Spring Boot Actuators

Introduction

Spring Boot Actuator provides production-ready features to monitor and manage Spring Boot applications. For example, in a production environment, Actuator can be integrated with Prometheus and Grafana to collect and visualize application metrics, helping teams track performance trends, diagnose issues, and ensure system reliability. It offers various tools to gain insights into application performance, system health, and environment properties, helping developers and operators maintain, debug, and optimize their applications.

Key Features

1. Health Monitoring: Provides health checks for the application and its components.
2. Metrics Collection: Tracks metrics such as memory usage, CPU usage, and request statistics.
3. Application Info: Exposes details like application version, build information, and custom metadata.
4. Environment Properties: Displays environment variables, configuration properties, and profiles.
5. Loggers Management: Allows runtime management of log levels.
6. HTTP Trace: Captures request/response traces.

Common Use Cases

1. Application Monitoring: Integrate with monitoring tools like Prometheus, Grafana, or New Relic.
2. Health Checks: Ensure application readiness in orchestrated environments like Kubernetes.
3. Debugging Issues: Quickly diagnose configuration or runtime issues.
4. Metrics Analysis: Evaluate application performance under load.
5. Dynamic Configuration: Adjust logging levels without redeploying.

Actuator Endpoints

Spring Boot Actuator exposes endpoints that provide insights into the application's state.

Default Endpoints

Spring Boot Actuator provides several built-in endpoints that help in monitoring and managing the state of an application. These endpoints offer valuable insights into application health, metrics, and configuration, making them indispensable for production support and troubleshooting.

1. **`/actuator/health`**:
   - Request: No parameters required.
   - Note: Additional health details can be customized with custom health indicators. For example, you can add database connection or cache service checks by implementing the `HealthIndicator` interface in your application. read here
   - Response: Displays the health status of the application and its dependencies. Example:

     ```
     {
         "status": "UP",
         "components": {
             "db": {
                 "status": "UP",
                 "details": {
                     "database": "PostgreSQL",
                     "validationQuery": "SELECT 1"
                 }
             },
             "diskSpace": {
                 "status": "UP",
                 "details": {
                     "total": 499963174912,
                     "free": 2342236160,
                     "threshold": 10485760
                 }
             }
     ```

```
        }
    }
```

2. `/actuator/metrics`:
   - Request: Optionally accepts a `tag` query parameter to filter metrics.
   - Response: Provides a list of available metrics or specific metric details. Example:

     ```json
     {
         "names": ["jvm.memory.used", "jvm.memory.max"]
     }
     ```

     For specific metrics:

     ```json
     {
         "name": "jvm.memory.used",
         "measurements": [
             {
                 "statistic": "VALUE",
                 "value": 25149456
             }
         ],
         "availableTags": [
             {
                 "tag": "area",
                 "values": ["heap", "nonheap"]
             }
         ]
     }
     ```

3. `/actuator/info`:
   - Request: No parameters required.
   - Response: Displays custom application information defined in `application.properties` or `build-info.properties`. Example:

     ```json
     {
         "app": {
     ```

```
        "name": "Demo Application",
        "version": "1.0.0"
    }
}
```

4. `/actuator/env`:
   - Request: No parameters or a specific property key as a query.
   - Response: Displays environment properties, including system properties and configuration files. Example:

```
{
    "activeProfiles": ["prod"],
    "propertySources": [
        {
            "name": "applicationConfig:
[classpath:/application.properties]",
            "properties": {
                "server.port": {
                    "value": "8080"
                }
            }
        }
    ]
}
```

5. `/actuator/loggers`:
   - Request: Optionally accepts a logger name as a path variable.
   - Response: Retrieves or updates the logging level of loggers. Example:

```
{
    "loggers": {
        "root": {
            "configuredLevel": "INFO",
            "effectiveLevel": "INFO"
        }
    }
}
```

6. `/actuator/httptrace`:
   - Request: No parameters required.
   - Response: Shows the most recent HTTP requests and responses. Example:

```
{
    "traces": [
        {
            "timestamp": "2024-12-25T12:00:00.000Z",
            "request": {
                "method": "GET",
                "uri": "/api/demo",
                "headers": {
                    "host": "localhost:8080"
                }
            },
            "response": {
                "status": 200,
                "headers": {
                    "content-length": "0"
                }
            }
        }
    ]
}
```

