(/)

Logging HTTP Requests with Spring Boot Actuator HTTP Tracing

Last modified: August 26, 2019

by Marcos Lopez Gonzalez (https://www.baeldung.com/author/marcos-lopezgonzalez/)

DevOps (https://www.baeldung.com/category/devops/)
Logging (https://www.baeldung.com/category/logging/)
Spring Boot (https://www.baeldung.com/category/spring/spring-boot/)

1. Introduction

When we work with microservices or web services in general, it's quite useful to know how our users interact with our services. This can be achieved by tracing all the requests that hit our services and collect this information to analyze it later.

There some systems available out there that can help us with this and can be easily integrated with Spring like Zipkin (https://www.baeldung.com/tracing-services-with-zipkin). However, **Spring Boot Actuator has this functionality built-in** and can be used through its *httpTrace* endpoint which traces all HTTP requests. In this tutorial, we'll show how to use it and how to customize it to fit better our requirements.

2. *HttpTrace* Endpoint Setup

For the sake of this tutorial, we'll use a Maven Spring Boot project (https://www.baeldung.com/spring-boot).

The first thing we need to do is to add the Spring Boot Actuator dependency (https://search.maven.org/search?q=a:spring-boot-starter-actuator%20AND%20g:org.springframework.boot) to our project:

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

After that, we'll have to enable the *httpTrace* endpoint in our application.

To do so, we just need to **modify our *application.properties* file to include the *httpTrace* endpoint**:

all of them by using the wildcard *.

Now, our *httpTrace* endpoint should appear in the actuator endpoints list of our application:

```
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator (http://localhost:8080/actuator)",
      "templated": false
    },
    "httptrace": {
      "href": "http://localhost:8080/actuator/httptrace (http://localhost:8080/actuator/httptrace)",
      "templated": false
    }
  }
}
```

Notice that we can list all the enabled actuator endpoints by going to the */actuator* endpoint of our web service.

3. Analyzing the Traces

Let's analyze now the traces that the *httpTrace* actuator endpoint returns.

Let's make some requests to our service, call the */actuator/httptrace* endpoint and take one of the traces returned:

```
{
  "traces": [
    {
      "timestamp": "2019-08-05T19:28:36.353Z",
      "principal": null,
      "session": null,
      "request": {
        "method": "GET",
        "uri": "http://localhost:8080/echo?msg=test (http://localhost:8080/echo?msg=test)",
        "upgrade-insecure-requests": [
          "1"
        ],
        "host": [
          "localhost:8080"
        ],
        "connection": [
          "keep-alive"
        ],
        "accept-encoding": [
          "gzip, deflate, br"
        ],
        "accept": [
          "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8"
        ],
        "user-agent": [
          "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko)
        ]
      },
      "remoteAddress": null
    },
    "response": {
      "status": 200,
      "headers": {
        "Content-Length": [
          "12"
        ],
        "Date": [
          "Mon, 05 Aug 2019 19:28:36 GMT"
        ],
        "Content-Type": [
          "text/html;charset=UTF-8"
        ]
      }
    },
    "timeTaken": 82
  }
]
}
```


As we can see, the response is divided into several nodes:

- *timestamp*: the time when the request was received
- *principal*: the authenticated user who did the request, if applicable
- *session*: any session associated with the request
- *request*: information about the request such as the method, full URI or headers
- *response*: information about the response such as the status or the headers
- *timeTaken*: the time taken to handle the request

We can adapt this response to our needs if we feel it's too verbose. We can tell Spring what fields we want to be returned by **specifying them in the *management.trace.http.include* property** of our *application.properties*:

```
management.trace.http.include=RESPONSE_HEADERS
```

In this case, we specified that we only want the response headers. Hence, we can see that fields that were included before like the request headers or the time taken are not present in the response now:

```
{
  "traces": [
    {
      "timestamp": "2019-08-05T20:23:01.397Z",
      "principal": null,
      "session": null,
      "request": {
        "method": "GET",
        "uri": "http://localhost:8080/echo?msg=test (http://localhost:8080/echo?msg=test)",
        "status": 200,
        "headers": {
          "Content-Length": [
            "12"
          ],
          "Date": [
            "Mon, 05 Aug 2019 20:23:01 GMT"
          ],
          "Content-Type": [
            "text/html;charset=UTF-8"
          ]
        }
      },
      "timeTaken": null
    }
  ]
}
```


All the possible values that can be included can be found in the source code (https://github.com/spring-projects/spring-boot/blob/8bc780762a44c7d57e6ddf15abb7071f499857ce/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/trace/http/Include.java), as well as the default ones.

4. Customizing the *HttpTraceRepository*

By default, the *httpTrace* endpoint only returns the last 100 requests and it stores them in memory. The good news is that we can also customize this by creating our own *HttpTraceRepository*.

Let's now create our repository. The *HttpTraceRepository* interface is very simple and we only need to implement two methods: *findAll()* to retrieve all the traces; and *add()* to add a trace to the repository.

For simplicity, our repository will also store the traces in memory and we'll store only the last GET request that hits our service:

```
@Repository
public class CustomTraceRepository implements HttpTraceRepository {

    AtomicReference<HttpTrace> lastTrace = new AtomicReference<>();

    @Override
    public List<HttpTrace> findAll() {
        return Collections.singletonList(lastTrace.get());
    }

    @Override
    public void add(HttpTrace trace) {
        if ("GET".equals(trace.getRequest().getMethod())) {
            lastTrace.set(trace);
        }
    }

}
```

Even though this simple example may not look very useful, we can see how powerful this can get and how we could store our logs anywhere.

5. Filtering the Paths to Trace

The last thing we're going to cover is how to filter the paths that we want to trace, so we can ignore some requests that we're not interested in.

If we play with the *httpTrace* endpoint a little after making some requests to our service, we can see that we also get traces for the actuator requests:

```
{
  "traces": [
    {
      "timestamp": "2019-07-28T13:56:36.998Z",
      "principal": null,
      "session": null,
      "request": {
        "method": "GET",
        "uri": "http://localhost:8080/actuator/ (http://localhost:8080/actuator/)",
         // ...
}
```

We may not find these traces useful for us and we'd prefer to exclude them. In that case, we just need to **create our own *HttpTraceFilter*** and specify what paths we want to **ignore in the *shouldNotFilter* method:**

```
@Component
public class TraceRequestFilter extends HttpTraceFilter {

  public TraceRequestFilter(HttpTraceRepository repository, HttpExchangeTracer tracer) {
      super(repository, tracer);
  }

  @Override
  protected boolean shouldNotFilter(HttpServletRequest request) throws ServletException {
      return request.getServletPath().contains("actuator");
  }
}
```

Notice that the *HttpTraceFilter* is just a regular Spring filter but with some tracing-specific functionality.

6. Conclusion

In this tutorial, we've introduced the *httpTrace* Spring Boot Actuator endpoint and shown its main features. We've also dug a bit deeper and explained how to change some default behaviors to fit better our specific needs.

As always, the full source code for the examples is available over on GitHub (https://github.com/eugenp/tutorials/tree/master/spring-boot-management).

Leave a Reply

Start the discussion...

Subscribe

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT-SIDE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)

KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

X

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE COURSES (HTTPS://COURSES.BAELDUNG.COM)
CONSULTING WORK (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS)
ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)