**Not worked approach 1**

package com.reqres.filters;

import io.restassured.filter.Filter;

import io.restassured.filter.FilterContext;

import io.restassured.filter.log.RequestLoggingFilter;

import io.restassured.filter.log.ResponseLoggingFilter;

import io.restassured.specification.FilterableRequestSpecification;

import io.restassured.specification.FilterableResponseSpecification;

import io.restassured.filter.log.LogDetail;

import io.restassured.response.Response;

public class CustomLoggingFilter implements Filter {

private final LogDetail requestLogDetail;

private final LogDetail responseLogDetail;

private final boolean logOnFailureOnly;

private final String methodToLog;

private final Integer statusCodeToLog;

// Constructor to configure logging

public CustomLoggingFilter(LogDetail requestLogDetail, LogDetail responseLogDetail,

boolean logOnFailureOnly, String methodToLog, Integer statusCodeToLog) {

this.requestLogDetail = requestLogDetail;

this.responseLogDetail = responseLogDetail;

this.logOnFailureOnly = logOnFailureOnly;

this.methodToLog = methodToLog;

this.statusCodeToLog = statusCodeToLog;

}

@Override

public Response filter(FilterableRequestSpecification requestSpec,

FilterableResponseSpecification responseSpec,

FilterContext ctx) {

// Log Request if specified

if (requestLogDetail != null) {

new RequestLoggingFilter(requestLogDetail).filter(requestSpec, responseSpec, ctx);

}

// Proceed with the request and get the response

Response response = ctx.next(requestSpec, responseSpec);

// Log Response if specified

if (responseLogDetail != null) {

new ResponseLoggingFilter(responseLogDetail).filter(requestSpec, responseSpec, ctx);

}

// Log only on failure

if (logOnFailureOnly && (response.statusCode() < 200 || response.statusCode() >= 300)) {

new RequestLoggingFilter(LogDetail.ALL).filter(requestSpec, responseSpec, ctx);

new ResponseLoggingFilter(LogDetail.ALL).filter(requestSpec, responseSpec, ctx);

}

// Log specific HTTP Method or Status Code

if ((methodToLog != null && methodToLog.equalsIgnoreCase(requestSpec.getMethod())) ||

(statusCodeToLog != null && response.statusCode() == statusCodeToLog)) {

new RequestLoggingFilter(LogDetail.ALL).filter(requestSpec, responseSpec, ctx);

new ResponseLoggingFilter(LogDetail.ALL).filter(requestSpec, responseSpec, ctx);

}

return response;

}

// Static methods for common logging strategies (logAll, logHeadersOnly, logBodyOnly, etc.)

public static CustomLoggingFilter logAll() {

return new CustomLoggingFilter(LogDetail.ALL, LogDetail.ALL, false, null, null);

}

public static CustomLoggingFilter logHeadersOnly() {

return new CustomLoggingFilter(LogDetail.HEADERS, LogDetail.HEADERS, false, null, null);

}

public static CustomLoggingFilter logBodyOnly() {

return new CustomLoggingFilter(LogDetail.BODY, LogDetail.BODY, false, null, null);

}

public static CustomLoggingFilter logCookiesOnly() {

return new CustomLoggingFilter(LogDetail.COOKIES, LogDetail.COOKIES, false, null, null);

}

public static CustomLoggingFilter logUri() {

return new CustomLoggingFilter(LogDetail.URI, LogDetail.ALL, false, null, null);

}

public static CustomLoggingFilter logUriAndStatusOnly() {

return new CustomLoggingFilter(LogDetail.URI, LogDetail.STATUS, false, null, null);

}

public static CustomLoggingFilter logOnFailureOnly() {

return new CustomLoggingFilter(LogDetail.ALL, LogDetail.ALL, true, null, null);

}

public static CustomLoggingFilter logPostOnly() {

return new CustomLoggingFilter(LogDetail.ALL, LogDetail.ALL, false, "POST", null);

}

public static CustomLoggingFilter logStatusCode(int statusCode) {

return new CustomLoggingFilter(LogDetail.ALL, LogDetail.ALL, false, null, statusCode);

}

}

expalin from basics

// In your test

.filters(CustomLoggingFilter.logHeadersOnly())

// calls this

→ logHeadersOnly() {

return new CustomLoggingFilter(...);

}

// which calls this constructor

→ CustomLoggingFilter(requestLogDetail, responseLogDetail, ...)

