

MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Software Engineering

Privacy-aware Crash Monitoring Platform for Android Applications

By: Florian Mötz, BSc

Student Number: 1810299012

Supervisors: Mag. Oliver Wana, MSc
Alexander Grafl, MMSc

Vienna, November 30, 2020



Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Vienna, November 30, 2020

Signature

Kurzfassung

In modernen Applikationen sind Fehler und Abstürze oft unausweichlich, da nicht alle Bugs während des Testens unterschiedlicher Szenarien gefunden werden können. Besonders in heterogenen Systemen wie Android Smartphones, auf welchen die Applikationen auf verschiedenen Betriebssystem-Versionen installiert sind, Systembibliotheken von diversen Herstellern implementiert wurden und in verschiedenen CPU Architekturen mit verschiedenen Hardware-spezifikationen laufen, ist das Testen aller möglichen Kombinationen nahezu unmöglich.

Um als Entwicklerin und Entwickler trotzdem die Möglichkeit zu bekommen, diese Fehler zu beheben, ist es unausweichlich, Informationen über aufgetretene Fehler und Abstürze sowie Details über das Smartphone, auf dem der Fehler aufgetreten ist, zu bekommen.

Da die Applikation auf einem komplett fremden System läuft, ist es nicht möglich, Zugang zu den Logs des Gerätes zu bekommen. Deswegen müssen mobile Applikationen eigene Lösungen implementieren, um Fehler- und Absturzdaten zu sammeln und diese an einen Server senden, um es der Entwicklerin oder dem Entwickler zu ermöglichen, diese Daten zu analysieren.

Es existiert bereits eine Reihe solcher Systeme zum Einbinden in Applikationen, jedoch bringen diese oft Nachteile mit sich, wie zum Beispiel proprietäre, nicht quell-offene Datenspeicherungssysteme, oder Restriktionen über welchen Verbreitungsweg eine Applikation installiert worden sein muss, um die Daten sammeln zu können.

Das Ziel dieser Arbeit ist es, Anforderungen an ein datenschutzfreundliches, einfach zu benutzendes Error-Reporting System zu finden und bestehende Lösungen anhand dieser Anforderungen zu evaluieren. Da keines der bestehenden Systeme alle Anforderungen erfüllen konnte, wird ein neues, kostenloses, selbst gehostetes System zur Analyse von Fehlern und Applikationsabstürzen in Android Applikationen vorgestellt.

Schlagworte: Android, Kotlin, Monitoring

Abstract

In modern applications, errors and crashes are often inevitable, as often not all bugs can be found while testing various different scenarios. Testing the application on all possible environments with all environmental contingencies is especially hard on Android, where the application runs on different operating systems, different implementations of the platform tools by different vendors as well as different CPU architectures with different specifications.

To be able to fix the errors and crashes that occur on user's devices, it is crucial to get details about the error and details about the smartphone environment the application is running on. Due to the application running on a user's device, there is also no default way to gain access to the device logs. Therefore, applications need to implement custom solutions to gather error and crash data, and to send this data to a server, where the developers can analyze the issue.

Such crash reporting systems are already available and widely used, but often come with various drawbacks, such as proprietary and closed source back-ends where the data is stored, or restrictions to the distribution channel of the application, or complicated to be set up.

The aim of this thesis is to evaluate requirements for a privacy aware, easy to use crash reporting system, and to evaluate existing solutions in regards to those requirements. As none of the analyzed solutions met all requirements, a new, free, selfhosted solution to report crashes in Android application is going to be introduced.

Keywords: Android, Kotlin, Monitoring

Contents

1	Introduction	1
1.1	Motivation	3
1.2	Crash Reporting System	4
2	State of the art	7
2.1	Selection of existing systems	7
2.2	Analysis of existing systems	8
2.2.1	Google Firebase	9
2.2.2	Android Vitals	13
2.2.3	ACRA	15
2.2.4	Sentry	17
2.2.5	Summary	19
3	Methodology	20
3.1	Source of requirements	20
3.2	Comparison of tools	21
3.3	Approach for a new system	22
3.4	Evaluation of results	22
4	Requirements to a privacy aware crash reporting system	23
4.1	Free to use	23
4.2	Open Source	23
4.3	Possibility to self-host the back-end service	24
4.4	Easy to set up	25
4.5	Configurable	25
4.6	Non-restrictive	26
5	Suggested solution	27
5.1	Development of the new system	27
5.2	Architectural overview of the system	28
5.3	Architectural decisions in regards to the requirements	28
5.4	Components of the system	30
5.4.1	Android library	30
5.4.2	Back-end system	38

