Spring Boot Actuator

1. Introduction to Spring Boot Actuator

Spring Boot Actuator is a powerful module that provides production-ready features to help you monitor and manage your Spring Boot application. It offers a range of built-in endpoints and capabilities that give insights into the internal workings of your application, making it easier to observe and interact with your running application.

1.1 Key Benefits

- **Production-Ready Features**: Provides essential monitoring and management features out of the box.
- **Insights into Application Health**: Offers health, metrics, and performance data.
- **Ease of Configuration**: Enables monitoring and management with minimal setup.
- **Customizable**: Allows customization and extension of monitoring capabilities.

---

2. Getting Started

2.1 Dependency Configuration

To add Spring Boot Actuator to your project, include the following dependency:

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

Gradle
```
dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-actuator'
}
```

2.2 Basic Configuration

By default, Actuator exposes a few endpoints over HTTP. To enable all endpoints, add the following to your application.properties or application.yml:

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

---

3. Built-in Endpoints

Spring Boot Actuator provides several out-of-the-box endpoints:

3.1 Health Endpoint

- **URL**: /actuator/health
- Provides application health information.
- Configurable to show detailed health checks.

Example Health Response:

```
{
  "status": "UP",
  "components": {
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": "500GB",
        "free": "250GB"
      }
    },
    "db": {
      "status": "UP"
    }
  }
}
```

3.2 Info Endpoint

- **URL**: /actuator/info
- Displays arbitrary application information.
- Configurable through application.properties.

Example Configuration:

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

3.3 Metrics Endpoint

- **URL**: /actuator/metrics
- Provides detailed metrics about the application.
- Supports JVM, system, and application-specific metrics.

3.4 Logging Endpoint

- **URL**: /actuator/loggers
- Allows runtime log level configuration.
- Supports viewing and modifying log levels.

3.5 Shutdown Endpoint

- **URL**: /actuator/shutdown
- Gracefully shuts down the application.
- Disabled by default and must be explicitly enabled.

---

4. Security Considerations

4.1 Securing Actuator Endpoints

To secure actuator endpoints, add the following dependency:

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

4.2 Endpoint-Level Security

You can configure security at the endpoint level as follows:

```
@Configuration
public class ActuatorSecurityConfig {
  @Bean
  public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
      .authorizeHttpRequests()
      .requestMatchers("/actuator/health", "/actuator/info")
        .permitAll()
      .requestMatchers("/actuator/**")
        .hasRole("ADMIN")
      .and()
      .httpBasic();
```

```
        return http.build();
    }
}
```

---

5. Custom Endpoints

5.1 Creating a Custom Endpoint

You can create custom endpoints using the @Endpoint annotation:

```
@Component
@Endpoint(id = "custom")
public class CustomActuatorEndpoint {
    @ReadOperation
    public Map<String, Object> customEndpoint() {
        return Map.of(
            "key1", "value1",
            "key2", "value2"
        );
    }
}
```

---

6. Performance Monitoring

6.1 Micrometer Integration

Spring Boot Actuator integrates with Micrometer for comprehensive monitoring. To enable it, add the following dependency:

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

---

7. Best Practices

1. **Minimal Exposure**: Only expose necessary endpoints in production.
2. **Secure Endpoints**: Always implement proper authentication for sensitive endpoints.

3. **Use Prometheus/Grafana**: For advanced monitoring and visualization.
4. **Custom Health Indicators**: Create application-specific health checks.
5. **Log Carefully**: Be mindful of sensitive information in logs.

---

8. Troubleshooting

8.1 Common Issues

- **Endpoint Not Visible**: Check the exposure configuration to ensure it is enabled.
- **Security Blocking Access**: Verify the security settings to ensure correct access control.
- **Performance Overhead**: Monitor the impact of metrics and health checks on performance.

---

9. Advanced Configuration

9.1 Custom Health Indicators

You can implement custom health indicators to check specific components of your application:

```
@Component
public class DatabaseHealthIndicator implements HealthIndicator {
    @Override
    public Health health() {
        if (isDatabaseHealthy()) {
            return Health.up().build();
        }
        return Health.down()
            .withDetail("Error", "Database connection failed")
            .build();
    }
}
```

Actuator REST API

1.1 Audit Events Endpoint (/actuator/auditevents)

- **Purpose**: Exposes audit events information
- **Use Cases**:
  - Track security-related events
  - Monitor user actions and system changes

Configuration Example:

```
management.endpoint.auditevents.enabled=true
```

Sample Response:

```
{
 "events": [
  {
   "timestamp": "2024-01-15T10:30:45.123Z",
   "principal": "admin",
   "type": "LOGIN_SUCCESS",
   "data": {
    "remoteAddress": "192.168.1.100"
   }
  }
 ]
}
```

Query Parameters

The endpoint uses query parameters to limit the events that it returns. The following table shows the supported query parameters:

Parameter	Description
after	Restricts the events to those that occurred after the given time. Optional.
principal	Restricts the events to those with the given principal. Optional.
type	Restricts the events to those with the given type. Optional.

