DOI 10.31891/2307-5732-2022-315-6-98-101
УДК 004.052

КАПІТАНЕЦЬ Степан
Хмельницький національний університет
e-mail: stepan.kapitanets@gmail.com
РАДЕЛЬЧУК Галина
Хмельницький національний університет
https://orcid.org/0000-0002-9728-4390
e-mail: gal_2015@ukr.net

ОСОБЛИВОСТІ ЛОГУВАННЯ ДАНИХ ТА ВПЛИВ ТИПУ ДОДАТКА НА ВИБІР МЕТОДОЛОГІЇ ЛОГУВАННЯ

Для підвищення надійності програмних систем та своєчасного виявлення проблем використовується журналювання (логування) даних. Вивчаючи журнали додатків, розробники програмного забезпечення можуть виявляти аномалії та помилки у роботі програмних систем, знайти причину і надати оновлення чи патч із виправленнями. Тим не менше, не дивлячись на важливість практики логування даних, у більшій частині випадків цей процес відбувається хаотично. Розробники пишуть код журналювання виключно на основі своїх власних міркувань або ж взагалі нехтують ним.

У статті описано дослідження, яке було зосереджене на стратегіях логування, що використовуються на реальних проектах різних типів. Було проведено дослідження методів логування десктопних та веб-додатків, а також запропоновано покращення для методології створення журнальованих даних, які роблять її більш гнучкою та зручною для команд розробників програмного забезпечення, незалежно від типів розроблюваних додатків.

Ключові слова: журналювання даних, логування, виявлення аномалій, виявлення помилок, методологія журналювання.

KAPITANETS Stepan, RADELCHUK Galina
Khmelnytsky National University, Ukraine

PECULIARITIES OF DATA LOGGING AND INFLUENCE OF APPLICATION TYPE ON THE CHOICE OF LOGGING METHODOLOGY

Software is not perfect and may contain bugs that cannot be detected by tests, or that will manifest themselves only over time. Sometimes problems can occur even in the code of third-party libraries or services that are used in the software system. To improve the reliability of software systems and timely detection of problems, data logging is used. Logging is a very important aspect of software development. By studying application logs, software developers can detect anomalies and errors in software systems, find the cause and provide an update or patch with fixes. Nevertheless, despite the importance of data logging practice, in most cases this process is chaotic. Developers write logging code solely based on their own considerations or neglect it altogether.

This study points to the need to improve the state of research on data logging practices. For example, more efforts should be made to define what exactly to log, to generate logging data in a clear way and from the developer's point of view with enough information to recreate the events even without having to run the application. This article describes ways in which this can be achieved.

This research aims to establish a comprehensive understanding of the state of the art in data logging research with an emphasis on identifying possible problems and gaps that will further shed light on potential future research directions. This paper describes a study that focused on logging strategies used on real projects of different types. It investigated logging methods for desktop and web applications, and proposed improvements to the logging methodology that make it more flexible and convenient for software development teams regardless of the types of applications being developed.

Keywords: data logging, logging, anomaly detection, error detection, logging methodology.

Вступ. Постановка проблеми

Безперервність у роботі стала невід'ємною вимогою для значної кількості сучасних програмних систем (ПС). З цієї причини постійний моніторинг програмного забезпечення (ПЗ) та своєчасне усунення аномалій стало необхідністю. Проте, сучасні ПС стають все більш розподіленими, що значно ускладнює процес виявлення аномалій та причин їх виникнення, особливо, якщо йдеться про складні ПС. Тому інформація про поведінку системи під час її виконання є критично важливою для своєчасного виявлення та усунення помилок. Як правило, дані, що містяться у журналах, є чи не єдиним джерелом інформації, яким можуть користуватися програмісти у процесі усунення неполадок та діагностики збоїв на вже запущених ПС. Щоб мати можливість знайти першопричину такої небажаної поведінки, застосунок зазвичай фіксує етапи свого виконання та змінні і записує їх у деяке попередньо визначене місце призначення. Застосункам часто потрібно реєструвати різні повідомлення, що описують поточний статус або сповіщають про певні події, та записувати їх у файл журналу. Однак, на практиці журнали часто можуть бути відсутніми або у них опускаються критичні деталі або записані повідомлення не мають стандартної форми.