6 Reflection	44
6.1 Applicable, existing systems	44
6.2 Applicability of the developed system	44
7 Future work	46
Appendices	47
A Statistics	47
B Koarl Configuration & Payload	48
Bibliography	52
List of Figures	59
List of Tables	60
List of Listings	61
List of Abbreviations	62

1 Introduction

In modern applications, errors are often inevitable. Especially in larger programs, the number of errors scales proportionate with the number of lines the code consists of, where as an industry standard it is assumed that (depending on the given study) a program will contain 1–25 errors per 1000 lines of code [63]. While errors can be dealt with silently in applications, uncaught errors can lead to crashes in mobile applications, which shuts down the application and besides user dissatisfaction can lead to data loss. Large testing companies even state, that in practice, due to the heterogeneity of mobile applications “it will be impossible to achieve ZERO crash rate” [74]. Mobile applications still gain in importance and popularity, as seen in the increasing number of app downloads in figure 11 in appendix A, where iOS and Android are the two most used operating systems as seen in figure 12 in appendix A.

While Apple produces and manufactures the devices as well as the operating system iOS and its tools, the Android operating system is available as an open operating system that manufacturers can use, adapt and ship with their mobile devices. The fact that Android devices and their firmware are built by multiple different vendors, which may customize the framework itself, or do not exactly implement the Android Compatibility Definition [40], errors or crashes may only occur on single devices due to different environmental factors or different implementations of some Application Programming Interfaces (APIs). This is not a new trend, as a study in 2013 already showed the trend that “mobile platforms are moving toward fragmentation rather than unification” [55] in terms of different platforms but also in terms of fragmentation within a platform, with different hardware specifications, screen sizes and resolutions, and different APIs on different versions of the operating systems.

Furthermore, due to the fact that the Android operating system is often adapted by manufacturers, devices often do not receive major operating system updates, as the operating system update would have to be adopted for the given device again. This leads to the fact that more different active operating system versions are actively being used (i.e. compared to iOS), as seen in figure 13 and figure 14 in appendix A, and therefore more operating system versions need to be supported by Android applications. Running an application on different operating system versions introduces more complexity and different behavior on the different versions for the developer to consider, which results in more complex code and therefore a higher error probability. Furthermore, as the devices are manufactured by different vendors which apply different adaptations on the Android operating system [27], the API provided by the system may behave differently on devices of those different vendors [48], which may introduce errors which only occur on those devices. Due to this heterogeneous device architecture, it is nearly impossible to test an application on all possible devices, and larger projects cannot be tested with all

possible test cases, data and environments in mind without proper testing tooling, which often costs effort, resources, money and time [28]. Therefore, crashes can occur on a user's device after roll-out, and as developer of an application, it is essential to get feedback of errors and crashes including logs of the crashes which happened on the user's device, to be able to fix the underlying bug of the crash in a new version of the application.

Besides the heterogeneity of the devices and the complexity of the applications, there are other trends which require developers to gain access to errors and crashes in the application after roll-out. Companies like Facebook, Google or Amazon have introduced a software development paradigm which can be described as "Move fast and break things" [86], in which products and features should be developed and shipped as fast as possible (without testing them completely), with the mindset that if something gets broken, it can be fixed later on in an upcoming release within a relatively small time frame (compared to development and release cycles used before). Using that paradigm, it is even more critical to automatically get errors and crashes from the users devices, to be able to know where crashes and errors occurred and to fix them in the next iteration.

Furthermore, in contrast to i.e. back-end applications, the application is run on a device owned and administered by the user, which means that there is no direct access to the device possible. If errors in a back-end application occur, application logs can usually be read easily, as the application runs on a server which is in control of the application developer, or the application developer has access to. With the application running on a user-owned device, there is no direct access to the device or application logs possible. Furthermore, as Android is an open operating system that can be used by anyone without a centralized architecture, by default the Android operating system does not provide a possibility to collect, analyze or gain access to device or application logs on end-user devices. Therefore there is a need for developers to include mechanisms or tools into their applications, which will collect data on errors and crashes as well as the respective logs to them, and share that data and logs in some way to the application developer.

