

A large, solid red rectangle occupies the upper half of the slide. The word "Actuator" is written in white text on the left side of this rectangle.

Actuator

Monitor to your work

What is Actuator

- Actuator comes with most endpoints disabled
- the only two available by default are **/actuator**, **/health** and **/actuator/health/{*path}info**.
- Actuator now shares the security config with the regular App security rules, so the security model is dramatically simplified

Below code is equal to **management.endpoints.web.exposure.include=***

- @Bean

```
public SecurityWebFilterChain securityWebFilterChain(  
    ServerHttpSecurity http) {  
    return http.authorizeExchange()  
        .pathMatchers("/actuator/**").permitAll()  
        .anyExchange().authenticated()  
        .and().build();  
}
```

Predefined endpoints in Actuator

- The path using the new property **management.endpoints.web.base-path**.
- some new endpoints have been added, some removed and some have been restructured
- **/auditevents** lists security audit-related events such as user login/logout. Also, we can filter by principal or type among other fields.
- **/beans** returns all available beans in our BeanFactory. Unlike /auditevents, it doesn't support filtering.
- **/conditions**, formerly known as /autoconfig, builds a report of conditions around autoconfiguration.
- **/configprops** allows us to fetch all @ConfigurationProperties beans.

Enabling Spring Boot Actuator

- To enable Spring Boot actuator endpoints to your Spring Boot application, we need to add the Spring Boot Starter actuator dependency in our build configuration file. Maven users can add the below dependency in your pom.xml file.

```
<dependency>
```

```
  <groupId>org.springframework.boot</groupId>
```

```
  <artifactId>spring-boot-starter-actuator</artifactId>
```

```
</dependency>
```

- Gradle users can add the below dependency in your build.gradle file.

```
compile group: 'org.springframework.boot', name: 'spring-boot-starter-actuator'
```

Predefined endpoints in Actuator

- **/env** returns the current environment properties. Additionally, we can retrieve single properties.
- **/flyway** provides details about our Flyway database migrations.
- **/health** summarizes the health status of our application.
- **/heapdump** builds and returns a heap dump from the JVM used by our application.
- **/info** returns general information. It might be custom data, build information or details about the latest commit.
- **/liquibase** behaves like **/flyway** but for Liquibase.
- **/logfile** returns ordinary application logs.

Predefined endpoints in Actuator

- **ENDPOINTS USAGE**
- `/metrics` To view the application metrics such as memory used, memory free, threads, classes, system uptime etc.
- `/env` To view the list of Environment variables used in the application.
- `/beans` To view the Spring beans and its types, scopes and dependency.
- `/health` To view the application health
- `/info` To view the information about the Spring Boot application.
- `/trace` To view the list of Traces of your Rest endpoints.

Predefined endpoints in Actuator

- **/loggers** enables us to query and modify the logging level of our application.
- **/metrics** details metrics of our application. This might include generic metrics as well as custom ones.
- **/prometheus** returns metrics like the previous one, but formatted to work with a Prometheus server.
- **/scheduledtasks** provides details about every scheduled task within our application.
- **/sessions** lists HTTP sessions given we are using Spring Session.
- **/shutdown** performs a graceful shutdown of the application.
- **/threaddump** dumps the thread information of the underlying JVM.

Creating Microservice

The screenshot shows a web browser window with multiple tabs. The active tab is titled 'localhost:9090/actuator'. The address bar shows the URL 'localhost:9090/actuator'. The browser interface includes navigation buttons (back, forward, refresh, home), a search bar, and a star icon for bookmarks. Below the address bar, there are tabs for 'JSON', 'Raw Data', and 'Headers'. The 'JSON' tab is selected, and the response is displayed in a collapsible tree view. The response is a JSON object with the following structure:

```
{
  "_links": {
    "self": {
      "href": "http://localhost:9090/actuator",
      "templated": false
    }
  },
  "beans": {
    "href": "http://localhost:9090/actuator/beans",
    "templated": false
  },
  "caches-cache": {
    "href": "http://localhost:9090/actuator/caches/{cache}",
    "templated": true
  },
  "caches": {
    "href": "http://localhost:9090/actuator/caches",
    "templated": false
  },
  "health": {
    "href": "http://localhost:9090/actuator/health",
    "templated": false
  },
  "health-path": {
    "href": "http://localhost:9090/actuator/health/{*path}",
    "templated": true
  },
  "info": {
    "href": "http://localhost:9090/actuator/info",
    "templated": false
  }
}
```


