Boilerplate logging: Logs for developers by developers

Homework 5

EVAN LEE, University of Colorado Boulder

The logging aspect of any software project always incurs a non-zero developer investment, both from a development and testing perspective as well as a operational and error tracing perspective. A significant portion of the thought and energy behind logging decisions arise from nearly-universal problems and use cases and yet still require each distinct developer to undergo the same rote process of decision making. Even with these factors and costs in play, logging is typically not considered as a first-class issue which leads to hidden cost in terms of development cycles or unnecessary iteration to assist in tracing and debugging. After having identified a widely realized use case of logging in dev-test cycles, I introduced and defined the concept of "boilerplate logging" and developed a python logging library that largely abstracts the human component to logging decisions, simultaneously solving the hidden up-front development cost in logging decision making as well as eliminating any future costs in iterating on reactive postmortem logging.

Additional Key Words and Phrases: logging, boilerplate

ACM Reference Format:
Evan Lee. 2020. Boilerplate logging: Logs for developers by developers: Homework 5. *ACM Forthcoming* CSCI 5535, Spring 2020 (April 2020), 5 pages.

1 INTRODUCTION

Logging as a language construct is a double-edged sword; developers want all the information they can get for use in troubleshooting, debugging, and development, but this often comes as a latent cost in terms of either lines of code or in logical paths designed specifically to handle development logging use cases. This causes a balancing act: certainly early into the development cycle, logging statements are vastly prevalent in code and the statements themselves are used to great effect on a regular basis. However, this pattern shifts as code becomes more mature. Production processes almost certainly deactivate nearly all of the logs put in place for development and leave lines of code orphaned and without use. In some situations – for the sake of performance or cleanliness – additional iteration is performed with the sole purpose of removing log lines that are no longer needed. This balancing act can be represented as a spectrum: from one side being extremely verbose logs to the other side being total lack of logs and only application code.

All of this activity is largely universal in the development world and leads to countless development cycles and engineering hours hidden away from the bottom line. In addition, some cases present themselves where application logic is even warped or influenced by the presence of logging which should be considered a second-class citizen when compared to core functionality and logic; take an example where code must be written to decide what a log message should itself say or be formatted as.

To reiterate: the cost of managing this balancing act is hidden at best and outright obscured at worst. On an individual basis, even considering changes to code that have to do with so-called boilerplate logging may incur a small, nearly insignificant cost but these small costs add up over the span of the industry and especially over the span of time. In addition, logging at its heart is a subjective activity. Different developers from different backgrounds using different languages

Author's address: Evan Lee, University of Colorado Boulder, Boulder, Colorado, evan.n.lee@colorado.edu.

with different frameworks all will use unique styles of logging and make unique decisions on log positions, formatting, substance, and level. This presents difficulty in the qualitative analysis of boilerplate logging broadly and generally throws a wrench into any standardization activity.

My contribution to this effort is two-fold: a collection and analysis of log usage in widely-used open-source libraries as well as an implementation that displays the ideas and positions presented below. Popular open-source libraries and tools are of particular interest in this space as not only do they have a wide range of contributors, but there is a communal sense of "being a good neighbor" when writing open-source code; particular care is given to the logging statements that are eventually committed and merged into the mainstream base and thus particular thought and effort is given to the balancing act between verbosity and utility.

2 OVERVIEW

In order to assist the reader in visualizing the analysis and ideas that this paper presents as well as the benefits of implementation, we will cover a code example from a widely-used python library for interacting with SOAP interfaces: python–zeep. The code snippet can be found in Listing 1. [1] Reading through this 31-line snippet, most of the lines written by the developer deal with logging in one way or another, whether that is making logging decisions (lines 2-5, 12-22) or dealing with emitting log messages themselves (lines 6, 24). In fact, only three lines (lines 8-10) deal with any actual functionality that this method is meant to drive.

One of the explicit goals of boilerplate logging is to provide key information to developers at significant breakpoints in code. The log messages in lines 6 and 24 are concerned mainly with two channels of boilerplate log statements – line 6 provide a message for the inputs of the example method and line 24 provides a message for the output of the example method. So while the format of the logging output may differ from Snippet 1 to Snippet 2, the code can be refactored to instead use boilerplate logging to accomplish the same goals by activating the required channels within the boilerplate logger itself.

