Spring Actuator Features and Best Practices

I explored the powerful capabilities of **Spring Actuator** for monitoring Spring Boot applications. By the end of the exercises, I understood how to enable Actuator, utilize its endpoints for gathering health and metrics information, and customize the behavior of the Actuator. Additionally, I implemented security measures for protecting sensitive information and explored potential integration with external monitoring tools.

1. Enabling Spring Actuator

The first step was to enable Spring Actuator by adding the necessary dependencies in the `pom.xml`:

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

After this, I configured the Actuator to expose all its endpoints using the `application.yml` file. This configuration ensured I could access key metrics and health data for the application:

```
management:
  endpoints:
    web:
      exposure:
        include: "*"
  security:
    enabled: true
  endpoint:
    health:
      show-details: always
```

2. Exploring Default Endpoints

Spring Actuator provides several **default endpoints** to monitor the application. I explored the following:

- **/actuator/health:** This endpoint checks the health of the application and its dependencies.
- **/actuator/metrics:** This endpoint retrieves various metrics like memory usage, active threads, and CPU load.
- **/actuator/env:** This provides details about the environment properties of the application.

These default endpoints made it easy for me to monitor the application without writing additional code.

3. Custom Actuator Endpoints

To demonstrate customization, I created a custom Actuator endpoint named **CustomEndpoint** to expose additional application information. The custom endpoint provided status information and allowed updates through HTTP methods (`GET`, `POST`, `DELETE`):

```
@Component
@Endpoint(id = "custom")
public class CustomEndpoint {

    private String status = "OK";

    @ReadOperation
    public CustomInfo getInfo() {
        return new CustomInfo(status);
    }

    @WriteOperation
    public void updateStatus(String newStatus) {
        this.status = newStatus;
    }

    @DeleteOperation
    public void deleteStatus() {
        this.status = null;
    }

    public static class CustomInfo {
```

```
        private String status;

        public CustomInfo(String status) {
            this.status = status;
        }

        public String getStatus() {
            return status;
        }
    }
}
```

By using this approach, I was able to expose application-specific data, which could be useful in a production environment.

4. Securing Actuator Endpoints

Given that Actuator exposes sensitive application data, securing these endpoints is crucial. I used **Spring Security** to protect Actuator endpoints by restricting access to users with the `ADMIN` role.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
        http.csrf(AbstractHttpConfigurer::disable);
        http.sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
        http.authorizeHttpRequests((requests)
                -> requests
                .requestMatchers("/actuator/**").hasRole("ADMIN")
                .anyRequest().permitAll());
        http.httpBasic(Customizer.withDefaults());
        return http.build();
    }

    @Bean
    public UserDetailsService userDetailsService(DataSource dataSource) {
        JdbcUserDetailsManager manager = new JdbcUserDetailsManager(dataSource);
        // Create sample users
        if (!manager.userExists("admin")) {
            manager.createUser(User.withUsername("admin")
                    .password(passwordEncoder().encode("adminPass"))
                    .roles("ADMIN")
                    .build());
        }
        return manager;
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
```

```
        return new BCryptPasswordEncoder();
    }
}
```

This setup allowed me to restrict access to the Actuator endpoints and ensured that only authenticated and authorized users could view or modify sensitive information.

I created a **GitHub repository** containing the Spring Boot application with the following features:

1. **Enabled Actuator endpoints**: All default endpoints like `/health`, `/metrics`, and `/env` are enabled.
2. **Custom Actuator endpoint**: The `/actuator/custom` endpoint provides status information and can be modified via HTTP requests.
3. **Secured Actuator endpoints**: The Actuator endpoints are protected, and only users with the `ADMIN` role can access them.

Conclusion

This lab was an insightful exploration of Spring Actuator's capabilities. I learned how to monitor applications, create custom endpoints, and secure sensitive data. Integrating Actuator into a Spring Boot application is essential for ensuring that the application remains healthy, observable, and secure, especially in a production environment.