How is Actuator exposing endpoints

- Spring Boot adds a discovery endpoint that returns links to all available actuator endpoints. This will facilitate discovering actuator endpoints and their corresponding URLs.
- By default, this discovery endpoint is accessible through the **/actuator** endpoint.
- Therefore, if we send a GET request to this URL, it'll return the actuator links for the various endpoints:

if we configure a custom management base path, then we should use that base path as the discovery URL.

How to change Actuator URL

- if we set the **management.endpoints.web.base-path** to **/mgmt**, then we should send a request to the **/mgmt** endpoint to see the list of links.
- When the management base path is set to **/**, the discovery endpoint is disabled to prevent the possibility of a clash with other mappings.

- We can add custom indicators easily. Opposite to other APIs, the abstractions for creating custom health endpoints remain unchanged. However, a new interface, `ReactiveHealthIndicator`, has been added to implement reactive health checks.
- A simple custom reactive health check:

Health check in Actuator

@Component

```
public class DownstreamServiceHealthIndicator implements ReactiveHealthIndicator {
```

@Override

```
public Mono<Health> health() {
```

```
    return checkDownstreamServiceHealth().onErrorResume(
```

```
        ex -> Mono.just(new Health.Builder().down(ex).build())
```

```
    );
```

```
}
```

```
private Mono<Health> checkDownstreamServiceHealth() {
```

```
    // we could use WebClient to check health reactively
```

```
    return Mono.just(new Health.Builder().up().build());
```

```
}
```

```
implementation 'org.springframework.boot:spring-boot-starter-webflux'

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
  <version>2.6.4</version>
</dependency>
```

Health check in Actuator

- create a health group named custom by adding this to our **application.properties**:
- `management.endpoint.health.group.custom.include`=*diskSpace,ping*
- `management.endpoint.health.group.custom.show-components`=*always*
- `management.endpoint.health.group.custom.show-details`=*always*
- `management.security.enabled`=*false*
- The custom group contains the diskSpace and ping health indicators. if we call the **/actuator/health** endpoint, it would tell us about the new health group in the JSON response:

```
{"status":"UP","groups":["custom"]}
```

Health check in Actuator

- we send the same request to **/actuator/health/custom**, we'll see more details:

```
{  
  "status": "UP",  
  "components": {  
    "diskSpace": {  
      "status": "UP",  
      "details": {  
        "total": 499963170816,  
        "free": 91300069376,  
        "threshold": 10485760  
      }  
    },  
    "ping": {  
      "status": "UP" }  
  }  
}
```

Health check in Actuator

- It's also possible to show these details only for authorized users:

```
management.endpoint.health.group.custom.show-components=when_authorized
```

```
management.endpoint.health.group.custom.show-details=when_authorized
```

```
management.endpoint.health.group.custom.status.http-mapping.up=207
```

- we're telling Spring Boot to return a 207 HTTP status code if the custom group status is UP.

Metrics in Spring Boot 2 Actuator

- metrics were replaced with Micrometer support, so we can expect breaking changes. If our application was using metric services such as GaugeService or CounterService, they will no longer be available.
- Instead, we're expected to interact with Micrometer directly. In Spring Boot 2.0, we'll get a bean of type MeterRegistry autoconfigured for us.
- Micrometer is now part of Actuator's dependencies, so we should be good to go as long as the Actuator dependency is in the classpath.

