* Use a modular structure — not a single Java file.
* Group filters by concern or category, placing them in logically separated classes/files. This keeps your code clean, readable, testable, and extensible.

/src/test/java

│

├── /filters ← 📂 Main package for custom filters

│ ├── BasicLoggingFilters.java ← RequestLoggingFilter, ResponseLoggingFilter shortcuts

│ ├── ConditionalFilters.java ← For filters with `.when()` logic

│ ├── SecurityLoggingFilter.java ← Masks sensitive fields

│ ├── PerformanceLoggingFilter.java ← Times requests, flags slow ones

│ ├── RetryLoggingFilter.java ← Behavioral log (e.g. retry attempts)

│ └── ExternalLogger.java ← Writes logs to file, DB, etc.

│

├── /utils

│ └── ConfigUtil.java ← Load config/env flags, e.g. enableLogging

│

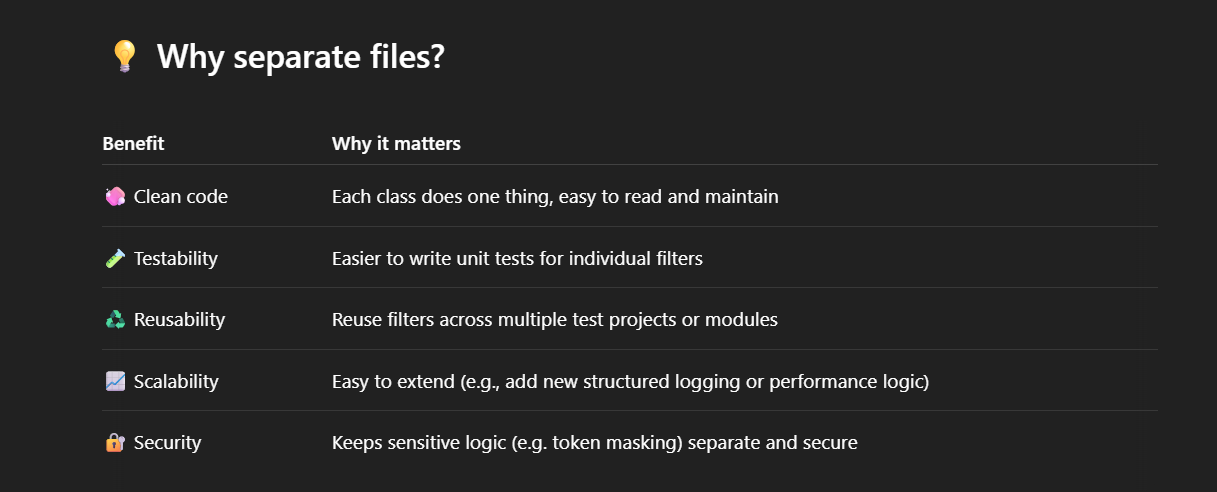
├── /listeners

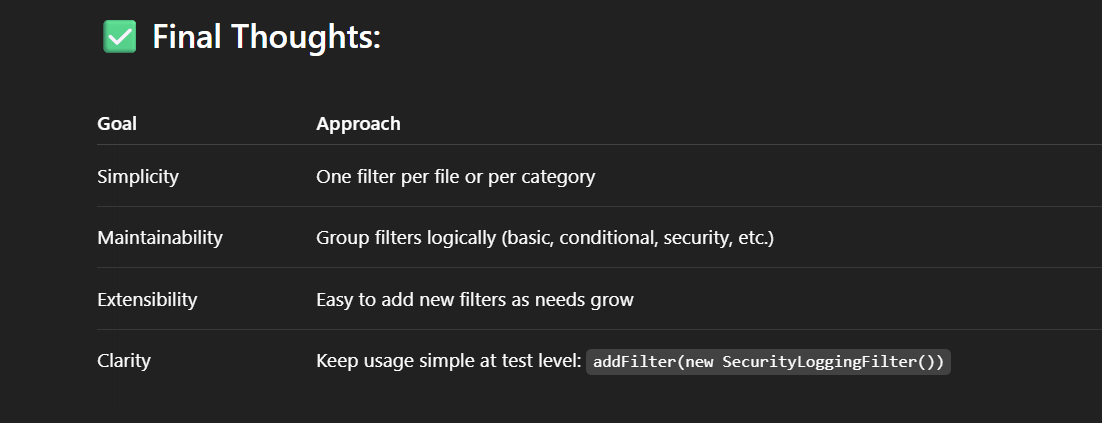
│ └── FailureLoggingListener.java ← Logs only on failure using TestNG hooks