**Not worked approach 2**

package com.reqres.filters;

import io.restassured.filter.Filter;

import io.restassured.filter.FilterContext;

import io.restassured.filter.log.LogDetail;

import io.restassured.response.Response;

import io.restassured.specification.FilterableRequestSpecification;

import io.restassured.specification.FilterableResponseSpecification;

public class RequestLoggingFilter implements Filter {

private final LogDetail logDetail;

private final boolean logOnFailureOnly;

private final String methodToLog;

public RequestLoggingFilter(LogDetail logDetail, boolean logOnFailureOnly, String methodToLog) {

this.logDetail = logDetail;

this.logOnFailureOnly = logOnFailureOnly;

this.methodToLog = methodToLog;

}

@Override

public Response filter(FilterableRequestSpecification requestSpec,

FilterableResponseSpecification responseSpec,

FilterContext ctx) {

// Log the request based on config BEFORE calling ctx.next

if (logDetail != null) {

logRequest(requestSpec, logDetail);

}

// Proceed with the request

Response response = ctx.next(requestSpec, responseSpec);

// Log on failure (after getting response)

if (logOnFailureOnly && (response.statusCode() < 200 || response.statusCode() >= 300)) {

logRequest(requestSpec, LogDetail.ALL);

}

// Log only specific method (e.g., GET)

if (methodToLog != null && methodToLog.equalsIgnoreCase(requestSpec.getMethod())) {

logRequest(requestSpec, LogDetail.ALL);

}

return response;

}

private void logRequest(FilterableRequestSpecification requestSpec, LogDetail detail) {

try {

switch (detail) {

case ALL:

requestSpec.log().all();

break;

case BODY:

requestSpec.log().body();

break;

case HEADERS:

requestSpec.log().headers();

break;

case COOKIES:

requestSpec.log().cookies();

break;

case PARAMS:

requestSpec.log().params();

break;

default:

// No logging

break;

}

} catch (Exception e) {

System.err.println("Error while logging request: " + e.getMessage());

e.printStackTrace();

}

}

// Factory methods

public static RequestLoggingFilter logAll() {

return new RequestLoggingFilter(LogDetail.ALL, false, null);

}

public static RequestLoggingFilter logHeadersOnly() {

return new RequestLoggingFilter(LogDetail.HEADERS, false, null);

}

public static RequestLoggingFilter logBodyOnly() {

return new RequestLoggingFilter(LogDetail.BODY, false, null);

}

public static RequestLoggingFilter logOnFailure() {

return new RequestLoggingFilter(null, true, null);

}

public static RequestLoggingFilter logForMethod(String method) {

return new RequestLoggingFilter(null, false, method);

}

}

package com.reqres.filters;

import io.restassured.filter.Filter;

import io.restassured.filter.FilterContext;

import io.restassured.filter.log.LogDetail;

import io.restassured.response.Response;

import io.restassured.specification.FilterableRequestSpecification;

import io.restassured.specification.FilterableResponseSpecification;

public class ResponseLoggingFilter implements Filter {

private final LogDetail logDetail;

private final boolean logOnFailureOnly;

private final Integer statusCodeToLog;

public ResponseLoggingFilter(LogDetail logDetail, boolean logOnFailureOnly, Integer statusCodeToLog) {

this.logDetail = logDetail;

this.logOnFailureOnly = logOnFailureOnly;

this.statusCodeToLog = statusCodeToLog;

}

@Override

public Response filter(FilterableRequestSpecification requestSpec,

FilterableResponseSpecification responseSpec,

FilterContext ctx) {

// Proceed with the request and get the response

Response response = ctx.next(requestSpec, responseSpec);

// Log response if specified

if (logDetail != null) {

logResponse(response, logDetail);

}

// Log only on failure if specified

if (logOnFailureOnly && (response.statusCode() < 200 || response.statusCode() >= 300)) {

logResponse(response, LogDetail.ALL);

}

// Log specific status code if specified

if (statusCodeToLog != null && response.statusCode() == statusCodeToLog) {

logResponse(response, LogDetail.ALL);

}

return response;