While for analyzing the error condition it is desirable to collect as many logs, environmental data as well as user data which led to the error condition as possible, this data may contain personally identifiable information as well as data which could identify the user as well as their behavior. Sharing such data may not be in the privacy interests of the application user and may be subject of requiring user consent for sharing this data. Application logs, as well as stack-traces and device information, may reveal information about the user, the application usage of an user as well as other possibly private information, which can be derived directly or indirectly from the collected data of the application. Furthermore, if large platforms collect crash data from multiple applications (i.e. by providing a crash reporting system), this platform could accumulate the data from different applications and therefore gather even more information about a user, often without the knowledge or consent of the user. Consequently, there is a discord between gaining as much information about errors and crashes easily from application crashes on user devices and the users privacy interests of not sharing personally identifiable

data and not contributing their private data into larger data accumulation companies. This thesis therefore does not only analyze technical details or system architectures of a crash reporting system, but also analyses whether existing solutions can be seen as privacy-aware in terms of handling the data collected from the user devices, and whether data accumulation over multiple applications – which the user is not informed about – is possible or likely. In honor of those thoughts, the hypothesis of this thesis has been set as:

How can a privacy-aware, easy-to-use crash reporting system for Android applications be developed?

Moreover, connected to that hypothesis, the following two subsequent hypothesis will be handled in this thesis:

What are requirements for a privacy-aware crash reporting system?

Are there already privacy-aware crash reporting systems available?

1.1 Motivation

For developers, it is always useful, if not crucial, to get insights about their applications stability, also after rolling out the application to their users. While when testing the application locally developers usually can see logs, error messages as well as stack-traces, it is quite hard to get them from an application which is installed on a users device. And while developers often test the main use-cases of an application or a feature they are currently developing, in real world situations (when the application is used by the users), rare edge-cases as well as unexpected data can lead to crashes of the application. Those logs and error messages are, however, crucial to be able to get insights on the behavior of the application as well as the user in how it led to the crash or error, to be able to make the application resilient to handle the case gracefully. Furthermore, as the test-data most often cannot reflect all possible variations of production data, it is also admirable to get insights into which data of the user led to the undesired behavior of the application [28].

While aiming to be able to get as much details as possible about the steps the user performed, the environment the application is running in (e.g. the device model or Android version) and the real data of the user which with which the situation should be reproducible, this data can be of sensitive nature, which is subject to not be shared in an insecure way and to anyone. Furthermore, if for every small issue and error case this data would be logged, the aggregation of that data would make it possible to get much more insights into the behavior of the user in a quite larger scope than needed to be able to get insights into the particular case.

While large companies often provide services free of charge, they often collect, store and often also share the data commercially with other providers, in order to be able to provide the service for free. As the most used crash reporting systems are free (and closed source), there

is no possibility to verify whether the company is handling user data in a privacy aware and non-accumulative way. With various scandals over the last years (such as the “*Cambridge Analytica*” case), more and more people as well as companies understand, that services or products free of charge often trade personal data of the user [50]. While personal data can be traded by companies for profit, systems and companies often only collect the data for possible later use, and may not have a profitable or malicious intention planned with the data. However, due to the fact that those systems collect the data, and the data is stored at the company, data leaks as well as targeted cyber attacks may disclose the data to the public or to be sold.

As people as well as companies and developers tend to get more aware of privacy and data protection, principles like the *privacy by design* have been formulated and are implemented by more and more systems [17]. A guideline has been published by the Information & Privacy Commissioner of Ontario in 2010, which introduced 7 foundational principles for systems to implement and to respect, so that an application honors a users privacy as a default and by design and not as an *opt-in* [1]. The 7 principles are listed in table 1.

1	Proactive not Reactive; Preventative not Remedial
2	Privacy as the Default Setting
3	Privacy Embedded into Design
4	Full Functionality - Positive-Sum , not Zero-Sum
5	End-to-End Security - Full Lifecycle Protection
6	Visibility and Transparency - Keep it Open
7	Respect for User Privacy - Keep it User-Centric

Table 1: The 7 Foundational Principles of Privacy by Design [1]

1.2 Crash Reporting System

As stated before, it is necessary for developers to get access to their applications logs, especially in the case of errors or crashes, as bugs can usually be found faster and more efficient with a stack trace available [75]. In back-end applications or websites, where the application runs on a server on which the application developer has most often access to the logs of the application, it is easy for developers to extract the logs generated by the application to be able to analyze errors and crashes. Furthermore, the hardware as well as the application environment (in terms of e.g. operating system, installed and running programs) are often known to the developer, or easy to obtain. In the mobile application sector, where the application is running on the physical devices of the users, to which the developer (or developing company in general) does not have access to in any way, it is not trivial to get access to the application logs. Furthermore, as Android is an open operating system, which many device manufacturers use as their operating

system, the hardware as well as the operating system environment in which the application is running in is quite heterogeneous and fragmented [95, 45]. Also, as in the Android ecosystem, operating system updates have been a tedious task for manufacturers [61], they often have not provided operating system updates for older device models [47], which led to the developers supporting a lot of different operating system versions [42]. While the official documentation of Android suggest to support “about 90% of the active devices” [42], this indicates that according to the official distribution chart from Google, shown in figure 2, which contains data from May 2019, all Android versions from *API Level 21* and newer should be supported, which sum up to 89.3%. This is a total of 10 different operating versions with different behavior, if the newest two versions (*Android 10 and 11*, which have not been available at the time of the statistic) are also taken into account. While the framework and APIs change with newer versions of Android (new APIs are added, existing APIs change their behavior, support for some functionalities may be dropped), applications may misbehave, crash or error on specific Android versions only.