Оскільки основним призначенням логів від самого початку було саме налагодження [1], вони можуть містити велику кількість даних. Основною метою ведення журналу є можливість проаналізувати вхідні дані та виклики, що призводять до збоїв у роботі додатку. Таким чином, однією з головних особливостей логів є те, що при неправильному їх формуванні обсяг журнальованих даних може стати дуже великим за дуже короткий проміжок часу. Інший важливий аспект, який слід враховувати на етапі розробки ПЗ, полягає в тому, що навіть сам процес ведення журналу може зайняти деякий час. А це ще раз підкреслює, що факт зловживання журналюванням повідомлень може принести більше шкоди, ніж користі.

Добре організований механізм логування разом з дотриманням основних правил їх оформлення та інструментом для аналізу аномалій не тільки не нашкодить продуктивності системи, а й надасть додаткові

бонуси у вигляді можливості моніторингу системи у реальному часі та стабільнішої роботи застосунку за рахунок своєчасного виявлення проблем.

Отже, **метою** статті є встановлення комплексного розуміння стану дослідження практики журналювання даних з акцентом на виявлення можливих проблем та прогалин, а також покращення для методології створення журнальованих даних, які роблять її більш гнучкою та зручною для команд розробників ПЗ, незалежно від типів розроблюваних додатків.

Виклад основного матеріалу

Особливості журналювання даних

Запис журналу призначений для того, щоб допомогти проаналізувати збої у поведінці додатку та виявити фрагмент серцевинного коду програми, який призводить до небажаної поведінки. Лог-файли, які першочергово призначені для налагодження [1], можуть допомогти дослідити, що і як сталося навіть без повторного запуску програми, якщо ці журнали містять всю необхідну інформацію. Однак, це не завжди так. На практиці журнали часто відсутні або опускають критичні деталі [2] або у багатьох випадках розробники додатків ведуть недостатню кількість логів [3]. Тому ведення журналів є дуже важливим при проектуванні ПС.

При включенні протоколювання у додаток важливо визначитися з тим, ЩО реєструвати і ЯК це робити. Кожен запис журналу має, якщо це можливо, реєструвати, що сталося, коли це сталося, хто спровокував подію і чому саме вона трапилася [4]. Звідси випливає, що запис у журналі має, по можливості, завжди відповідати на наступні питання.

1. Що сталося?
2. Коли це сталося?
3. Де це сталося?
4. Хто спровокував подію?
5. На кого вплинула ця подія?
6. Чому це сталося?
7. Що спричинило це?

У відповіді на перше запитання слід вказати, який тип події стався (попередження, помилка або інформація) і на що вплинула ця подія (система, компонент, ресурс даних, обліковий запис користувача). Також слід вказати стан події (наприклад, якщо деякий процес не вдалося завершити).

На друге питання відповідає часова мітка. Для розподілених додатків важливо включати також інформацію про часовий пояс, оскільки сервери можуть знаходитися у різних частинах світу, і фактичний час може бути змінений (наприклад, у випадку коли сервер не перемикається між зимовим та літнім часом).

Відповідь на третє питання має розповісти про те, яка система (додаток, клас) були задіяні, і частково може вказувати на причину виникнення проблеми (наприклад, у випадку коли база даних була недоступна, що й спричинило збій у роботі програми).

На четверте питання відповідає ідентифікатор того, хто спричинив проблему, тобто чиї дії призводять до її виникнення (наприклад, можна вказати ім’я користувача).

П'яте питання дозволяє дізнатися, чи вплинула подія, що сталася, лише на одного користувача (який її спричинив) чи вона зачепила більше кінцевих користувачів.

Шосте питання говорить про причину виникнення проблеми (наприклад, неправильний пароль, не встановлені налаштування, не заданий обов’язковий параметр тощо).

