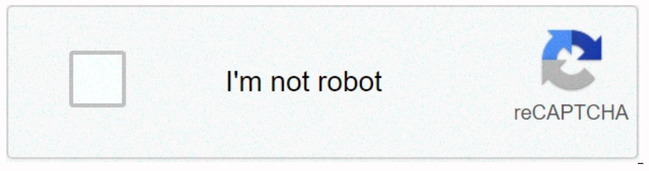

Continue

Spring resttemplate interceptor order

In this tutorial, we'll learn how to implement a spring RestTemplate interceptor. An example will be passed in which an interceptor will be created that adds a custom header to the response. In addition to changing the header, some of the other use cases where a RestTemplate interceptor is useful are: request and request registration Retry requests with a configurable request denial back-off strategy based on certain request parameters By altering the URL address of request 3. Creating the interceptor In most programming paradigms, interceptors are an essential part that allows programmers to control execution by intercepting it. The spring framework also supports a variety of interceptors for different purposes. Spring RestTemplate allows you to add interceptors that implement the ClientHttpRequestInterceptor interface. The intercept(HttpRequest, byte[], ClientHttpRequestExecution) method of this interface will intercept the specified request and return the response by giving us access to the requested, body, and execute objects. The ClientHttpRequestExecution argument will be used to perform the actual execution and pass the request to the next job chain. As a first step, we create an interceptor class that implements the ClientHttpRequestInterceptor interface: the public class RestTemplateHeaderModifierInterceptor implements ClientHttpRequestInterceptor , @Override public ClientHttpResponse intercept(HttpRequest request, byte[] body, ClientHttpRequestExecution execution) throws IOException , ClientHttpResponse response , execute.execute(request, body); response.getHeaders().add(Foo Our interceptor will be invoked for each incoming request and will add a foo custom header to each response, once execution is completed and returned. Because the intercept() method included the request and the body as arguments, you can also make any changes to the request or even deny the request execution under certain conditions. 4. Setting up the RestTemplate Now that we have created our interceptor, we create the restTemplate bean and add our interceptor to it: @Configuration public class RestClientConfig - @Bean public RestTemplate restTemplate() - RestTemplate restTemplate - new RestTemplate(); List Interceptors<ClientHttpRequestInterceptor> : restTemplate.getInterceptors(); if (CollectionUtils.isEmpty(interceptors)) - interceptors , new ArrayList (); , <> interceptors.add(new RestTemplateHeaderModifierInterceptor()); restTemplate.setInterceptors(interceptors); return restTemplate; In some cases, interceptors may have already been added to the RestTemplate object. So to make sure everything works as expected, our code will 30 the list of interceptors if it's empty. As shown in the code, we are using the default constructor to create the Object RestTemplate, but there are some scenarios where you need to read the request/response flow twice. For example, if we want our interceptor to work </ClientHttpRequestInterceptor> </ClientHttpRequestInterceptor> a request/response logger, so we have to read it twice - the first time from the interceptor and the second time from the client. The default implementation allows you to read the response stream only once. To meet these specific scenarios, Spring provides a special class called BufferingClientHttpRequestFactory. As the name suggests, this class buffers the request/response in JVM memory for multiple use. The following shows how the RestTemplate object is initialized by using BufferingClientHttpRequestFactory to enable request/response stream caching: RestTemplate restTemplate - new RestTemplate(new BufferingClientHttpRequestFactory(new SimpleClientHttpRequestFactory()); 5. Testing this sample Here is the JUnit test case to test our RestTemplate interceptor: public class RestTemplateIntegrationTest - @Autowired RestTemplate restTemplate; @Test public void givenRestTemplate_whenRequested_thenLogAndModifyResponse() - LoginForm loginForm - new LoginForm(username, password); HttpEntity<LoginForm> requestEntity - new HttpEntity<LoginForm>(loginForm); HttpHeaders headers: new HttpHeaders(); headers.setContentType(MediaType.APPLICATION_JSON); ResponseEntity<String> responseEntity - restTemplate.postForEntity(requestEntity, String.class); assertThat(responseEntity.getStatusCode(), is(equalTo(HttpStatus.OK)); assertThat(responseEntity.getHeaders().get(Foo).get(0), is(equalTo(bar))); Here, we used the freely hosted HTTP request and response service to our data. This test service will return the request body along with some metadata. 6. Conclusion This tutorial is all about how to set up an interceptor and add it to the Object RestTemplate. This type of interceptors can also be used to filter, monitor, and control incoming requests. A common use case for a RestTemplate interceptor is the header change, detailed in this article. And, as always, you can find the sample code on on the Github project. It's a Maven-based project, so it should be as easy to import and run as it is. Generic fund I just announced the new Learn Spring course, focused on the fundamentals of Spring 5 and Spring Boot 2: >> CHECK OUT THE COURSE Sometimes it is useful to record HTTP requests and responses when working with a spring RestTemplate. If you need a granular check on exactly what is registered, you can use a custom interceptor to add recording before and after the remote call. Creating an Interceptor requires you to create a class that extends ClientHttpRequestInterceptor and implement the interception method as follows. @Slf4j @Component class LoggingInterceptor implementa ClientHttpRequestInterceptor @Override un'intercettazione Pubblica ClientHttpResponse (richiesta HttpRequest, byte[] body, esecuzione ClientHttpRequestExecution) genera IOException - logRequest(request, body); ClientHttpResponse risposta : execution.execute(richiesta, corpo); logResponse(risposta); </String></LoginForm></LoginForm> </String></LoginForm></LoginForm> the answer; } private void logRequest(HttpRequest request, byte[] body) throws IOException { if (log.isDebugEnabled()) { log.debug(============================request begin==); log.debug(URI : {}, request.getURI()); log.debug(Method : {}, request.getMethod()); log.debug(Headers : {}, request.getHeaders()); log.debug(Request body: {}, new String(body, UTF-8)); log.debug(============================request end==); } } private void logResponse(ClientHttpResponse response) throws IOException { if (log.isDebugEnabled()) { log.debug(== log.debug(Status code : {}, response.getStatusCode())); log.debug(Status text : {}, response.getStatusText()); log.debug(Headers : {}, response.getHeaders()); log.debug(Response body: {}, StreamUtils.copyToString(response.getBody(), Charset.defaultCharset())); log.debug(==); } } } When this class is registered with a RestTemplate Spring, it will call the intercept method before the request is sent, which allows you to register the request. In the logRequest method, I took the information from the request I want to log. Of course, you can record as much or as little as you like here. After registering the request, I call the execute method on the ClientHttpRequestExecution object to submit the request. When the response is received, log the status, headers, and body. Configuring RestTemplate To ensure that the interceptor is called, you must register it with RestTemplate . In the following example I only added the LoggingInterceptor, but you are free to add multiple interceptors and Spring chain them together for you at run time. @Bean public RestTemplate createRestTemplate(LoggingInterceptor loggingInterceptor, RestErrorHandler restErrorHandler) - RestTemplate restTemplate - new RestTemplate(); restTemplate.setInterceptors(Collections.singletonList(loggingInterceptor)); return restTemplate; Of course, you can only enable debug logging on the RestTemplate package, but I like to use an interceptor as if it gives you a granular control to record exactly what you want. To want.