Metrics in Spring Boot 2 Actuator

- completely new response from the */metrics* endpoint:

```
{  
  "names": [  
    "jvm.gc.pause",  
    "jvm.buffer.memory.used",  
    "jvm.memory.used",  
    "jvm.buffer.count",  
    // ...  
  ]  
}
```


Metrics in Spring Boot 2 Actuator

- There are no actual metrics as we got in 1.x.
- To get the actual value of a specific metric, we can now navigate to the desired metric, e.g., **/actuator/metrics/jvm.gc.pause**, and get a detailed response:

Customizing the **/info** Endpoint

- The `/info` endpoint remains unchanged. As before, we can add git details using the respective Maven or Gradle dependency:

```
<dependency>
```

```
<groupId>pl.project13.maven</groupId>
```

```
<artifactId>git-commit-id-plugin</artifactId>
```

```
</dependency>
```

- we could also include build information including name, group, and version using the Maven or Gradle plugin:

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<executions>

<execution>

<goals>

<goal>build-info</goal>

</goals>

</execution>

</executions>

</plugin>

Customizing the /info Endpoint

@Component

@Endpoint(id = "features")

public class FeaturesEndpoint {

private Map<String, Feature> features = new ConcurrentHashMap<>();

@ReadOperation

public Map<String, Feature> features() {

return features;

}

@ReadOperation

public Feature feature(@Selector String name) {

return features.get(name);

}

Creating a Custom Endpoint

@WriteOperation

```
public void configureFeature(@Selector String name, Feature feature) {  
    features.put(name, feature);  
}
```

@DeleteOperation

```
public void deleteFeature(@Selector String name) {  
    features.remove(name);  
}
```

```
public static class Feature {  
    private Boolean enabled;  
    // [...] getters and setters  
}
```

Creating a Custom Endpoint

To get the endpoint, we need a bean. In our example, we're using `@Component` for this. Also, we need to decorate this bean with `@Endpoint`.

The path of our endpoint is determined by the `id` parameter of `@Endpoint`. In our case, it'll route requests to `/actuator/features`.

Once ready, we can start defining operations using:

- `@ReadOperation`: It'll map to HTTP GET.

- `@WriteOperation`: It'll map to HTTP POST.

- `@DeleteOperation`: It'll map to HTTP DELETE.

When we run the application with the previous endpoint in our application, Spring Boot will register it

Extending Existing Endpoints

- we want to make sure the production instance of our application is never a SNAPSHOT version.
- We decide to do this by changing the HTTP status code of the Actuator endpoint that returns this information, i.e., **/info**. If our app happened to be a SNAPSHOT, we would get a different HTTP status code.
- We can easily extend the behavior of a predefined endpoint using the **@EndpointExtension** annotations, or its more concrete specializations **@EndpointWebExtension** or **@EndpointJmxExtension**:

@Component

@EndpointWebExtension(endpoint = InfoEndpoint.class)

public class InfoWebEndpointExtension {

private InfoEndpoint delegate;

// standard constructor

@ReadOperation

public WebEndpointResponse<Map> info() {

Map<String, Object> info = this.delegate.info();

Integer status = getStatus(info);

return new WebEndpointResponse<>(info, status);

}

private Integer getStatus(Map<String, Object> info) {

// return 5xx if this is a snapshot

return 200;

}}

Extending Existing Endpoints

Enable All Endpoints

Only the **/health** and **/info** endpoints are exposed by default.

We need to add the following configuration to expose all endpoints:

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

To explicitly enable a specific endpoint (e.g., **/shutdown**), we use:

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

To expose all enabled endpoints except one (e.g., **/loggers**), we use:

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

```
management.endpoints.web.exposure.exclude=loggers
```

Endpoints

- **Endpoints** are sensitive-meaning they're not fully public, or most information will be omitted-while a handful are not, e.g., **/info**
- **/health** shows application health information (a simple status when accessed over an unauthenticated connection or full message details when authenticated); it's not sensitive by default.
- **/info** displays arbitrary application info; it's not sensitive by default.
- **/metrics** shows metrics information for the current application; it's sensitive by default.
- **/trace** displays trace information (by default the last few HTTP requests).