Сьоме питання говорить про те, який тип події стався, тобто яка дія призвела до цього (вхід у систему, запуск деякої операції тощо).

На перший погляд може здатися, що у правильній організації протоколювання немає нічого складного. Однак, стратегія логування залежить від середовища застосування та кінцевого користувача, на якого розрахована ПС.

Реєстрація всіх відповідей на набір запитань може означати велику кількість записів у журналі (а розмір файлу журналу може вплинути на роботу програми). Якщо реєструється занадто великий обсяг даних, то з часом пам’ять може бути переповнена даними журналу. Також важливо заздалегідь визначити, протягом якого періоду часу файли журналів будуть зберігатися, а потім видалятися. Без видалення старих лог-файлів сховище буде переповнюватися історичними даними, які вже не є актуальними, і призвести до проблем з пам’яттю для програми. Аналогічне очищення слід проводити для автономних додатків при вході у систему.

Записи із журналу представляють собою дані, що дозволяють їх читачу витягати інформацію про те, що не так, і перетворити її на знання про те, як цьому запобігти та як це виправити. Важливо заздалегідь вирішити, які дані можуть допомогти читачеві журналу, щоб включити їх у запис. Записуючи у журнал все, ми зберігатимемо там дуже велику кількість надлишкової інформації, що призведе до марного використання пам’яті та і, врешті-решт, до її переповнення.

Інший аспект полягає у тому, що логування також займає певний час [4] і тому важливо вирішити, коли його краще проводити: метод або дію розпочинати до журналювання, паралельно з ним чи після нього. Логування перед завершенням обробки запиту може призвести до затримок, а його виконання перед ним не включатиме результатів проведеної роботи. Існує також інформація, яка ніколи не повинна включатися до лог-файлів (наприклад, паролі або інші конфіденційні дані користувачів).

Методології журналювання даних на основі типу розроблюваного додатку

Проведене дослідження складалося з огляду літератури та досліджень тематичних проектів з розробки ПЗ. У вибірку було включено ПЗ з відкритим кодом, розроблене з використанням різних мов програмування (зокрема, Java та C#). Проекти додатків на Java представляли собою веб-додатки, а проекти

ПС на C# були десктопними застосунками. Це є дуже важливим фактом, який пояснює велику кількість відмінностей у результатах проведеного дослідження.

У проектах на Java використовувалися Log for J (Log4J) від Apache [5], Simple Logging Facade for Java (SLF4J) [6] та LogBack [7]. Жоден з досліджуваних проектів не використовував нативний вбудований пакет Java.Util.Logging [8]. Розробники ПЗ переходили з Log4J на SLF4J та на фреймворк LogBack через накладні витрати та обмеження Log4J [3]. Усі наймолодші проекти використовували LogBack. Причиною цього слугують менші накладні витрати та більші можливості розробників.

Ведення журналу здійснювалося безпосередньо шляхом виклику відповідного методу на екземплярі логера, а також через аспекти до та після виконання методу. Другий підхід використовувався дуже рідко.

Розробники реєстрували ім'я користувача, який увійшов у систему один раз (в момент запуску програми або після його успішної авторизації у додатку), мітку часу, ім'я java-класу, який виконував операцію та трасування стеку в разі виникнення виключення (у блоках try catch). Повідомлення, які зберігалися у журналі шляхом ручного виклику логера, містили переважно етапи чистої бізнес-логіки (наприклад, користувач авторизований, права доступні, дію виконано тощо).

Журнали містили велику кількість інформації, яка реєструвалася самим застосунком за допомогою фреймворку Spring, а не розробниками. Це призводило до того, що файли ставали надзвичайно великими вже протягом декількох днів. Файли журналів додавалися кожного разу, коли починалася нова транзакція. Максимальний розмір файлу був заданий у налаштуваннях XML (Extensible Markup Language) і коли він досягався, фреймворк логування починав записувати дані у новий файл без втручання розробника ПЗ. Це налаштування дозволило зменшити кількість лог-файлів.