│

└── /tests

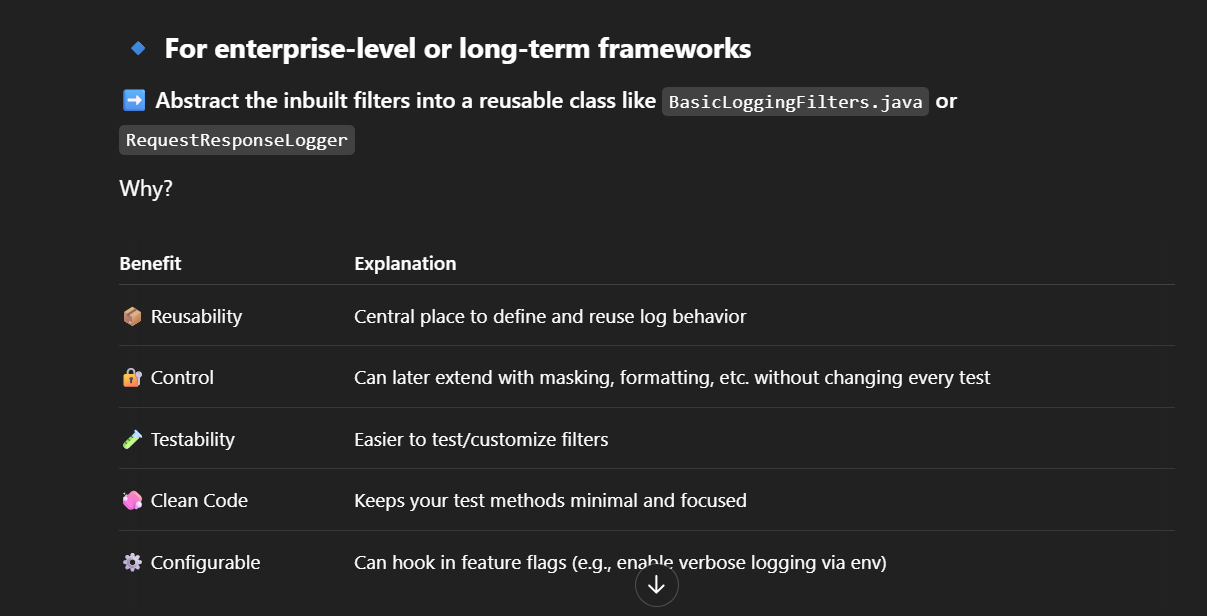
└── YourTestClass.java





**BasicLoggingFilters (RequestResponseLogger)**

Why



**Conclusion: Industry standard = abstraction**

* Yes — create RequestResponseLogger.java (or BasicLoggingFilters.java) as a wrapper over RequestLoggingFilter and ResponseLoggingFilter.
* Create RequestResponseLogger.java to house reusable logging combinations.
* Use .filters(...) in test code.
* Extend later with custom logic (masking, conditional logging, slow-request alerts).

**Suggested Methods for RequestResponseLogger**

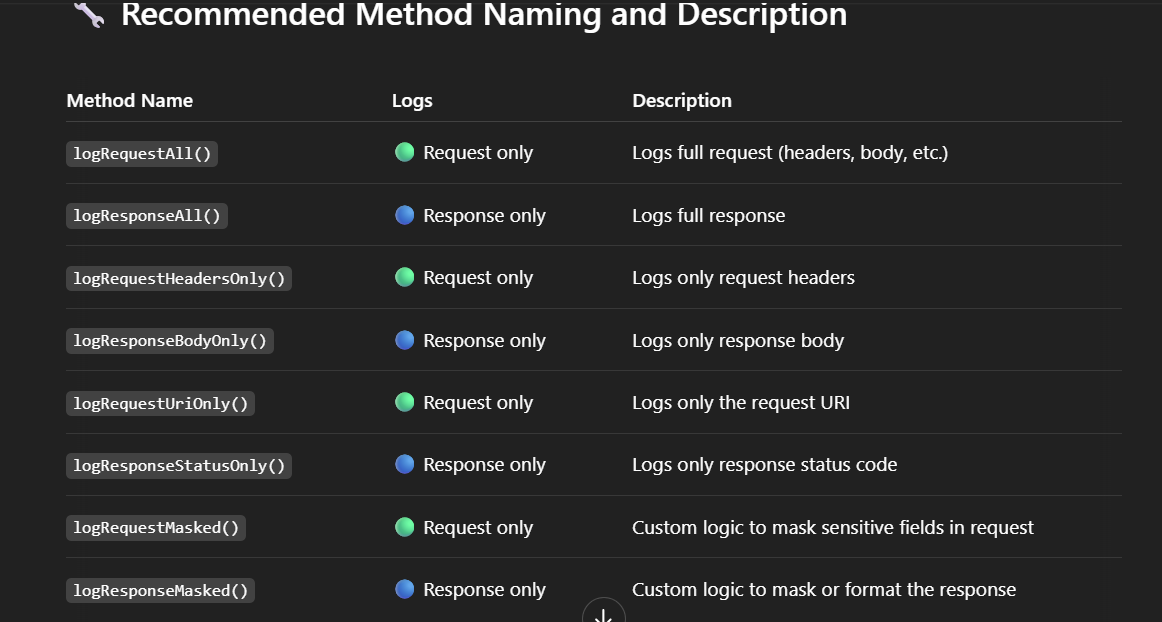
| **Method Name** | **Description** |
| --- | --- |
| logAll() | Logs everything (headers, body, cookies, etc.) for both request and response. Useful during full debug. |
| logHeadersOnly() | Logs only the headers of request and response. Lightweight and secure. |
| logBodyOnly() | Logs only the body of request and response. Useful when debugging payload issues. |
| logCookiesOnly() | Logs only the cookies. Less used, but helpful when cookie-based auth is involved. |
| logUriAndStatusOnly() | Logs request URI and response status. Fast and concise, great for CI environments. |
| logOnFailureOnly() | Logs request and response only if the test fails (can integrate with TestNG/JUnit listeners). |
| logPostOnly() | Logs only when the request method is POST. Useful to avoid cluttering logs for GETs. |
| logStatusCode(int statusCode) | Logs only when the response returns a specific status code (e.g., 500). |
| logMasked() | Logs request/response but masks sensitive data (e.g., passwords, tokens). Needs custom implementation. |
| logWithTimer() | Logs along with timing information — useful to find slow APIs. |
| logToFile(String fileName) | Sends all logs to a file instead of the console. Great for CI or debugging failed builds. |