Android Version	API Level	Distribution
2.3.3 - 2.3.7	10	0.3%
4.0.3 - 4.0.4	15	0.3%
4.1.x	16	1.2%
4.2.x	17	1.5%
4.3	18	0.5%
4.4	19	6.9%
5.0	21	3.0%
5.1	22	11.5%
6.0	23	16.9%
7.0	24	11.4%
7.1	25	7.8%
8.0	26	12.9%
8.1	27	15.4%
9	28	10.4%

Table 2: Relative number of devices running a given version of the Android platform. Data collected during a 7-day period ending on May 7, 2019 by Google [41].
Any versions with less than 0.1% distribution are not shown.

Therefore, not only is there no general access for application developers to application or error logs, but information about the environment in which the application is running, especially when errors or crashes occur, is not only attractive, but often also crucial to be able to reproduce the error or get insights in why the application behaved in a way that a given error case has been triggered. This led to application developers to implement custom error logging and crash

handlers into their applications, in order to catch logs, errors and crashes, and to send them to either their own or a third party back-end server, to which the developers have access and are able to inspect the data. Such systems should not only display the logs of the error or crash, but also provide additional information about the device and the state of the running application. Such information may contain additional data to the crash / error, such as a stack-trace, other threads currently running or user specific data (e.g. user-id) to be able to check on which (user-) data the error occurred, or the state of the device environment at the time of crash (e.g. memory left in the device, screen-orientation of the device). Furthermore, static information about the device, such as the device model, manufacturer or version of the operating system, may also be crucial to analyze for the error report. While this is a mechanism or system that should be easy to set-up and implement and at best not create any additional cost or maintenance effort, products like the *Android Vitals* from the Google Play Store or *Crashlytics* from Firebase have found affection in the Android developer community, as they provide such a service for free and with just a few steps of set-up. As those platforms provide analytical data as well, the crash data is often connected to some user-data or data not necessarily needed to reproduce the crash scenario as well.

2 State of the art

As stated before, there are already well established and widely used systems available. This chapter will analyze which systems are already available to be used for analyzing crashes and errors in Android applications.

2.1 Selection of existing systems

The search for already existing systems has been performed exploratory by using search engines with different, matching keywords for finding such systems. This method has been chosen as for well established or good working solutions in the Android development community either blog posts or other resources with the given keywords are being published. Additionally, tools that are in use by my employer *Tailored Apps* as well as tools that have been evaluated sometime in the past by my employer have been chosen. The systems found have been pre-filtered according to the following criteria:

- The system is available online, and can be, at least for a free trial, downloaded or requested in an automatic way. This means, that there was no need to get in contact to a sales-person or manually request demo access to the system. This requirement was set in order to promote systems which seem to be available for all developers, independent of hidden fees or subscription traps for a demo or any politics (as contacting a sales person could also mean that the providing company may turn down the request due to arbitrary reasons).
- The system has at least any licensing or pricing model which does not include any fees or any payment. This requirement has been set, as additional costs for such a system for small companies as well as side-projects of developers without any (extra) budget would not be able to get access to the system at all. Furthermore, if all plans include some sort of fee or payment, the inhibition level to choose such a system may be higher.

As such a crash reporting tool often is paired with analytical features, there are also data analytic tools which offer crash reporting analysis, too. Those systems, which are often for enterprise applications targeting larger businesses, are often only possible to integrate with the combined analytical software, which means that implementing the crash-reporting part of the system forces the user to implement analytical software of a third party as well, which could possibly use the data to personalize ads or create user profiles of the application user. As in

such systems the crash reporting is only a small sub-part of an analytical software, and therefore are often only to be used when using the larger analytical software, too, such systems have not been considered. This has been done as analytical features, especially if the main purpose of a tool is analytical data collection, are contradictory with systems which implement privacy-by-design, as stated in section 1.1. If such a system would be included only for the purpose of error and crash data collection, as a side effect data collection would still be performed. Even if the data is not used or needed by the developer, the data would be collected which is not needed for the feature the system is included for, and could be used by the system for analytical purposes nevertheless by the provider of the system.