Однак, кількість зареєстрованих даних все одно була дуже великою і це спричиняло проблеми з доступним обсягом пам'яті. Старі журнали видалялися вручну з періодичністю у декілька місяців або у момент, коли служба моніторингу сповіщала про низький обсяг доступного сховища на сервері.

Під час супроводу розроблених додатків журнали були проаналізовані розробниками ПЗ та адміністраторами середовища для аналізу першопричин виявлених проблем. За результатами аналізу було виявлено, що знайти причини небажаної поведінки додатків вдавалося не завжди через запізнілі повідомлення (коли журнали вже були видалені) або через те, що записи у журналах не дозволяли змоделювати етап роботи програми, який призвів до її неправильної поведінки.

Розробники в основному протоколювали прямі повідомлення або виключення, але вхідні параметри, які могли призвести до таких подій, не були записані у журнал, а записи журналів, включені у ці файли, були лише частково корисними для аналізу першопричини збою.

Файли журналів допомагали лише в обмеженій кількості випадків (в основному, у той час, коли облікові дані для входу були неправильними або коли база даних була недоступною).

У середовищі .NET також існує декілька фреймворків логування [9]. До найвідоміших з них на сьогоднішній день відносяться Log for .NET (Log4Net) від Apache [10, 11] та .NET Log (NLog) [12], який набуває все більшої популярності. Ці фреймворки подібні до фреймворків, відомих у мові Java; їх потрібно викликати безпосередньо. Подібно до аспектного програмування в Spring та ведення журналу до та після фактичного виконання методу, що забезпечується анотацією, в C# даний функціонал пропонується бібліотекою PostSharp [13], яка дозволяє також протоколювати як вхідні так і вихідні параметри методу.

Незважаючи на всі ці можливості, розробники ПЗ, як правило, використовували свої власні імплементації логувальників. У всіх восьми досліджуваних проектах розробники віддавали перевагу своїй власній реалізації. Логувальник, як правило, моделювався як окремий, синглтонний клас [14], який включається у проект та викликається вручну.

Таблиця 1

Порівняння використовуваних підходів до логування у Java та C#

Питання для відповіді при логуванні	Підхід, що використовувався у Java-проектах	Підхід, що використовувався у C#-проєктах
Що сталося?	Трасування стека виключення в журналі. Прямі повідомлення в лог, що інформують про те, що метод був успішно завершений.	Трасування стеку зареєстрованого виключення. Прямі повідомлення журналу, що інформують про те, який метод був виконаний і на якому етапі.
Коли це сталося?	Мітка часу сервера	Мітка часу операційної системи комп'ютера
Де це сталося?	Ім'я Java-класу, метод, що викликається	Ім'я методу виклику, виключення
Хто спровокував подію?	Ім'я користувача	Ім'я користувача
На кого вплинула ця подія?	Ім'я користувача	Компонент запуску та ім'я користувача
Чому це сталося?	Трасування стека виключення в журналі. Прямі повідомлення в лог, що інформують про те, що метод був успішно завершений.	Трасування стеку зареєстрованого виключення. Прямі повідомлення журналу, що інформують про те, який метод був виконаний і на якому етапі.
Що спричинило це?	Трасування стека з винятком	Трасування стеку зареєстрованого виключення

Власноруч реалізований логувальник розпоряджався налаштуваннями, заданими у власному класі, і ці налаштування можна було встановити заново у додатку. Таким чином, логувальник працював автоматично після включення у проект додатку. Використання інших фреймворків (на кшталт Log4Net чи NLog) потребувало б наявності XML-файлу налаштувань.

Відповіді на сім запитань, описаних вище, спираючись на досліджувані проекти, подані у таблиці 1.

