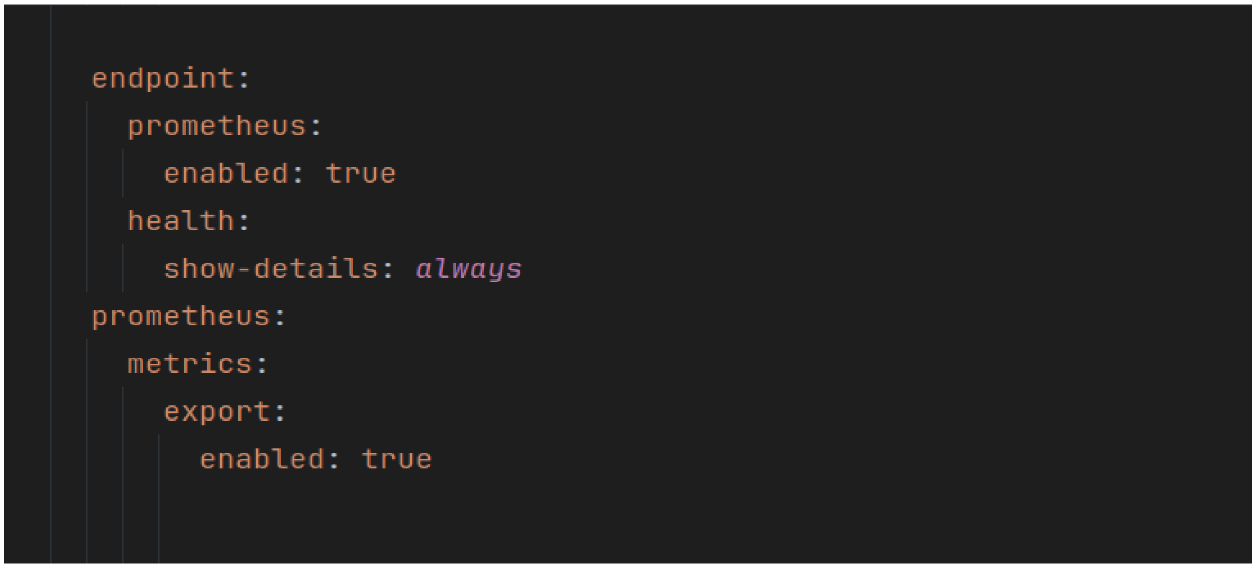
```
@Bean   samuelamo001
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    return http
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests(authorizeRequests -> authorizeRequests
                    .requestMatchers("/actuator/**").hasRole("ADMIN")
                    .anyRequest().permitAll())
            .httpBasic(Customizer.withDefaults())
            .build();
}
```

Metrics and Monitoring Integration

In addition to Actuator, I integrated **Micrometer** and **Prometheus** for in-depth application monitoring. These tools are crucial for observing application performance and providing detailed metrics.

Micrometer Setup:
Micrometer was integrated as the metrics facade, allowing me to collect application-specific metrics. By adding the micrometer-core dependency and configuring prometheus as the metrics registry, I ensured that all metrics collected by Micrometer would be available for Prometheus. The configuration to enable Prometheus metrics in application.yml is as follows:

This makes the application export its metrics data in a format that Prometheus can scrape, enabling me to monitor various operational metrics in real-time.

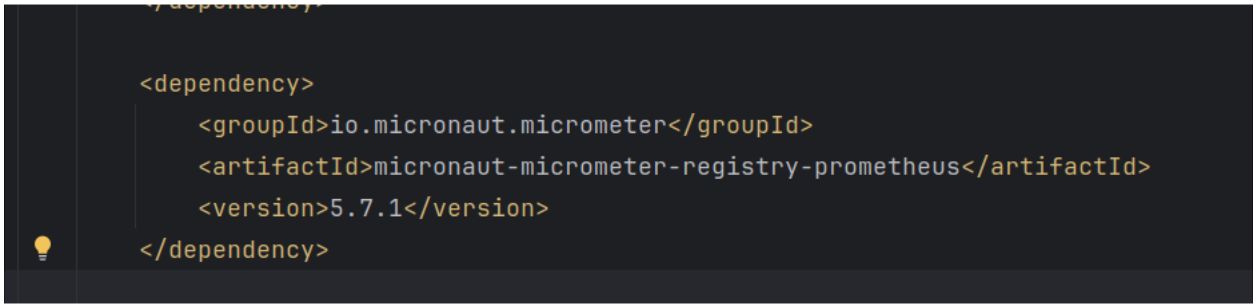


```
endpoint:
  prometheus:
    enabled: true
  health:
    show-details: always
prometheus:
  metrics:
    export:
      enabled: true
```

Prometheus Integration:
Prometheus was integrated by exposing a /actuator/prometheus endpoint that serves application metrics in a format understood by Prometheus. These metrics include HTTP request counts, memory usage, garbage collection statistics, and custom metrics defined using Micrometer.

To set this up, I simply added the required dependency:

With the Prometheus server configured to scrape the metrics endpoint, I was able to view key performance metrics from the application in the Prometheus dashboard. This integration ensures that I have continuous insights into the system's health, allowing for proactive scaling or debugging if necessary.



```
</dependency>

<dependency>
    <groupId>io.micronaut.micrometer</groupId>
    <artifactId>micronaut-micrometer-registry-prometheus</artifactId>
    <version>5.7.1</version>
</dependency>
```

Best Practices with Spring Boot Actuator and Metrics

While integrating Spring Boot Actuator, Micrometer, and Prometheus, I followed several best practices:

- **Security of Endpoints**:
  Ensuring that only authorized users can access critical Actuator endpoints is paramount. By configuring role-based access control, I minimized the risk of exposing sensitive operational information to unauthorized users.
- **Fine-tuning Metrics**:
  Using the Micrometer, I ensured that only relevant and necessary metrics were collected. It's important to avoid metric over-collection, which can lead to unnecessary storage usage and processing overhead.
- **Centralized Monitoring with Prometheus**:

  Exposing Prometheus metrics via the /actuator/prometheus endpoint is essential for a centralized monitoring system. This allows not just the application, but all related services to be observed from a single Prometheus instance.

Conclusion

In this phase of the project, I have successfully set up Spring Boot Actuator along with Prometheus and Micrometer for monitoring. By integrating these tools, I can efficiently manage and monitor the application's performance and health. Moving forward, I plan to continue improving on this setup, adding more custom metrics and alerts to ensure robust application monitoring and faster response times in case of performance issues.