}

private void logResponse(Response response, LogDetail detail) {

try {

switch (detail) {

case ALL:

response.then().log().all();

break;

case BODY:

response.then().log().body();

break;

case HEADERS:

response.then().log().headers();

break;

case COOKIES:

response.then().log().cookies();

break;

case PARAMS:

response.then().log().params();

break;

default:

// No logging

break;

}

} catch (Exception e) {

System.err.println("Error while logging response: " + e.getMessage());

e.printStackTrace();

}

}

// Factory methods for common use cases

public static ResponseLoggingFilter logAll() {

return new ResponseLoggingFilter(LogDetail.ALL, false, null);

}

public static ResponseLoggingFilter logHeadersOnly() {

return new ResponseLoggingFilter(LogDetail.HEADERS, false, null);

}

public static ResponseLoggingFilter logBodyOnly() {

return new ResponseLoggingFilter(LogDetail.BODY, false, null);

}

public static ResponseLoggingFilter logOnFailure() {

return new ResponseLoggingFilter(null, true, null);

}

public static ResponseLoggingFilter logForStatusCode(int statusCode) {

return new ResponseLoggingFilter(null, false, statusCode);

}

}

**Worked from calude**

package com.reqres.filters;

import io.restassured.filter.Filter;

import io.restassured.filter.FilterContext;

import io.restassured.filter.log.LogDetail;

import io.restassured.response.Response;

import io.restassured.specification.FilterableRequestSpecification;

import io.restassured.specification.FilterableResponseSpecification;

public class RequestLoggingFilter implements Filter {

private final LogDetail logDetail;

private final boolean logOnFailureOnly;

private final String methodToLog;

public RequestLoggingFilter(LogDetail logDetail, boolean logOnFailureOnly, String methodToLog) {

this.logDetail = logDetail;

this.logOnFailureOnly = logOnFailureOnly;

this.methodToLog = methodToLog;

}

@Override

public Response filter(FilterableRequestSpecification requestSpec,

FilterableResponseSpecification responseSpec,

FilterContext ctx) {

// First check if we should log based on method

boolean shouldLog = methodToLog == null ||

methodToLog.equalsIgnoreCase(requestSpec.getMethod());

// Log request details before sending if configured to do so and method matches

if (shouldLog && logDetail != null && !logOnFailureOnly) {

logRequest(requestSpec, logDetail);

}

// Proceed with the request - avoid manipulating the filter chain during iteration

Response response = ctx.next(requestSpec, responseSpec);

// After getting response, check if we should log on failure

if (shouldLog && logOnFailureOnly && (response.statusCode() < 200 || response.statusCode() >= 300)) {

logRequest(requestSpec, LogDetail.ALL);

}

return response;

}

private void logRequest(FilterableRequestSpecification requestSpec, LogDetail detail) {

try {

switch (detail) {

case ALL:

System.out.println("REQUEST: " + requestSpec.getMethod() + " " + requestSpec.getURI());

System.out.println("Headers: " + requestSpec.getHeaders());

if (requestSpec.getBody() != null) {

System.out.println("Body: " + requestSpec.getBody().toString());

}

break;

case HEADERS:

System.out.println("REQUEST HEADERS: " + requestSpec.getHeaders());

break;

case BODY:

if (requestSpec.getBody() != null) {

System.out.println("REQUEST BODY: " + requestSpec.getBody().toString());

}

break;

case COOKIES:

System.out.println("REQUEST COOKIES: " + requestSpec.getCookies());

break;

case PARAMS:

System.out.println("REQUEST PARAMS: " + requestSpec.getQueryParams());

break;

default:

// No logging

break;

}

} catch (Exception e) {

System.err.println("Error while logging request: " + e.getMessage());

}

}

// Factory methods

public static RequestLoggingFilter logAll() {

return new RequestLoggingFilter(LogDetail.ALL, false, null);

}

public static RequestLoggingFilter logHeadersOnly() {

return new RequestLoggingFilter(LogDetail.HEADERS, false, null);

}

public static RequestLoggingFilter logBodyOnly() {

return new RequestLoggingFilter(LogDetail.BODY, false, null);

}

public static RequestLoggingFilter logOnFailure() {

return new RequestLoggingFilter(null, true, null);

}

public static RequestLoggingFilter logForMethod(String method) {

return new RequestLoggingFilter(null, false, method);