Як бачимо, команди розробників Java та C# обрали абсолютно різні стратегії ведення журналів. Команди C#-розробників використовують перевагу знань про кінцевого користувача (оскільки в основному досліджувалися саме .NET-проекти, орієнтовані на десктопні застосунки), що допомогло їм розробити власний логувальник та власний процес логування. Java-розробники для ведення журналів використовували фреймворки та можливості, які надає фреймворк Spring (включаючи аспектне логування). Проте, журнали були величезними у порівнянні з журналами C#; записи журналу не завжди були корисними для аналізу першопричини і приводили більше до переповнення пам'яті, ніж до документації помилки. Це говорить про існування значних проблем із високонавантаженими серверними додатками, які існують через неможливість застосувань деяких оптимізацій, які доступні для десктопних застосунків. До цих проблем належить висока складність виявлення аномалій в ручному режимі через дуже велику кількість даних, яка генерується програмною системою за короткі проміжки часу, та високе навантаження на пам'ять. Це свідчить про необхідність розробки нового методу, який дозволить слідкувати за аномаліями у ПС з великим обсягом даних.

Висновки

Стаття присвячена темі логування даних та досліджувала підходи, які використовуються командами розробників ПЗ. Розглянуті основні аспекти ведення логів та визначено сім основних питань, на які важливо отримати відповіді при журналюванні даних. Також були визначені та обговорені питання, пов'язані з обсягом записів у лог-файлах, проаналізовані записи журналів та здійснено їх порівняння з попередньо визначеними питаннями, щоб показати, де записи відповідають цим питанням, а також щоб визначити можливості для вдосконалення.

Результати дослідження показують, що важливо думати про логування ще під час розробки програми та продумати відповіді на вищезгадані питання. Не варто забувати, що зберігання усієї використовуваної у додатку інформації не є найкращим способом логування. Для цього необхідно проаналізувати важливість тих чи інших даних для виявлення аномалій у роботі системи. Перегляд записів журналу з точки зору розробника може значно поліпшити записи журналу і скоротити час, необхідний для виявлення та аналізу першопричин виникнення неполадок у системі.

Також сформульовано пропозиції, які у процесі дослідження були визначені, як важливі, і які можуть бути використані для удосконалення методології логування, що використовується командами розробників на практиці при розробці високонавантажених ПС.

References

1. Chuvakin A. & Peterson G. How to Do Application Logging Right. IEEE Security & Privacy, 2010. Vol. 8, no. 4, July/Aug. P. 82-85.

2. Marty R. Cloud application logging for forensics. Proceedings of the 2011 ACM Symposium on Applied Computing (SAC). March 21-24, 2011, TaiChung, Taiwan. P. 178-184.

3. Idan H. The complete guide to Java logging in production. OverOps. URL: https://land.overops.com/java-logging-in-production-ebook/

4. Xu W., Huang L., Fox A., Patterson D., Jordan M. I. Detecting large-scale system problems by mining console logs. SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. October, 2009. Pages 117–132. https://doi.org/10.1145/1629575.1629587

5. Apache Log4j 2. Logging Servises. URL: http://logging.apache.org/log4j/2.x/

6. Simple Logging Facade for Java (SLF4J). SLF4J. URL: http://www.slf4j.org/

7. Logback Project. LOGBACK. URL: http://logback.qos.ch/

8. Package java.util.logging. Java™ Platform, Standard Edition 8 API Specification. URL: https://docs.oracle.com/javase/7/docs/api/java/util/logging/packagesummary.html

9. Java Logging Tools and Libraries. Guide and directory of Java logging tools, libraries and articles. URL: http://www.java-logging.com/

10. .NET Logging Tools and Libraries. The definitive directory and guide to .NET logging tools, frameworks and articles. URL: http://www.dotnetlogging.com/

11. The Apache log4net project. Logging Servises. URL: https://logging.apache.org/log4net/

12. Flexible & free open-source logging for .NET. NLog. URL: https://nlog-project.org/

13. PostSharp Diagnostics: Logging and Tracing. #POSTSHARP. URL: https://www.postsharp.net/¬diagnostics/net-logging

14. Clarke S., Harrison W., Ossher H., Tarr P. Subject-oriented design: towards improved alignment of requirements, design, and code. ACM SIGPLAN Notices. Oct. 1999. Volume 34. Issue 10. P. 325-339. https://doi.org/10.1145/320385.320420