2.2 Analysis of existing systems

In this section, the different existing systems will be listed and analyzed according to the following aspects described in the following section. The selection of this criteria has been deduced from the requirements for a privacy-aware crash reporting system described in section 4 and their source described in section 3.1.

Pricing The price to use the system has been analyzed and classified. As for smaller apps or projects little budget is available, an additional crash reporting tool should not introduce another cost item for the project. Furthermore, as also often the ability to get insights into crashes is seen as a *nice-to-have* feature for developers instead of a crucial tool for quality assurance, the cost for such a system is often not seen as justified towards project managers and not planned in the project budget. Therefore, the initial cost (set-up costs excluding developer resources) as well as recurring costs such as license fees or pricing plans have been analyzed.

Self-hosted The back-end can be seen as a large data collection software, where unknown data processing with accumulated user-data is possible to be performed. Besides the fact that it is often unknown which kind of data is transferred to the back-end, it is furthermore unknown in closed source back-ends or back-ends where hosting is provided, which kind of data processing is performed. Furthermore, the geographical location of the server may be subject of different data protection laws, and therefore may be subject of different laws in terms of which data access by authorities is mandatory to be enabled, and for which purposes and to which authorities any data may be requested by authorities. Due to those facts it seems logical for privacy aware applications or developers to be able to host the back-end server of the system on-premise, as their own cloud service or on a server which is in control of the company, and not part of a third party entity.

Effort to set-up Due to the same reason stated in the pricing section, the effort to set-up an existing system may also be a crucial factor. As managers may not see the advantage to have

such a system in place, and they may tend to push the project to be finished on their planned date, an additional feature such as setting up a crash reporting tool should not introduce much effort. Therefore, another key figure in the analysis of existing systems has been the effort for developers to set the system up.

Set of features Another key factor for the analysis of already existing systems has been the set of features they provide. While many of the systems are paired with or included in analytical software, the feature-set of the systems often include a lot of analytical features, which most often would not be needed for a plain error and crash reporting tool. The focus has been set here to analyze the features directly connected to the analysis of errors and crashes as well as the environment in which they occurred. Such features may include automatic collection of crashes, the possibility to add custom data to crash reports or the ability to log a non-fatal error, which did not result in an application crash. However, due to the privacy considerations, also the possibility to disable the analytical features on both the developer side as well as the end-user side have been analyzed.

Additional Restrictions Furthermore, including the library into the project should not introduce any restrictions for the application. This includes any restriction in the usage or deployment of the application, that would not exist if the crash reporting system would not have been included. Examples for such restrictions are the way an application may be distributed and installed on a device, the preconditions of external services being installed on the device (such as the Google Play Services, which are not part of the Android Operating System but only of *Play Protect Certified Android devices* [33], and therefore not part of all devices running Android), or an increase in the minimum Software Development Kit (SDK) version, which leads to older devices no longer being able to install the application.

2.2.1 Google Firebase

Firebase is a platform with a collection of tools managed by Google, which aims to “Build apps fast, without managing infrastructure” [37]. One of the provided systems is *Crashlytics*, a tool to collect and analyze crashes and errors in Android and iOS applications. The tool was a standalone project developed by the company *Crashlytics*, until it was acquired by Twitter and later on Google to join the Firebase platform [73]. Within Firebase, Crashlytics is only a small part of the large chain of tools and services offered, which include analytic, databases, authentication services and more [37]. Figure 1 shows the dashboard of Firebase Crashlytics, where the application crashes are listed as well as the stability of the application is shown.

Pricing

Google Firebase provides two different pricing plans ('Spark Plan' and 'Blaze Plan'), which include different products of the Firebase platform and different quotas and prices thereof. While