}

}

package com.reqres.filters;

import io.restassured.filter.Filter;

import io.restassured.filter.FilterContext;

import io.restassured.filter.log.LogDetail;

import io.restassured.response.Response;

import io.restassured.specification.FilterableRequestSpecification;

import io.restassured.specification.FilterableResponseSpecification;

public class ResponseLoggingFilter implements Filter {

private final LogDetail logDetail;

private final int maxStatusCode;

public ResponseLoggingFilter(LogDetail logDetail, int maxStatusCode) {

this.logDetail = logDetail;

this.maxStatusCode = maxStatusCode;

}

@Override

public Response filter(FilterableRequestSpecification requestSpec,

FilterableResponseSpecification responseSpec,

FilterContext ctx) {

// Execute the request first

Response response = ctx.next(requestSpec, responseSpec);

// Then log response based on configuration

boolean shouldLog = maxStatusCode <= 0 || response.statusCode() <= maxStatusCode;

if (shouldLog && logDetail != null) {

logResponse(response, logDetail);

}

return response;

}

private void logResponse(Response response, LogDetail detail) {

try {

switch (detail) {

case ALL:

System.out.println("RESPONSE Status: " + response.getStatusCode() + " " + response.getStatusLine());

System.out.println("Headers: " + response.getHeaders());

System.out.println("Body: " + response.asPrettyString());

break;

case HEADERS:

System.out.println("RESPONSE HEADERS: " + response.getHeaders());

break;

case BODY:

System.out.println("RESPONSE BODY: " + response.asPrettyString());

break;

case STATUS:

System.out.println("RESPONSE STATUS: " + response.getStatusCode() + " " + response.getStatusLine());

break;

default:

// No logging

break;

}

} catch (Exception e) {

System.err.println("Error while logging response: " + e.getMessage());

}

}

// Factory methods

public static ResponseLoggingFilter logAll() {

return new ResponseLoggingFilter(LogDetail.ALL, 0);

}

public static ResponseLoggingFilter logHeadersOnly() {

return new ResponseLoggingFilter(LogDetail.HEADERS, 0);

}

public static ResponseLoggingFilter logBodyOnly() {

return new ResponseLoggingFilter(LogDetail.BODY, 0);

}

public static ResponseLoggingFilter logStatusOnly() {

return new ResponseLoggingFilter(LogDetail.STATUS, 0);

}

public static ResponseLoggingFilter logOnlySuccessResponses() {

return new ResponseLoggingFilter(LogDetail.ALL, 299);

}

}

**CustomLoggingFilter.java which line 1st is called**

public Response filter(FilterableRequestSpecification requestSpec,

FilterableResponseSpecification responseSpec,

FilterContext ctx)

explain this new RequestLoggingFilter(requestLogDetail).filter(requestSpec, responseSpec, ctx);

**What’s happening here:**

* new RequestLoggingFilter(requestLogDetail)  
  → This creates a new instance of Rest Assured's RequestLoggingFilter.  
  → You pass it a LogDetail, which tells it *what part* of the request you want to log (e.g., HEADERS, BODY, ALL, etc.).
* .filter(requestSpec, responseSpec, ctx)  
  → After creating the filter, you're immediately calling its filter method.  
  → This filter method actually **logs the request** based on the LogDetail you gave earlier.  
  → It uses:
  + requestSpec → the request that is about to be sent
  + responseSpec → the expected response structure (though here it's not modified)
  + ctx (FilterContext) → allows passing the request to the next filter or executing it.

**In simple words:**  
✅ You are manually telling Rest Assured to **log the request** at this point,  
✅ Choosing what to log (headers, body, URI, etc.)  
✅ Using the provided request and context.

**Why use it like this?**

Normally, in Rest Assured you can set up filters globally or automatically.  
But in your CustomLoggingFilter, you want **fine control** — when and what exactly to log — so you call the .filter() manually inside your custom filter.

**my question .filters(CustomLoggingFilter.logHeadersOnly()) // Customize as per your need (logBodyOnly, logAll, etc.)**

**is given part who it is handled in the respsoe as till not tiggred**

— which is **given** before .when().get(),  
can also **affect the response**, even though the response hasn't happened yet!  
(**In short:** How can it handle something that has not happened yet?)