Take the output of the log message as-written on line 6:

```
HTTP Post to https://api.colorado.edu/:
{"test": "test"}
```

and take the output of a boilerplate logger log message that would be emitted upon the invocation of the post method:

```
'post' CALLED WITH:
address=https://api.colorado.edu/
message={"test": "test"}
headers={}
```

The formatting of the messages differ, but the same set of information is imparted to the reader.

In addition, consider the additional benefit that boilerplate logging provides in the above example. Take a situation where a future maintainer of the code must also capture the header object being provided to the post method in the logs for any reason; now an additional iteration of code must be developed and committed.

[1] Source: python-zeep/transports.py, lines 47-84.

```python
    def post(self, address, message, headers):
        if self.logger.isEnabledFor(logging.DEBUG):
            log_message = message
            if isinstance(log_message, bytes):
                log_message = log_message.decode("utf-8")
            self.logger.debug("HTTP Post to %s:\n%s", address, log_message)

        response = self.session.post(
            address, data=message, headers=headers, timeout=self.operation_timeout
        )

        if self.logger.isEnabledFor(logging.DEBUG):
            media_type = get_media_type(
                response.headers.get("Content-Type", "text/xml")
            )

            if media_type == "multipart/related":
                log_message = response.content
            else:
                log_message = response.content
                if isinstance(log_message, bytes):
                    log_message = log_message.decode(response.encoding or "utf-8")

            self.logger.debug(
                "HTTP Response from %s (status: %d):\n%s",
                address,
                response.status_code,
                log_message,
            )

        return response
```

Listing 1. An example of code explosion due to boilerplate logging.

```
    import boilerplate as bp

    """
    Not passing anything or passing "all" is the same as:
    @bp.log("inputs")
    @bp.log("outputs")
    """

    @bp.log
    def post(self, address, message, headers):
        response = self.session.post(
            address, data=message, headers=headers, timeout=self.operation_timeout
        )
        return response
```

Listing 2. The same code as Listing 1, but using the boilerplate logger.

The contributions of this paper include:

- The definition and boundaries of boilerplate logging.
- The concept of "channels" which are sub-categories of boilerplate logging with different levels and formats.
- Data and analysis of widely-used open source projects implemented in python and the characterization of logging statements within.
- An implementation of the one of the idea presented in a Python 3+ logging library.

3 RESEARCH & ANALYSIS

4 IMPLEMENTATION

5 EMPIRICAL EVALUATION

6 RELATED WORK

6.1 Standarization.

The industry has made similar efforts in the area of standarization of logs, which really deals with the abstraction of the human element away and instead introduces a controlled process around software logging. Some ideas introduced are the concepts of log composition [2] and enhancement [4]. However, this research deals explicitly with the idea of standardization through total abstraction, whereas similar efforts continue to leave an aspect of the human element in the practice of logging.

6.2 Decision making.

There is more interest in understanding the motivation and usage behind log statements. Other efforts include attempting to understand the justification behind the locations developers place logging statements [1] and the quality and characteristics behind logging statements themselves [3]. This research takes a similar approach to perform its analysis but utilizes the insights gleaned in a more naïve context, preferring instead a more black-and-white, dumb approach to categorization or use case analysis.

7 CONCLUSION

ACKNOWLEDGMENTS

TBD

REFERENCES

[1] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? an empirical study on logging practices in industry. In *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 24–33. https://doi.org/10.1145/2591062.2591175

[2] Mark Marron. 2018. Log++ logging for a cloud-native world. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2018, Boston, MA, USA, November 6, 2018*, Tim Felgentreff (Ed.). ACM, 25–36. https://doi.org/10.1145/3276945.3276952

[3] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 102–112. https://doi.org/10.1109/ICSE.2012.6227202

[4] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving Software Diagnosability via Log Enhancement. *ACM Trans. Comput. Syst.* 30, 1 (2012), 4:1–4:28. https://doi.org/10.1145/2110356.2110360