Spring Actuator

Spring boot Actuator is one of the popular feature of Spring Boot for creating production ready applications. Actuator exposes set of endpoints to monitor your application health, view metrics, secure them with spring security and many other features that are necessary for you to declare your application production ready.

Features of Spring Boot Actuator

1. Spring Boot Actuator module allows you to monitor your applications in 2 ways; either by leveraging HTTP endpoints or JMX (Java Management Extension). Most of the applications leverage HTTP endpoints.

2. **Endpoints:** Actuator module is bundled with a rich set of HTTP endpoints to monitor and interact with the application. You can even create custom endpoints to fulfil your needs, and also leverage spring-security to secure actuator endpoints. Few examples are

   - info
   - health
   - metrics
   - beans
   - caches
   - loggers
   - mappings

3. **Metrics:** Spring Boot Actuator provides support for a instrumentation library, called **Micrometer**, which is a **vendor neutral application metrics facade** that captures and exposes application metrics to different monitoring solutions such as Prometheus, Dynatrace, New Relic, DataDog and many more… It provides interfaces for **timers, gauges, counters, distribution summaries, and long task timers** with a dimensional data model.

4. Actuator module is highly (yet simply) configurable, and you can utilize your application.[properties|yaml] file to meet most of your customized needs.

Health endpoint is one of the important actuator endpoint, It helps in:

- **Monitoring Application Health**: The Health endpoint provides a simple and standardized way to monitor the health of your application. It allows you to quickly check whether your application is running correctly or encountering any issues. Monitoring the health status of your application helps you identify and respond to potential problems early, reducing downtime and ensuring a better user experience.
- **Proactive Issue Detection**: By regularly polling the Health endpoint, monitoring tools and systems can proactively detect any degradation or failure in your application's health. This enables your operations team to take immediate action before issues escalate and impact users.
- **Integration with Orchestration Tools**: In modern production environments, applications are often managed and orchestrated by tools like Kubernetes, Docker Swarm, or cloud platforms.
- **Operational Insights**: The Health endpoint provides essential information about your application's internals. You can customize the endpoint to include details about database connections, external service status, resource availability, and more. These details are valuable for troubleshooting and gaining insights into your application's behavior during runtime.

Customizing Health endpoint by implementing HealthIndicator

Implementing the HealthIndicator interface allows you to create custom health indicators to monitor specific aspects of your application.

The HealthIndicator (package: org.springframework.boot.actuate.health) interface is part of the Spring Boot Actuator module, and it provides a single method, health(), that returns a Health object representing the health status of your custom indicator. It provides several static methods to create instances of Health, as well as to add additional details and contextual information about the health check.

Here are some commonly used static methods of the Health class:

up(): Creates a Health instance with an UP status, indicating that the application is in a healthy state.

down(): Creates a Health instance with a DOWN status.

unknown(): Creates a Health instance with an UNKNOWN status.

status(Status status): Creates a Health instance with a custom status. The Status enum represents the health status, and it can be UP, DOWN, or UNKNOWN.

withDetail(String key, Object value): Adds additional details to the Health instance. These details provide more information about the health status and can be useful for troubleshooting and monitoring.

withDetails(Map details): Adds multiple details to the Health instance at once, using a Map of key-value pairs.

Let's now write an implementation of the HealthIndicator interface i.e. ExternalServiceHealthIndicator.class. It's a typical example, where I am returning a Health object based on HTTP statusCode received from external service call.

```
@Component

public class ExternalServiceHealthIndicator implements HealthIndicator {

  private final Random randomizer = new Random();

  private final List<Integer> statusCodes = List.of(200, 204, 401, 404, 503);

  @Override

  public Health health() {

    int randomStatusCode = statusCodes.get(randomizer.nextInt(statusCodes.size()));

    Health.Builder healthBuilder = new Health.Builder();

    return (switch(randomStatusCode) {

      case 200, 204 -> healthBuilder.up()

          .withDetail("External_Service", "Service is Up and Running ✅")

          .withDetail("url", "https://example.com");

      case 503 -> healthBuilder.down()

          .withDetail("External_Service", "Service is Down 🔻")

          .withDetail("alternative_url", "https://alt-example.com");

      default -> healthBuilder.unknown().withException(new RuntimeException("Received
status: " + randomStatusCode));

    }).build();

  }

}
```

@ConditionalOnEnabledHealthIndicator annotation

@ConditionalOnEnabledHealthIndicator is a meta-annotation in org.springframework.boot.actuate.autoconfigure.health package to conditionally enable or disable the health check provided by that specific indicator and requires indicatorName as a mandatory String argument which can be enabled with the externalised application configuration. Let's see an example in action.

```
@Component

@ConditionalOnEnabledHealthIndicator("external_service_health")

public class ExternalServiceHealthIndicator implements HealthIndicator {

   @Override

   public Health health() {

      ...

   }

}
```