- We can customize each endpoint with properties using the format endpoints.[endpoint name].[property to customize]. Three properties are available:

Configuring Existing Endpoints

- **id:** by which this endpoint will be accessed over HTTP
- **enabled:** if true, then it can be accessed; otherwise not
- **sensitive:** if true, then need the authorization to show crucial information over HTTP

Example, adding the following properties will customize the **/beans** endpoint:

```
endpoints.beans.id=springbeans
```

```
endpoints.beans.sensitive=false
```

```
endpoints.beans.enabled=true
```

/health Endpoint

- Health information is collected from all the beans implementing the ***HealthIndicator*** interface configured in our application context.
- Some information returned by ***HealthIndicator*** is sensitive in nature, but we can configure ***endpoints.health.sensitive=false*** to expose more detailed information like disk space, messaging broker connectivity, custom checks, and more.
- We should also disable security by setting ***management.security.enabled=false*** for unauthorized access.
- We could also implement our own custom health indicator, which can collect any type of custom health data specific to the application and automatically expose it through the ***/health*** endpoint:

```
@Component("myHealthCheck")
```

```
public class HealthCheck implements HealthIndicator {
```

```
    @Override
```

```
    public Health health() {
```

```
        int errorCode = check(); // perform some specific health check
```

```
        if (errorCode != 0) {
```

```
            return Health.down()
```

```
                .withDetail("Error Code", errorCode).build();
```

```
        }
```

```
        return Health.up().build();
```

```
    }
```

```
    public int check() {
```

```
        // Our logic to check health
```

```
        return 0;
```

```
    }}
```

/health Endpoint

/info Endpoint

We can also customize the data shown by the /info endpoint:

`info.app.name=Spring Sample Application`

`info.app.description=This is my first spring boot application`

`info.app.version=1.0.0`

And the sample output:

```
{  
  "app" : {  
    "version" : "1.0.0",  
    "description" : "This is my first spring boot application",  
    "name" : "Spring Sample Application"  
  }  
}
```

/metrics Endpoint

- The metrics endpoint publishes information about OS and JVM as well as application-level metrics.
- Once enabled, we get information such as memory, heap, processors, threads, classes loaded, classes unloaded, and thread pools along with some HTTP metrics as well.
- In order to gather custom metrics, we have support for gauges (single-value snapshots of data) and counters, i.e., **incrementing/decrementing** metrics.
- Let's implement our own custom metrics into the **/metrics** endpoint.
- We'll customize the login flow to record a successful and failed login attempt:

@Component

```
public class CustomEndpoint implements  
Endpoint<List<String>> {
```

@Override

```
public String getId() {  
    return "customEndpoint";  
}
```

@Override

```
public boolean isEnabled() {  
    return true;  
}
```

@Override

```
public boolean isSensitive() {  
    return true;  
}
```

Creating a New Endpoint

@Override

```
public List<String> invoke() {  
    // Custom logic to build the output  
    List<String> messages = new  
    ArrayList<String>();  
    messages.add("This is message 1");  
    messages.add("This is message 2");  
    return messages;  
}
```

In order to access this new endpoint, its id is used to map it. In other words we could exercise it hitting /customEndpoint.

Further Customization

#port used to expose actuator

`management.port=8081`

#CIDR allowed to hit actuator

`management.address=127.0.0.1`

#Whether security should be enabled or disabled altogether

`management.security.enabled=false`

Besides, all the built-in endpoints except `/info` are sensitive by default.

If the application is using Spring Security, we can secure these endpoints by defining the default security properties (**username, password, and role**) in the **application.properties** file:

`security.user.name=admin`

`security.user.password=secret`

`management.security.role=SUPERUSER`