Spring Boot Actuator: A Comprehensive Guide

Trainer name:

Shreyansh Kumar

Introduction

In the realm of microservices and distributed systems, monitoring and managing applications becomes increasingly complex. Spring Boot Actuator addresses this challenge by providing production-ready features that help you monitor and manage your application. This document explores Spring Boot Actuator in depth, covering its endpoints, implementation, and customization options.

Spring Boot Actuator is essentially a sub-project of Spring Boot that adds several production-grade services to your application with little developer effort. It offers features like health checks, metrics gathering, HTTP tracing, and more out of the box. These features are crucial for any application running in production, making Actuator an indispensable tool in the Spring Boot ecosystem.

Understanding Spring Boot Actuator

Spring Boot Actuator provides HTTP endpoints or JMX beans to monitor and manage your application. It gives you insights into what's happening inside a running application. With Actuator, you can check the health of your application, view metrics, understand traffic patterns, and more.

The beauty of Actuator lies in its simplicity. With minimal configuration, you get a wealth of features that would otherwise require significant development effort. This allows developers to focus on building business logic while still having robust monitoring capabilities.

Adding Actuator to Your Spring Boot Project

Adding Actuator to your Spring Boot project is straightforward. You simply need to include the appropriate dependency in your build file.

For Maven projects, add the following to your pom.xml:

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

For Gradle projects, add this to your build.gradle:

```groovy
implementation 'org.springframework.boot:spring-boot-starter-actuator'
```

Once you've added the dependency, Spring Boot automatically configures Actuator with sensible defaults. By default, most endpoints except for `/health` are disabled for security reasons. You can enable them in your application properties.

Configuring Actuator Endpoints

After adding the Actuator dependency, you'll want to configure which endpoints are exposed and how they're secured. This is done through your application.properties or application.yml file.

To enable all endpoints over HTTP:

```
management.endpoints.web.exposure.include=*
```

To enable specific endpoints:

```
management.endpoints.web.exposure.include=health,info,metrics
```

To exclude specific endpoints:

```
management.endpoints.web.exposure.exclude=env,beans
```

You can also change the base path for Actuator endpoints (default is `/actuator`):

```
management.endpoints.web.base-path=/management
```

For security reasons, you might want to configure a different port for Actuator endpoints:

```
management.server.port=8081
```

Core Actuator Endpoints

Spring Boot Actuator provides numerous endpoints. Here are the most important ones:

Endpoint Name	Description	Example URL
Health	Provides health status of the application.	/actuator/health
Info	Displays arbitrary application information.	/actuator/info
Metrics	Provides various application metrics.	/actuator/metrics
Environment (Env)	Exposes environment properties, including application properties & system vars.	/actuator/env
Loggers	Shows and modifies logger configurations at runtime.	/actuator/loggers
Thread Dump	Performs a thread dump of the JVM.	/actuator/threaddump
Heap Dump	Generates a heap dump for memory analysis.	/actuator/heapdump
Shutdown	Allows the application to be gracefully shut down (disabled by default).	/actuator/shutdown
Mappings	Displays all request mappings in the application.	/actuator/mappings
Beans	Shows all Spring beans in the application context.	/actuator/beans
Conditions	Displays evaluated conditions for auto-configuration.	/actuator/conditions
Config Props	Shows all @ConfigurationProperties beans.	/actuator/configprops
Scheduled Tasks	Displays the scheduled tasks in the application.	/actuator/scheduledtasks

HTTP Trace	Displays HTTP trace information (last 100 requests by default).	/actuator/httptrace
Audit Events	Shows audit events generated by the application.	/actuator/auditevents
Caches	Displays all available caches.	/actuator/caches
Integration Graph	Shows the Spring Integration graph.	/actuator/integrationgraph
Custom Endpoints	Developers can define their own actuator endpoints.	/actuator/custom

Health Endpoint (/actuator/health)

The health endpoint provides basic health information about your application. It's often used by monitoring tools to check if the application is up and running.

```
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class CustomHealthIndicator implements HealthIndicator {
    @Override
    public Health health() {
        int errorCode = checkService(); // perform some specific health check
        if (errorCode != 0) {
            return Health.down().withDetail("Error Code", errorCode).build();
        }
        return Health.up().build();
    }

    private int checkService() {
        // Logic to check service health
        return 0;
    }
}
```

Info Endpoint (/actuator/info)

The info endpoint displays arbitrary application information. You can configure static information in your properties file:

```
info.app.name=My Spring Application
info.app.description=A demo Spring Boot application
info.app.version=1.0.0
```

Or you can provide dynamic information through an InfoContributor:

```
import org.springframework.boot.actuate.info.Info;
import org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;
```

```java
import java.util.HashMap;
import java.util.Map;

@Component
public class CustomInfoContributor implements InfoContributor {
    @Override
    public void contribute(Info.Builder builder) {
        Map<String, Object> details = new HashMap<>();
        details.put("serverTime", System.currentTimeMillis());
        builder.withDetail("runtime", details);
    }
}
```