Let’s go step-by-step:

1. **At .given() phase**  
   When you call .filters(CustomLoggingFilter.logHeadersOnly()),  
   you are just **registering** the filter with the request.  
   You are telling Rest Assured:

"Hey, when this request runs, use this custom filter to intercept the full flow."

✅ **Nothing is executed yet.**  
✅ The filter is just **attached** to the request at this point.

1. **At .when().get() phase**  
   Now when you call .when().get("/users/2"), **Rest Assured starts processing the request**.

Before actually sending the HTTP call to server, it checks:

* Are there any filters registered on this request?
* Oh yes! CustomLoggingFilter is there.

Then it **calls the filter() method** inside your CustomLoggingFilter:

public Response filter(FilterableRequestSpecification requestSpec,

FilterableResponseSpecification responseSpec,

FilterContext ctx) {

// here request logging happens

// then ctx.next(requestSpec, responseSpec) sends the actual request

// then response logging happens

}

✅ First inside filter(), **request logging** happens.

✅ Then .next() is called — this is the moment **real HTTP request** is sent.

✅ After getting the response, **response logging** happens inside the same filter() method.

**In simple words:**

* .filters() **registers** your custom behavior.
* Only when .when().get() is called, your filter **runs** — handling **both request and response** inside **the same filter method**.

**public Response filter(FilterableRequestSpecification requestSpec, FilterableResponseSpecification responseSpec, FilterContext ctx) {**

**what is FilterableRequestSpecification , FilterableResponseSpecification , FilterContext**

**1. FilterableRequestSpecification**

* It’s an **interface** that extends RequestSpecification.
* **Represents** the **request** that is *about to be sent*.
* Gives you **full control** over the request:
  + You can **read** details (method, URL, headers, body, etc.)
  + You can **modify** the request before sending it if you want.

✅ Example of what you can access through requestSpec:

requestSpec.getMethod(); // GET, POST, etc.

requestSpec.getURI(); // https://reqres.in/api/users/2

requestSpec.getHeaders(); // all request headers

requestSpec.getBody(); // request body

requestSpec.addHeader("key", "value"); // add new header

**2. FilterableResponseSpecification**

* It’s an **interface** extending ResponseSpecification.
* **Represents** the **expected behavior** or **rules** about the response *before* it's received.
* (⚡ In normal usage, you don't touch it much inside a custom filter.)
* Can be used for setting up **validations** or **expectations** (but usually, it’s more for chaining).

responseSpec.getContentType(); // expected content-type, etc.

**. FilterContext**

* The **magic object** that connects **request -> real server -> response**.
* It’s the **execution pipeline** control.

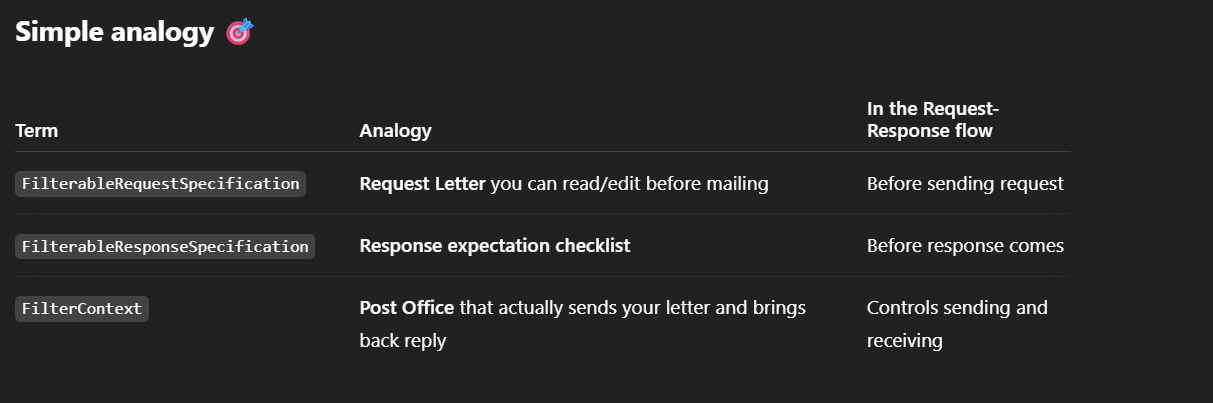
ctx.next(requestSpec, responseSpec);

to **actually send** the HTTP request to server and get the real **response** back.