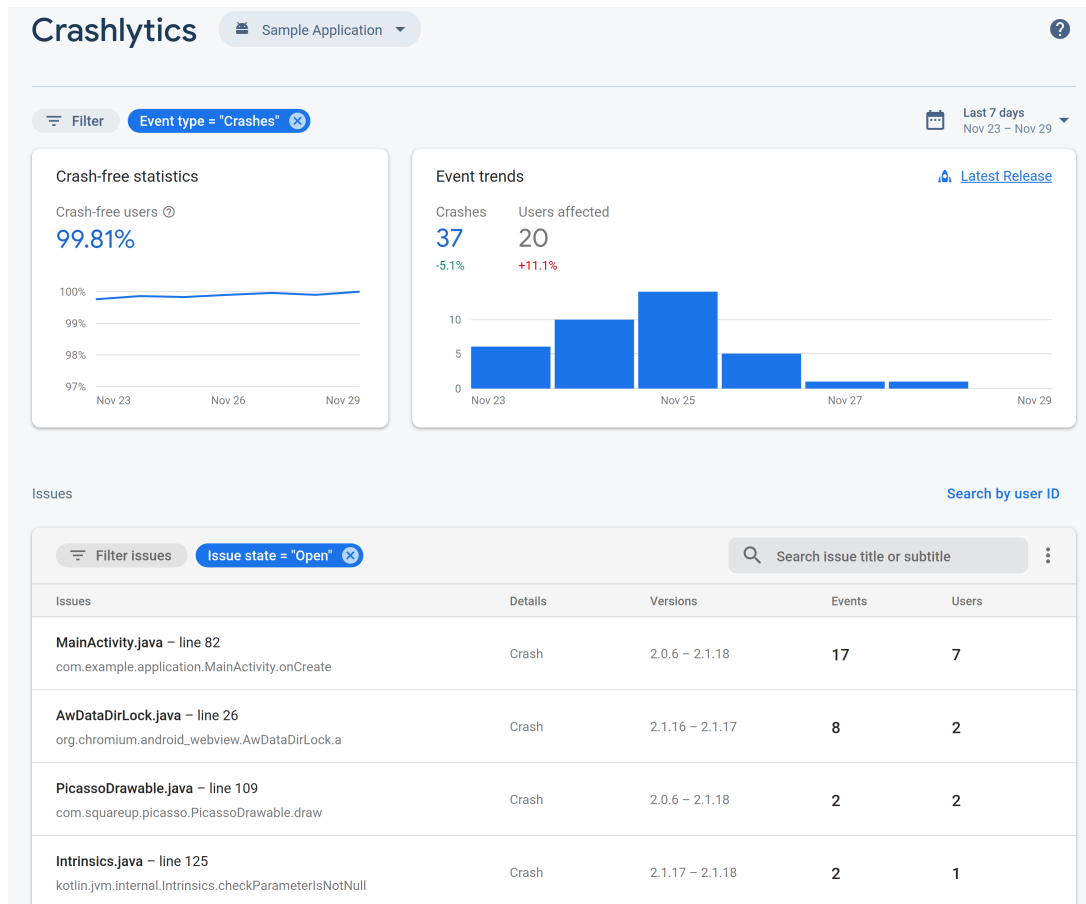


Figure 1: Dashboard of Firebase Crashlytics, which lists the top application crashes from the last seven days. Screenshot from console.firebase.google.com

the 'Spark plan' is free of charge, if only provides a limited set of features as well as a limited quota limit on different products of Firebase. The 'Blaze Plan' contains all products and quotas of the free plan, but includes more features as well as does not contain a hard-limit on quota limits, but introduces a "pay as you go" system, where exceeding requests after the free quota are billed per number of requests. Crashlytics, which is the error and crash reporting tool from Firebase, is free of charge in both plans, meaning that there are no licensing costs or fees at any point [39].

Self-Hosted

Firebase does not provide the ability to self host any of their tools, which is part of their offer. Therefore the data collection and processing is performed on Google servers in undisclosed back-end applications. According to their privacy statement, the location of those back-end services run on a global infrastructure, which may be any of the data center locations world wide run by Google [30]. In the terms and conditions is stated, that Google has the right reserved, to disclose aggregated information of all applications and data shared using Crashlytics [29].

Therefore it can be assumed, that data aggregation is performed by Google with the data shared using Crashlytics.

Effort to set-up

The setup within the mobile application is well documented and straightforward. A Google account is needed to be able to set up a new project with an application to monitor in the dashboard, as described in [35]. After the application has been set up in the dashboard, the dependency for the Crashlytics library needs to be added to the application, as well as a JavaScript Object Notation (JSON) file which can be downloaded from the dashboard, as described in [38]. This JSON file includes the information to which project in the dashboard this application belongs to and should send its data to. After that, the setup is complete after the application has been started once. There is no need to initialize the library in any way in the application code, as the initialization is performed in the background at application startup. The whole setup is well-documented, additionally a quick-guide in 4 simple steps guides through the set-up, which can be seen in figure 2.