Metrics Endpoint (/actuator/metrics)

The metrics endpoint provides metrics information for your application. Spring Boot Actuator uses Micrometer, which provides a facade over various monitoring systems.

```java
import io.micrometer.core.instrument.Counter;
import io.micrometer.core.instrument.MeterRegistry;
import org.springframework.stereotype.Service;

@Service
public class MyService {
    private final Counter counter;

    public MyService(MeterRegistry registry) {
        this.counter = registry.counter("service.invocations");
    }

    public void doSomething() {
        counter.increment();
        // business logic
    }
}
```

Environment Endpoint (/actuator/env)

The environment endpoint exposes properties from Spring's ConfigurableEnvironment. This includes environment variables, JVM properties, application properties, and more.

Loggers Endpoint (/actuator/loggers)

The loggers endpoint shows and modifies the configuration of loggers in your application. You can view the current log levels and change them at runtime.

Threaddump Endpoint (/actuator/threaddump)

The threaddump endpoint performs a thread dump of the application's JVM.

Heapdump Endpoint (/actuator/heapdump)

The heapdump endpoint creates a heap dump of the JVM used by your application.

Shutdown Endpoint (/actuator/shutdown)

The shutdown endpoint allows the application to be gracefully shut down. This endpoint is disabled by default for security reasons.

```
management.endpoint.shutdown.enabled=true
```

Mappings Endpoint (/actuator/mappings)

The mappings endpoint displays all @RequestMapping paths in your application.

Beans Endpoint (/actuator/beans)

The beans endpoint displays all the Spring beans in your application's context.

Conditions Endpoint (/actuator/conditions)

The conditions endpoint shows the conditions that were evaluated on configuration and auto-configuration classes.

Configprops Endpoint (/actuator/configprops)

The configprops endpoint displays all @ConfigurationProperties.

Scheduledtasks Endpoint (/actuator/scheduledtasks)

The scheduledtasks endpoint displays the scheduled tasks in your application.

Httptrace Endpoint (/actuator/httptrace)

The httptrace endpoint displays HTTP trace information (by default, the last 100 HTTP request-response exchanges). Note that in newer versions of Spring Boot, you need to provide your own HttpTraceRepository bean:

```
import org.springframework.boot.actuate.trace.http.HttpTraceRepository;
import
org.springframework.boot.actuate.trace.http.InMemoryHttpTraceRepository;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class HttpTraceActuatorConfiguration {
    @Bean
    public HttpTraceRepository httpTraceRepository() {
        return new InMemoryHttpTraceRepository();
    }
}
```

Auditevents Endpoint (/actuator/auditevents)

The auditevents endpoint displays audit events for your application.

Caches Endpoint (/actuator/caches)

The caches endpoint exposes available caches.

Integrations Endpoint (/actuator/integrationgraph)

The integrations endpoint shows the Spring Integration graph.

Creating Custom Actuator Endpoints

Spring Boot Actuator allows you to create custom endpoints to expose application-specific information or operations.

```java
import org.springframework.boot.actuate.endpoint.annotation.Endpoint;
import org.springframework.boot.actuate.endpoint.annotation.ReadOperation;
import org.springframework.boot.actuate.endpoint.annotation.WriteOperation;
import org.springframework.stereotype.Component;

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

    private String status = "OK";

    @ReadOperation
    public CustomData getData() {
        return new CustomData(status);
    }

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

    public static class CustomData {
        private final String status;

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

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

This creates a new endpoint at `/actuator/custom` that supports both GET and POST operations.

Securing Actuator Endpoints

Actuator endpoints often expose sensitive information, so it's important to secure them properly. If you're using Spring Security, you can configure it to protect your Actuator endpoints:

```
import
org.springframework.boot.actuate.autoconfigure.security.servlet.EndpointReque
st;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSec
urity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class ActuatorSecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .requestMatcher(EndpointRequest.toAnyEndpoint())
            .authorizeRequests()
                .requestMatchers(EndpointRequest.to("health",
"info")).permitAll()
                .anyRequest().hasRole("ACTUATOR")
            .and()
            .httpBasic();
        return http.build();
    }
}
```

This configuration allows anonymous access to the health and info endpoints but requires ACTUATOR role for all other endpoints.

Customizing Health Indicators

Health indicators are components that contribute to the overall health status of your application. Spring Boot provides several built-in health indicators for databases, disk space, and more. You can also create custom health indicators:

```
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
```

```java
import org.springframework.stereotype.Component;

@Component
public class CustomHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        boolean healthy = checkExternalService();

        if (healthy) {
            return Health.up()
                    .withDetail("externalService", "Available")
                    .build();
        } else {
            return Health.down()
                    .withDetail("externalService", "Unavailable")
                    .build();
        }
    }

    private boolean checkExternalService() {
        // Logic to check if external service is available
        return true;
    }
}
```

Customizing Metrics

Spring Boot Actuator uses Micrometer for metrics collection. Micrometer provides a vendor-neutral facade for the most popular monitoring systems. You can customize metrics collection:

```java
import io.micrometer.core.instrument.Counter;
import io.micrometer.core.instrument.MeterRegistry;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;

@Component
public class CustomMetrics {

    private final MeterRegistry meterRegistry;

    public CustomMetrics(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
    }

    @PostConstruct
    public void init() {
        Counter.builder("custom.metric")
```

```
                .description("A custom metric")
                .tag("region", "us-east")
                .register(meterRegistry);
    }

    public void incrementCustomMetric() {
        meterRegistry.counter("custom.metric", "region", "us-
east").increment();
    }
}
```

Integrating with Monitoring Systems

One of the strengths of Spring Boot Actuator is its ability to integrate with various monitoring systems. Here are some examples:

Prometheus

To integrate with Prometheus, add the Micrometer Prometheus registry:

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

This automatically adds a `/actuator/prometheus` endpoint that exposes metrics in a format that Prometheus can scrape.

Grafana

Grafana can visualize metrics from various sources, including Prometheus. Once you've set up Prometheus to scrape your application, you can create dashboards in Grafana to visualize the data.

Elastic Stack

For integration with the Elastic Stack (Elasticsearch, Logstash, Kibana), you can use Logstash to collect logs and metrics from your application and store them in Elasticsearch. Kibana can then be used to visualize the data.

Advanced Actuator Configurations

Customizing Endpoint IDs

You can customize the IDs of Actuator endpoints:

```
management.endpoints.web.path-mapping.health=healthcheck
```

This changes the health endpoint from `/actuator/health` to `/actuator/healthcheck`.

Enabling JMX Endpoints

By default, JMX endpoints are enabled but HTTP endpoints are mostly disabled. You can configure which JMX endpoints are exposed:

```
management.endpoints.jmx.exposure.include=health,info
```

CORS Configuration

If your monitoring frontend is hosted on a different domain, you might need to configure CORS:

```
management.endpoints.web.cors.allowed-origins=https://example.com
management.endpoints.web.cors.allowed-methods=GET,POST
```

Customizing the Actuator Base Path

You can change the base path for Actuator endpoints:

```
management.endpoints.web.base-path=/admin
```

This changes the base path from `/actuator` to `/admin`.

Best Practices for Using Actuator in Production

When using Spring Boot Actuator in production, consider the following best practices:

1. **Secure your endpoints**: Only expose the endpoints you need and secure them appropriately.
2. **Monitor the right metrics**: Focus on metrics that provide actionable insights.
3. **Set up alerts**: Configure alerts for critical metrics to be notified of issues.
4. **Use distributed tracing**: For microservices architectures, consider using distributed tracing tools like Spring Cloud Sleuth and Zipkin.
5. **Regularly review health checks**: Ensure your health checks accurately reflect the health of your application.
6. **Implement custom health indicators**: Create custom health indicators for external dependencies.
7. **Configure appropriate logging levels**: Use the loggers endpoint to adjust logging levels as needed.
8. **Backup before using destructive endpoints**: Be cautious with endpoints like shutdown.
9. **Consider rate limiting**: Protect your application from excessive Actuator requests.
10. **Document your custom endpoints**: Ensure your team knows what custom endpoints are available and what they do.

Troubleshooting Common Actuator Issues

Endpoints Not Available

If endpoints are not available, check:

- Is the Actuator dependency included?
- Are the endpoints enabled in your configuration?
- Are the endpoints exposed over HTTP?

Security Issues

If you're having security issues:

- Check your Spring Security configuration
- Ensure you're using the correct authentication credentials
- Verify that your security rules are correctly defined

Performance Concerns

If Actuator is impacting performance:

- Consider disabling unused endpoints
- Use sampling for metrics collection
- Configure appropriate caching for endpoints

Custom Health Indicators Not Working

If custom health indicators aren't working:

- Ensure they're properly annotated and in the component scan path
- Check for exceptions in your health check logic
- Verify that the health endpoint is correctly configured

Conclusion

Spring Boot Actuator is a powerful tool for monitoring and managing your Spring Boot applications. With minimal configuration, it provides a wealth of features that help you understand what's happening inside your application in production.

By leveraging Actuator's endpoints, you can gain insights into application health, performance metrics, environment configuration, and more. Custom endpoints and health indicators allow you to extend Actuator's capabilities to meet your specific needs.

When properly secured and configured, Actuator becomes an indispensable part of your production environment, helping you ensure the reliability and performance of your Spring Boot applications.

Remember that while Actuator provides many features out of the box, it's important to customize it to your specific requirements and security needs. With the knowledge gained from this document, you should be well-equipped to implement and configure Spring Boot Actuator in your applications.