Response Structure

The response contains details of all of the audit events that matched the query. The following table describes the structure of the response:

Path	Type	Description
events	Array	An array of audit events.
events.[].timestamp	String	The timestamp of when the event occurred.
events.[].principal	String	The principal that triggered the event.
events.[].type	String	The type of the event.

1.2 Beans Endpoint (/actuator/beans)

- **Purpose**: Displays all Spring beans in the application context
- **Use Cases**:
    - Inspect bean configurations
    - Debug dependency injection

Sample Response Snippet:

```
{
 "contexts": {
   "application": {
     "beans": {
       "userService": {
         "scope": "singleton",
         "type": "com.example.UserService",
         "dependencies": ["userRepository"]
       }
     }
   }
 }
}
```

Response Structure

The response contains details of the application’s beans. The following table describes the structure of the response:

Path	Type	Description
`contexts`	`Object`	Application contexts keyed by id.
`contexts.*.parentId`	`String`	Id of the parent application context, if any.
`contexts.*.beans`	`Object`	Beans in the application context keyed by name.
`contexts.*.beans.*.aliases`	`Array`	Names of any aliases.
`contexts.*.beans.*.scope`	`String`	Scope of the bean.
`contexts.*.beans.*.type`	`String`	Fully qualified type of the bean.
`contexts.*.beans.*.resource`	`String`	Resource in which the bean was defined, if any.
`contexts.*.beans.*.dependencies`	`Array`	Names of any dependencies.

1.3 Caches Endpoint (/actuator/caches)

- **Purpose**: Provides information about cache configurations
- **Features**:
  - List cache names
  - Display cache statistics

  Configuration:
  management.endpoint.caches.enabled=true

Retrieving All Caches

To retrieve the application’s caches, send a `GET` request to `/actuator/caches`. Example using `curl`:

```
$ curl 'http://localhost:8080/actuator/caches' -i -X GET
```

Response Example:

```
{
 "cacheManagers" : {
  "anotherCacheManager" : {
   "caches" : {
    "countries" : {
     "target" : "java.util.concurrent.ConcurrentHashMap"
    }
   }
  },
  "cacheManager" : {
   "caches" : {
    "cities" : {
     "target" : "java.util.concurrent.ConcurrentHashMap"
    },
    "countries" : {
     "target" : "java.util.concurrent.ConcurrentHashMap"
    }
   }
  }
 }
}
```

Response Structure:

Path	Type	Description
`cacheManagers`	Object	Cache managers keyed by ID
`cacheManagers.*.caches`	Object	Caches in the application context keyed by name
`cacheManagers.*.caches.*.target`	String	Fully qualified name of the native cache

Retrieving Caches by Name

To retrieve a specific cache, make a `GET` request to `/actuator/caches/{name}`. Example for cache named "cities":

```
$ curl 'http://localhost:8080/actuator/caches/cities' -i -X GET
```

Response Example:

```
{
  "target" : "java.util.concurrent.ConcurrentHashMap",
  "name" : "cities",
  "cacheManager" : "cacheManager"
}
```

Query Parameters:

Parameter	Description
`cacheManager`	Name of the cacheManager to qualify the cache (optional if the name is unique)

Response Structure:

Path	Type	Description
`name`	String	Cache name
`cacheManager`	String	Cache manager name
`target`	String	Fully qualified name of the native cache

Evict All Caches

To clear all caches, send a `DELETE` request to `/actuator/caches`:

```
$ curl 'http://localhost:8080/actuator/caches' -i -X DELETE
```

Evict a Cache by Name

To evict a specific cache, send a `DELETE` request to `/actuator/caches/{name}`. Example for cache "countries" under `anotherCacheManager`:

$ curl
'http://localhost:8080/actuator/caches/countries?cacheManager=anotherCacheManager' -i
-X DELETE -H 'Content-Type: application/x-www-form-urlencoded'

Request Structure:

Parameter	Description
`cacheManager`	Name of the cacheManager to qualify the cache (optional if the name is unique)

1.4 Conditions Evaluation Report (/actuator/conditions)

- **Purpose**: Shows auto-configuration report
- **Benefits**:
  - Understand why certain configurations are applied
  - Troubleshoot auto-configuration issues
- **Typical Use**:
  - Debugging configuration problems
  - Understanding conditional bean creation

Retrieving the Report

To retrieve the application's condition evaluation report, make a `GET` request to `/actuator/conditions`. Example using `curl`:

$ curl 'http://localhost:8080/actuator/conditions' -i -X GET

Response Example:

```
{
 "contexts" : {
  "application" : {
   "positiveMatches" : {
    "EndpointAutoConfiguration#propertiesEndpointAccessResolver" : [ {
     "condition" : "OnBeanCondition",
        "message" : "@ConditionalOnMissingBean (types:
org.springframework.boot.actuate.endpoint.EndpointAccessResolver; SearchStrategy: all)
did not find any beans"

      } ],

      "EndpointAutoConfiguration#endpointOperationParameterMapper" : [ {

        "condition" : "OnBeanCondition",

        "message" : "@ConditionalOnMissingBean (types:
org.springframework.boot.actuate.endpoint.invoke.ParameterValueMapper; SearchStrategy:
all) did not find any beans"

      } ]

    },

    "negativeMatches" : {

      "WebFluxEndpointManagementContextConfiguration" : {

        "notMatched" : [ {

          "condition" : "OnWebApplicationCondition",

          "message" : "not a reactive web application"

        } ],

        "matched" : [ {

          "condition" : "OnClassCondition",

          "message" : "@ConditionalOnClass found required classes
'org.springframework.web.reactive.DispatcherHandler',
'org.springframework.http.server.reactive.HttpHandler'"

        } ]

      }

    },

    "unconditionalClasses" : [
"org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration" ]

  }

}
```

```
}
```

Response Structure:

Path	Type	Description
`contexts`	Object	Application contexts keyed by ID
`contexts.*.positiveMatches`	Object	Classes and methods with conditions that were matched
`contexts.*.positiveMatches.*.[].condition`	String	Name of the condition
`contexts.*.positiveMatches.*.[].message`	String	Details of why the condition was matched
`contexts.*.negativeMatches`	Object	Classes and methods with conditions that were not matched
`contexts.*.negativeMatches.*.notMatched`	Array	Conditions that were not matched
`contexts.*.negativeMatches.*.notMatched.[].condition`	String	Name of the condition
`contexts.*.negativeMatches.*.notMatched.[].message`	String	Details of why the condition was not matched

`contexts.*.negativeMatches.*.matched`	Array	Conditions that were matched
`contexts.*.negativeMatches.*.matched.[].condition`	String	Name of the condition
`contexts.*.negativeMatches.*.matched.[].message`	String	Details of why the condition was matched
`contexts.*.unconditionalClasses`	Array	Names of unconditional auto-configuration classes if any
`contexts.*.parentId`	String	ID of the parent application context, if any

1.5 Configuration Properties (/actuator/configprops)

- **Purpose**: Displays all @ConfigurationProperties beans
- **Use Cases**:
  - Verify configuration binding
  - Inspect current application properties

  Example Configuration:
  ```
   @ConfigurationProperties(prefix = "app.database")
  public class DatabaseProperties {
     private String url;
     private String username;
  }
  ```

1.6 Environment Endpoint (/actuator/env)

- **Purpose**: Shows active environment and configuration properties
- **Features**:
  - Display system environment variables
  - Show Spring profiles

    - Inspect configuration sources

  Configuration:

```
management.endpoint.env.keys-to-sanitize=password,secret
```

1.7 Flyway Endpoint (/actuator/flyway)

- **Purpose**: Provides Flyway database migration information
- **Features**:
    - List applied migrations
    - Show migration status

  Dependency Required:

```
 <dependency>
   <groupId>org.flywaydb</groupId>
   <artifactId>flyway-core</artifactId>
</dependency>
```

1.8 Health Endpoint (/actuator/health)

- **Purpose**: Application health monitoring

  Detailed Configuration:

```
 management.endpoint.health.show-details=always
management.health.db.enabled=true
management.health.diskspace.enabled=true
```

Retrieving the Health of the Application

To retrieve the overall health of the application, make a **GET request** to `/actuator/health`:

```
$ curl 'http://localhost:8080/actuator/health' -i -X GET \

    -H 'Accept: application/json'
```

Example Response:

```
{

  "status": "UP",
}
"components": {
  "broker": {
    "status": "UP",
    "components": {
      "us1": {
        "status": "UP",
        "details": {
          "version": "1.0.2"
        }
      },
      "us2": {
        "status": "UP",
        "details": {
          "version": "1.0.4"
        }
      }
    }
  },
  "db": {
    "status": "UP",
    "details": {
      "database": "H2",
      "validationQuery": "isValid()"
    }
  },
  "diskSpace": {
   "status": "UP",
   "details": {
    "total": 77851254784,
    "free": 48648368128,
    "threshold": 10485760,
    "path": "/home/runner/work/spring-boot/spring-boot/spring-boot-project/spring-boot-actuator-autoconfigure/.",
    "exists": true
   }
  }
 }
}
```

Response Structure

- **status**: The overall status of the application (e.g., `UP`, `DOWN`).
- **components**: Contains various components (e.g., `broker`, `db`, `diskSpace`) with their health status.
- **components.*.status**: The status of individual components.
- **components.*.details**: Further details regarding the health of specific components, if available.