Add Firebase to your Android app

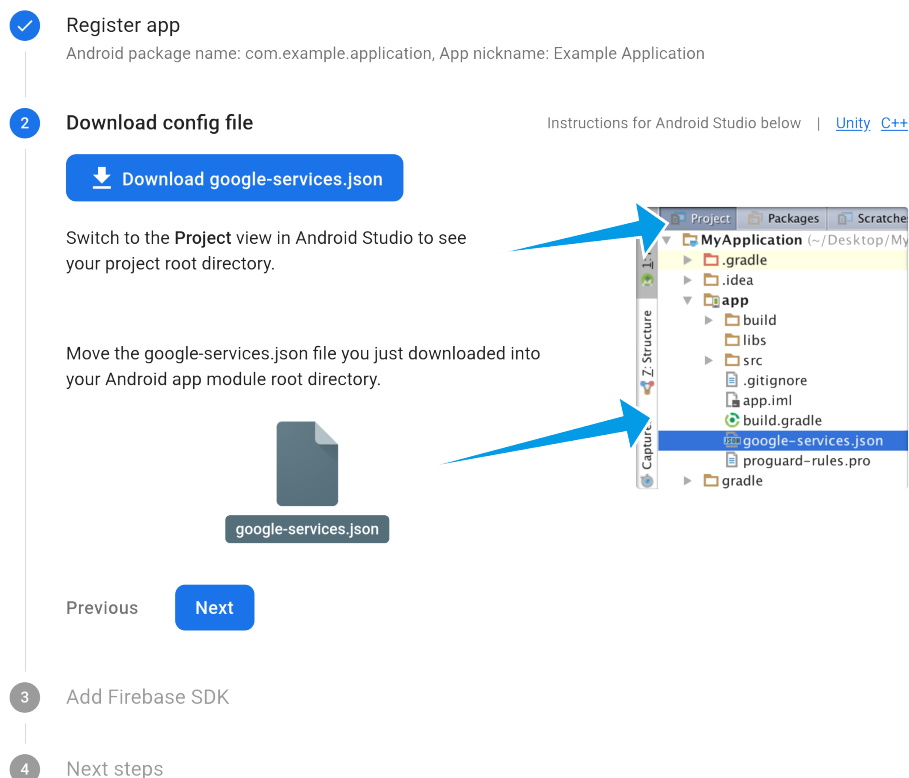


Figure 2: Set-up instructions from the Firebase console. Screenshot from console.firebase.google.com

Set of features

Firebase Crashlytics automatically detects application crashes and uploads them to a back-end services. Furthermore, the library provides the possibility to log a non-fatal crash, which is an exception that can be passed to a method, which does not need to be an uncaught exception or application crash. Additionally, Firebase Crashlytics provides additional features. Within the application, so-called custom keys can be added for a user. Those keys are then present in the dashboard at any crash this user reported. A sample use-case would be to set the user id as a custom key to Crashlytics, so that for every crash you would be able to connect the crash to the data of the user to be able to reproduce the crash with the original data. The sample with the user id is even better implemented in the library to set user identifiers, which internally sets a custom key, but in the dashboard shows several crashes from the same user more nicely. Furthermore, the library enables custom log messages to be sent, which are then added to crash reports as log messages. With such log messages, more information about what led to the crash or error could be gathered. Another advertised feature from Firebase Crashlytics is the possibility to enable opt-in reporting, so that the library (and therefore the crash reporting) will only be started upon explicitly initializing the library, instead of automatically. In that way, an opt-in consent of the user can be achieved [38].

Additional Restrictions There are no additional restrictions introduced in the application when implementing the Android library. However, the minimum SDK version is defined as 16, which means that the application will be limited to be installed on devices with Android version 4.1 and higher. While this includes nearly all active Android devices, and has no effect if the application itself is not declaring a lower minimum SDK version - which most applications are not, as according to the official site, it is recommended to support only at least 90% of platform versions [42] - competing libraries declare a minimum SDK version of 14, which includes 0.3% more devices according to the Android version distribution table 2 [41].

Summary Table 3 summarizes the analysis of Firebase Crashlytics in regards to the aspects defined in section 2.2.

Website	firebase.google.com/docs/crashlytics
Pricing	Free
Self-hosted	No
Effort to set-up	Trivial
Set of features	Error & crash reporting, Android & iOS, custom keys and log messages
minSdkVersion	16

Table 3: Summary of Google Firebase Crashlytics

2.2.2 Android Vitals

The Google Play Store itself provides also a platform for crash and Application Not Responding (ANR) reporting, called *Android Vitals*, which is accessible through the Google Play Store. Application crashes, events where applications are not being responsive to the user, called ANR, as well as excessive app wake-ups in the background are automatically detected and uploaded to the Google Play Store Console [10]. There, a separate section lists all recorded crashes and errors as seen in figure 3. Therefore, this data is available without any additional setup in the Play Store for all applications which are installed on a device which has the Play Store (and Play Services) installed, with restrictions as stated below. The data is on eligible devices collected automatically, if the user has not opted out of this data sharing in the device settings. In order to not be able to identify single user of applications if only little data for an application is available, the Android Vitals dashboard will hide the data until enough devices shared data so that anonymized reports can be generated [10].