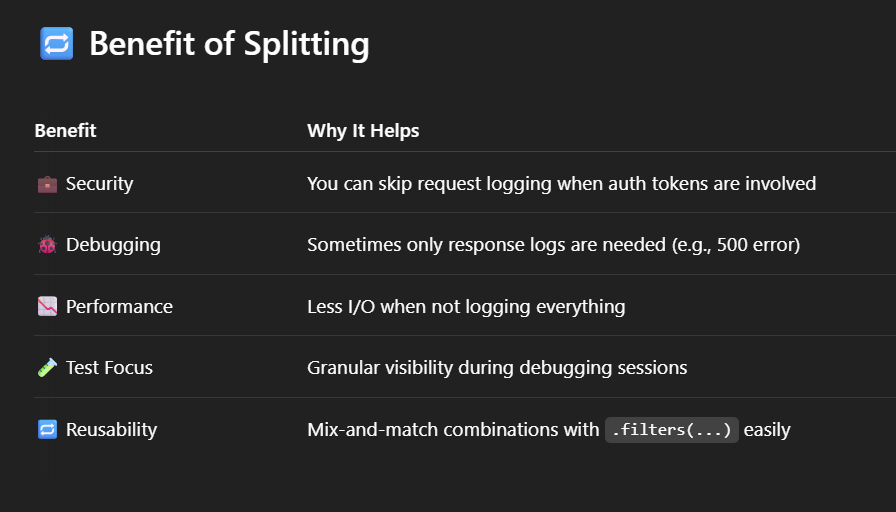
**RequestResponseLogger**

Yes, the current method list I gave **combines** both **request and response logging** in most cases. That’s ideal for full visibility, but sometimes you **only want to log one side** — for example:

* ✅ **Only request logging** when debugging payloads.
* ✅ **Only response logging** when verifying backend behavior.
* ✅ Avoiding request logging to **protect sensitive info** (like tokens).

**Yes — add additional methods for logging only request or only response**.  
These give you **flexibility and control**, and it aligns with industry-level test design (especially in secure or performance-conscious environments).





**public** **static** Filter logRequestAll() {

**return** **new** RequestLoggingFilter(LogDetail.ALL);

}

**public** **static** Filter logResponseAll() {

**return** **new** ResponseLoggingFilter(LogDetail.ALL);

}

**public** **static** Filter logRequestHeadersOnly() {

**return** **new** RequestLoggingFilter(LogDetail.HEADERS);

}

**public** **static** Filter logResponseBodyOnly() {

**return** **new** ResponseLoggingFilter(LogDetail.BODY);

}

**Final Takeaway**

✅ **Yes — add both combined and separate (request-only / response-only) methods.**  
This gives you maximum flexibility and aligns with real-world QA and test automation frameworks.

**public static Filter vs public static List<Filter>**

 **Use List<Filter> for standard logging setups** (since your tests share the same logging behavior).

 **Tests are isolated**: Even though the logging is the same across tests, each test method gets its own filter setup independently.

 **Maintainability**: You can easily add, update, or change your logging strategy without needing to modify individual test methods.

## Best Practices <a name="best-practices"></a>

1. **Production vs Development Logging**:
   * Use detailed logging (logAll()) in development
   * Switch to logOnFailureOnly() in production tests
2. **Performance Considerations**:
   * Excessive logging can slow down tests
   * Consider logUriAndStatusOnly() for performance tests
3. **Sensitive Data**:
   * Be careful with logAll() in production - might log credentials
   * Consider creating a filter that redacts sensitive information
4. **Organizing Filters**:
   * Create reusable filter instances
   * Store common configurations as constants