7. `/actuator/beans`:
   - Request: No parameters required.
   - Response: Lists all Spring beans in the application context, including their scope and dependencies. Example:

```
{
    "beans": [
        {
            "bean": "exampleBean",
            "scope": "singleton",
                "type": "com.example.DemoBean"
            }
        ]
}
```

8. `/actuator/configprops`:
    - Request: No parameters required.
    - Response: Displays configuration properties and their values. Example:

```
{
    "contexts": {
        "application": {
            "beans": {
                "dataSource": {
                    "prefix": "spring.datasource",
                    "properties": {
                        "url":
"jdbc:mysql://localhost:3306/demo",
                        "username": "root"
                    }
                }
            }
        }
    }
}
```

9. `/actuator/threaddump`:
    - Request: No parameters required.
    - Response: Dumps all threads' stack traces. Example:

```
{
    "threads": [
        {
            "threadName": "main",
            "threadState": "RUNNABLE",
            "stackTrace": [

"com.example.DemoApplication.main(DemoApplication.jav
a:10)"
            ]
        }
    ]
}
```

10. `/actuator/mappings`:
    - Request: No parameters required.
    - Response: Displays all request-to-handler mappings. Example:

    ```
    {
        "mappings": [
            {
                "handler": "demoController",
                "methods": ["GET"],
                "pattern": "/api/demo"
            }
        ]
    }
    ```

Custom Endpoints

You can define your own endpoints by implementing the `@Endpoint` annotation.

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

    @ReadOperation
    public String customEndpoint() {
        return "Custom Endpoint Response";
    }
}
```

Configurations

Enabling Actuator

Add the following dependency to your `pom.xml`:

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

Security Configurations

Actuator endpoints can expose sensitive data, so secure them adequately.

- Use Spring Security to protect endpoints.
- Limit access based on roles or IP ranges.

Example of securing endpoints in a cloud environment (e.g., AWS):

In AWS, you can use an API Gateway in front of your application to provide an extra layer of security. Configure the API Gateway to expose only specific Actuator endpoints and integrate it with AWS IAM or Cognito for authentication.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()

.requestMatchers(EndpointRequest.to(HealthEndpoint.class,
InfoEndpoint.class)).permitAll()

.requestMatchers(EndpointRequest.toAnyEndpoint()).hasRole("ADM
IN")
        .and()
        .httpBasic();
}
```

Additionally, restrict access using security groups and network ACLs in AWS to ensure only trusted IP ranges can access the endpoints. This setup helps prevent unauthorized access and secures sensitive application metrics and configurations.

Actuator endpoints can expose sensitive data, so secure them adequately.

- Use Spring Security to protect endpoints.
- Limit access based on roles or IP ranges.
  Example:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()

.requestMatchers(EndpointRequest.to(HealthEndpoint.class,
InfoEndpoint.class)).permitAll()

.requestMatchers(EndpointRequest.toAnyEndpoint()).hasRole("ADM
IN")
        .and()
        .httpBasic();
}
```

Customizing Endpoints

In `application.properties` or `application.yml`:

For example, using `application.yml`:

```
management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics
  endpoint:
    health:
      show-details: always
```

or `application.properties`

```
management.endpoints.web.exposure.include=health,info,metrics
management.endpoint.health.show-details=always
```

Production Recommendations

- Expose only necessary endpoints.
- Use HTTPS for secure communication.
- Integrate with monitoring systems.
- Enable authentication and authorization for all endpoints.

Best Practices

1. **Restrict Exposure**: Only expose required endpoints in production. This can be achieved effectively by leveraging network-level firewalls to block unauthorized access and API gateways to control and authenticate traffic. For instance, configure your cloud provider's security settings to allow access only from specific IP ranges or trusted networks.
2. **Use Filters**: Exclude sensitive information using `management.endpoint.<name>.enabled` flags.
3. **Monitor Effectively**: Integrate with APM tools for automated monitoring.
4. **Health Groups**: Define custom health indicators using `HealthIndicator`.
5. **Performance**: Evaluate the performance impact of enabled metrics and traces.