Retrieving the Health of a Component

To retrieve the health of a specific component (e.g., `db`), make a **GET request** to `/actuator/health/{component}`:

```
$ curl 'http://localhost:8080/actuator/health/db' -i -X GET \
  -H 'Accept: application/json'
```

Example Response:

```
{
  "status": "UP",
  "details": {
    "database": "H2",
    "validationQuery": "isValid()"
  }
}
```

Retrieving the Health of a Nested Component

If a component contains nested components (e.g., `broker/us1`), use a **GET request** to `/actuator/health/{component}/{subcomponent}`:

```
$ curl 'http://localhost:8080/actuator/health/broker/us1' -i -X GET \
    -H 'Accept: application/json'
```

Example Response:

```
{
  "status": "UP",
  "details": {
    "version": "1.0.2"
  }
}
```

Response Structure for Nested Components

- **status**: Status of the specific part of the application (e.g., `UP`, `DOWN`).
- **details**: Additional details of the health of that specific part.

The health endpoint supports retrieving the health status of any component and its nested subcomponents through `/actuator/health/{component}/{subcomponent}`, allowing flexibility depending on the application's setup.

1.9 Heap Dump Endpoint (/actuator/heapdump)

- **Purpose**: Generate JVM heap dump
- **Use Cases**:
  - Memory leak investigation
  - Performance troubleshooting
- **Caution**: Use carefully in production

1.10 HTTP Exchanges (/actuator/httpexchanges)

- **Purpose**: Record HTTP request-response exchanges
- **Configuration**:

```
 @Bean
public HttpExchangeRepository httpTraceRepository() {
    return new InMemoryHttpExchangeRepository();
}
```

1.11 Info Endpoint (/actuator/info)

- **Purpose**: Display custom application information
- **Configuration**:

```
 info.app.name=MyApplication
info.app.description=Sample Spring Boot App
info.app.version=1.0.0
```

1.12 Spring Integration Graph (/actuator/integrationgraph)

- **Purpose**: Visualize Spring Integration message flows
- **Dependency**:

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

1.13 Liquibase Endpoint (/actuator/liquibase)

- **Purpose**: Database schema migration tracking
- **Dependency**:

```
 <dependency>
 <groupId>org.liquibase</groupId>
    <artifactId>liquibase-core</artifactId>
</dependency>
```

1.14 Log File Endpoint (/actuator/logfile)

- **Purpose**: Access application log files
- **Configuration**:
  logging.file.name=logs/application.log
  management.endpoint.logfile.enabled=true

1.15 Loggers Endpoint (/actuator/loggers)

- **Purpose**: Runtime log level management
- **Features**:
  - View current log levels
  - Modify log levels dynamically

1.16 Mappings Endpoint (/actuator/mappings)

- **Purpose**: Display all HTTP endpoint mappings
- **Use Cases**:
  - Inspect REST endpoint configurations
  - Troubleshoot routing issues

1.17 Metrics Endpoint (/actuator/metrics)

- **Purpose**: Application and system metrics
- **Features**:
  - JVM metrics
  - System resource metrics
  - Custom application metrics

1.18 Prometheus Endpoint (/actuator/prometheus)

- **Purpose**: Expose metrics in Prometheus format

Dependency:

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

-

1.19 Quartz Endpoint (/actuator/quartz)

- **Purpose**: Scheduled job monitoring

Dependency:

```
 <dependency>
```

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

-

1.20 SBOM Endpoint (/actuator/sbom)

- **Purpose**: Generate Software Bill of Materials
- **Use Cases**:
  - Dependency tracking
  - Security compliance

1.21 Scheduled Tasks Endpoint (/actuator/scheduledtasks)

- **Purpose**: List and monitor scheduled tasks
- **Features**:
  - Show @Scheduled method details
  - Display task schedules

1.22 Sessions Endpoint (/actuator/sessions)

- **Purpose**: Manage and inspect HTTP sessions
- **Requires**:
  - Spring Session dependency
  - Web application context

1.23 Shutdown Endpoint (/actuator/shutdown)

- **Purpose**: Gracefully shut down the application

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

-
- **Caution**: Use with extreme care in production

1.24 Application Startup Endpoint (/actuator/startup)

- **Purpose**: Track application startup process
- **Features**:
  - Startup step timeline
  - Performance insights

1.25 Thread Dump Endpoint (/actuator/threaddump)

- **Purpose**: Generate JVM thread dump

- **Use Cases**:
    - Diagnose thread deadlocks
    - Performance analysis