* If you **don't call** ctx.next(), the request will **never** be sent.

✅ What happens inside ctx.next():

* Sends requestSpec to the server
* Waits for the response
* Applies any post-processing



**In super short:**

* requestSpec → The Request you can **see or change** before sending.
* responseSpec → The expected behavior of the Response (rarely used inside filters).
* ctx → **Actually sends** the request and **returns** the response.

**FilterContext ctx**

**Response response = ctx.next(requestSpec, responseSpec);**

**not able to undesrtand it**

Response response = ctx.next(requestSpec, responseSpec);

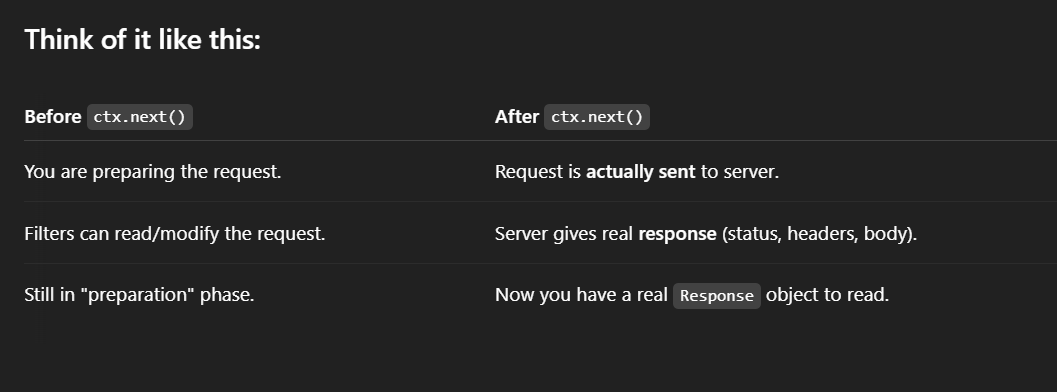
This line is **SUPER IMPORTANT**.  
Let’s explain it clearly:

**What is happening with ctx.next(requestSpec, responseSpec)?**

* ctx is the **FilterContext**.
* next() means:  
  👉 "**Now please send the actual HTTP request to the server!**" 👉 "**Then, return me the Response that comes back!**"

✅ It is the **trigger point** where:

* The HTTP request (GET/POST/PUT etc.) is actually **executed**
* The real server (like https://reqres.in) **receives** it
* A real HTTP **response** comes back
* You get a Response Java object to work with



**Very simple analogy:**

* requestSpec = your **email draft** (you are writing)
* ctx.next() = you **click "Send"**
* response = you **get a reply email** from the recipient.

// 1. You may log or modify the request

logRequestDetails(requestSpec);

// 2. ctx.next(...) sends the request and waits for response

Response response = ctx.next(requestSpec, responseSpec);

// 3. Now you can log or check the actual response

logResponseDetails(response);

**Summary:**

ctx.next(requestSpec, responseSpec)  
➡️ **Sends** the request  
➡️ **Returns** the real Response from server  
➡️ Without calling ctx.next(), the HTTP call would **never happen**.

✅ So, **before ctx.next**, you are just *preparing* things.  
✅ **After ctx.next**, you *have the real response* and can *log* it or *validate* it.

**Response response = ctx.next(requestSpec, responseSpec);**

**when it is called ?**

**When is ctx.next(requestSpec, responseSpec) called?**

👉 It is called **inside your filter() method**.  
👉 **When your filter is executing**, and **you** explicitly tell Rest Assured:

"**Ok, now send the request to the server.**"

That moment is **when ctx.next(...) is called**.

**Full Flow**

When you run your test:

java

CopyEdit

Response response = given()

.spec(getRequestSpec())

.filters(CustomLoggingFilter.logHeadersOnly())

.when()

.get("/users/2")

.then()

.extract()

.response();

Here's what happens internally:

1. .filters(CustomLoggingFilter.logHeadersOnly()) → You **attached** your custom filter to this request.
2. .when().get("/users/2") → Rest Assured says:

"Hey, there’s a filter attached! I need to execute its filter() method."