The Custom indicator can be enabled/disabled by setting management.health.<custom-indicator>.enabled property true/false in the application's configuration. If the property is not present or set to false, the indicator will not be enabled.

```
management.health.external_service_health.enabled: true
```

Custom Endpoints

No doubt, By default, Spring Actuator exposes a wider set of pre-defined endpoints that can be accessed to gather information about your application's health, metrics, configuration, and more. But there are scenarios where you might want to create custom Actuator endpoints to extend the monitoring and management capabilities of your application.

Here are some benefits and scenarios where custom endpoints can be useful:

- **Administrative Tasks:** You can use custom endpoints to perform administrative tasks on your application. For example, triggering a specific maintenance routine, clearing cache, or reloading configurations can be exposed through custom endpoints.

- **Custom Health Checks:** While Actuator provides a default health check endpoint, And in the last Post we deep dived into customizing your health endpoint. But your application might have specific health checks that are beyond the scope of the built-in checks. Custom health check endpoints allow you to implement and expose these checks.
- **Security and Access Control:** By creating custom endpoints, you have fine-grained control over who can access specific monitoring and management information. This allows you to restrict access to sensitive endpoints, ensuring that only authorized users or systems can interact with them.
- **Third-party Integrations:** If your application relies on external services or APIs, you can create custom endpoints to monitor the health and status of these integrations. This can be especially useful to quickly identify issues with external dependencies.
- **Complex Aggregations:** The default Actuator endpoints provide basic metrics, but in some cases, you might need to perform complex aggregations and calculations to derive meaningful insights.

Building Custom Endpoint

Now, Let's try to build our own custom HTTP actuator endpoint.

@Endpoint

@Endpoint annotation is a core component of Spring Actuators. Once you annotate your class with @Endpoint annotation and provide id field (mandatorily) as custom value, This class will now represent your custom endpoint.

```
@Component

@Endpoint(id="custom")

public class CustomActuatorEndpoint {

   …

}
```

Do not forget to make your endpoint class spring-managed component by defining @Component annoation.

This will make following actuator endpoint available to be served: http://your-example-web-app:2121/actuator/custom. Not let's define **Operations** to add the definition for this custom endpoint.

Endpoint Operations

Spring Boot Actuator endpoints support mainly 3 basic operations to interact with custom endpoints. You can define methods and annotate them with these annotations.

- **@ReadOperation:** @ReadOperation is used to handle HTTP GET requests. It is used to retrieve information from the Actuator endpoint. Typically, this operation is used to expose various metrics, health information, configurations, and other read-only data.
- **@WriteOperation:** This operation type is used to handle HTTP POST requests to modify the state of the application or perform actions that have side effects.
- **@DeleteOperation:** As it states, @DeleteOperation is used to handle HTTP DELETE requests to remove or delete resources from the application. It can be helpful when you want to provide an API for removing resources or performing specific cleanup tasks.

It is always recommended to secure your actuator endpoint with spring security while exposing sensitive information or performing write/delete operations that alters the state/config of your application.

It is quite similar to defining an endpoint using Get/Post/Delete Mappings.

Example Custom Endp

This is how your ReleaseNote data object looks like

```
public class ReleaseNote {

  private String version;

  private LocalDateTime date;

  private String changeLogData;

  // Constructors

  // Getters/Setters

}
```

Now, Let's write ReleaseNoteEndpoint class which uses ReleaseNotesDataRepository to manage persistence of your ReleaseNote object.

```
@Component
```

```
@Endpoint(id="releaseNotes")
public class ReleaseNotesEndpoint {

  @Autowired
  private ReleaseNotesDataRepository releaseNotesDataRepository;

  @ReadOperation
  public List<ReleaseNote> getReleaseNotes() {
    return releaseNotesDataRepository.getReleaseNoteList();
  }

  @WriteOperation
  public ReleaseNote addReleaseNote(String version, String changeLogData) {
    ReleaseNote releaseNote = new ReleaseNote(version, LocalDateTime.now(),
changeLogData);
    return releaseNotesDataRepository.addReleaseNote(releaseNote);
  }

  @DeleteOperation
  public void deleteReleaseNote(@Selector String version) {
    releaseNotesDataRepository.deleteReleaseNote(version);
  }
}
```

Spring Actuator comes with **@Selector** annotation that is used as an argument of an @Endpoint method to indicate the path parameter for your endpoint