1. Inside your CustomLoggingFilter.filter(...):
   * First, you **log** the request (optional).
   * Then **YOU** manually call:

java

CopyEdit

Response response = ctx.next(requestSpec, responseSpec);

* + **At this moment**, Rest Assured:
    - sends the real HTTP call to the server (GET /users/2)
    - waits for the server's reply
    - captures the Response

1. After ctx.next() finishes:
   * You now have the actual Response from server.
   * Then you can log the response, validate it, or return it.

**In very simple words:**

ctx.next(...) is **called inside the filter** by **your code**,  
when **you** are ready to **send the HTTP request** to the server.

**You control it manually.**

**A Mini Example**

@Override

public Response filter(FilterableRequestSpecification requestSpec,

FilterableResponseSpecification responseSpec,

FilterContext ctx) {

System.out.println("👉 Request is being prepared...");

// Send the real HTTP request

Response response = ctx.next(requestSpec, responseSpec);

System.out.println("👉 Response received!");

return response;

}

When your test runs, it will print:

👉 Request is being prepared...

👉 (Request sent to server, waiting...)

👉 Response received!

**Conclusion**

* ctx.next(requestSpec, responseSpec) is called **inside your filter**,
* **At the point where you decide** to send the HTTP request,
* **Triggered automatically** when .when().get() or .post() happens.

**In debug mode,  
after .spec(...), .filters(...),  
immediately, CustomLoggingFilter.filter(...) is getting called —  
meaning even before .when().get(...) is fully visible.**

**So you're wondering:**

* **Is response being fetched too early?**
* **Is some information missing because .when() execution is not "clearly" separate?**

✅ **You are half-correct**, but there’s a super important thing to understand:

**The truth about Rest Assured execution:**

When you write:

java

CopyEdit

given()

.spec(...)

.filters(...)

.when()

.get("/users/2")

🔵 **All of this is one *big chained command*.**  
It's **prepared together first**, but **the real network call happens ONLY when .when().get() is reached**.

⚡ **In Java**, method chaining like this:

java

CopyEdit

given().something().something().when().something()

**does not immediately execute** after each .something().  
It **collects configuration**, **builds internal object**, and **finally acts** when .when().get() triggers.

🔵 **But** — internally, the way Rest Assured is implemented:

* .given() returns a **RequestSpecification object**.
* .filters(...) **modifies** the RequestSpecification (adds filter to the list).
* .when() **calls internal execution**, which **processes** filters, starting with filter().
* As part of executing the **first filter**, **inside filter()**, you call ctx.next(requestSpec, responseSpec), which **sends** the request.

⚡ In other words:  
➡️ **Stepping through .filters(...) is not execution yet**,  
➡️ **Stepping into .filter() is starting execution triggered by .when().get().**

**Answer to your doubt:**

* Even though you placed the **debugger** in .filters(...),
  + **Rest Assured actually delays execution**.
  + **Real sending happens** when you hit .when() and inside your **custom filter’s** ctx.next().
* You are **not missing any request information**.
* **All method, path, path params, etc.** are fully built inside requestSpec *before* ctx.next() is called.

✅ Your response will **have all correct info** — no partial data.

**Why it looks confusing during debug:**

When you step through debugger slowly:

* .filters(CustomLoggingFilter.logHeadersOnly())
* You *enter* CustomLoggingFilter because .when().get(...) immediately calls the filters.
* So it "feels" like execution is happening at .filters(...),  
  ➔ but **internally it was triggered by .when().get()**, because it started the call chain.

**It's a visual/debugger illusion!**

**Visual flow you are seeing (more exact):**

given() <-- prepare spec

.spec() <-- apply spec

.filters() <-- attach filter

.when() <-- TRIGGERS execution

|

---> your CustomLoggingFilter.filter()

|

---> log request

---> ctx.next(requestSpec, responseSpec) <-- SEND request

---> log response

✅ So again:

* Your ctx.next() is **correctly called after all configuration is ready**.
* **Nothing is missing in your request**.
* **Response is correct and complete**, even for .head(), .options(), etc.

**In Super Short:**

| **Myth** | **Reality** |
| --- | --- |
| Filters execute immediately when .filters(...) is called | No! Filters are **registered** then, but **executed** after .when().get() triggers the call |
| Response is missing info because filter looks early | No! Full info is prepared in requestSpec, and response is properly populated |