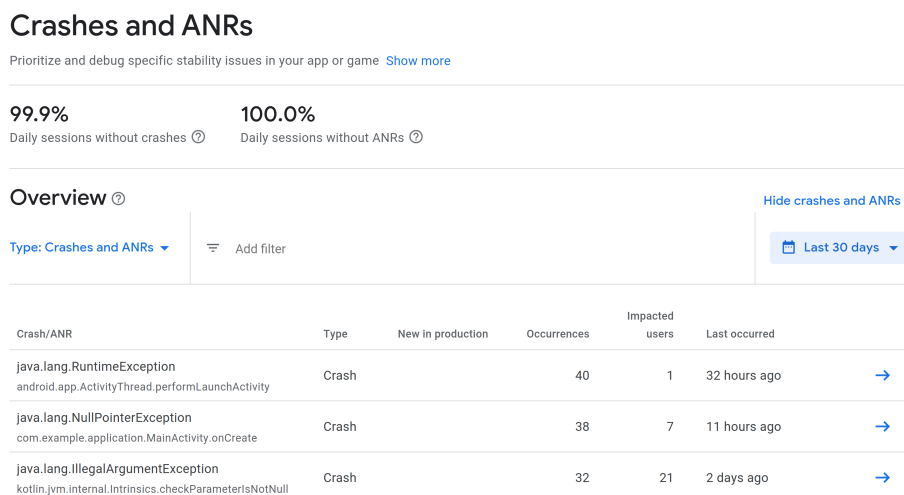


Figure 3: Crashes and ANRs listed in the Google Play Console. *Screenshot from play.google.com/console*

Pricing

Android Vitals are free of any charge, but there is the precondition that the application is published in the play store. When registering a 'Google Play Developer account', a one-time fee of \$25 USD is charged. This is a one-time payment in order to be able to use the Google Play Store as a Developer to e.g. publish applications [31]. As such an account is needed to gain access to the Android Vitals, this fee has been evaluated to be required to use Android Vitals.

Self-Hosted

Android Vitals is part of the Google Play Store Console and therefore not able to run custom / self hosted. There is no specific privacy policy for the Google Play Store, the Google Play

Store Console or the Android Vitals available, on all sites the default privacy policy of google is referenced [43]. While those privacy terms state, that personally identifiable information is only shared for legal reasons, the explicit consent or affiliated businesses for processing in the given privacy policy, the privacy policy states *non-personally identifiable information* may be shared publicly and with any partners [43].

Effort to set-up

There is no in-app setup or any enabling / inclusion of a library needed, as the crash reporting is performed on a system level, and not on an application level. Therefore, the effort to set up this kind of crash reporting system is not more than publishing the application in the Google Play Store. As this is already done / needs to be done for most apps, this will not be seen as additional effort for including the crash reporting system into the application. Therefore, there is practically no additional effort needed.

Set of features

The Android Vitals can automatically detect crashes as well as ANRs. However, as this system collects crashes on a system level, Android Vitals provide much more data [32]:

- Battery usage: stuck wake locks as well as excessive application wake-ups
- Stability: ANR as well as crashes
- Render time: Render cycles that lead to lowered frame rates as well as frozen UI frames
- Application startup time: The time the application needs to start (from different states)
- Permission denials by the user

However, as there is no library to be included in the application itself, it is not possible to add custom logs, custom keys or non-fatal errors to the collected data.

Additional Restrictions As no separate library needs to be included into the application, there is no restriction in terms of the minimum SDK version. However, the application must be distributed via the Google Play store, and the smartphone must have the Google Play Services as well as the Google Play store installed.

Summary Table 4 summarizes the analysis of Android Vitals in regards to the aspects defined in section 2.2.

Website	developer.android.com/topic/performance/vitals
Pricing	USD 25 ^a
Self-hosted	No
Effort to set-up	No additional setup needed
Set of features	Application crashes, ANR, excessive wake-ups, performance issues
minSdkVersion	None

^aOne-time payment required to create a developer account to publish applications

Table 4: Summary of Android Vitals

2.2.3 ACRA

ACRA (acronym for **A**pplication **C**rash **R**eports for **A**ndroid) is an Android library which enables to automatically upload crash reports of an application to a report server. ACRA is open sourced and does not provide a hosted instance of a crash report server, so it is self hosted only.

Pricing

ACRA is open source and does not introduce any payments of licensing fees. The Android library is licensed under the Apache 2.0 License, officially referenced back-end services for ACRA are also open sourced and also to not introduce any payments. With the Apache 2.0 license, commercial and private use as well as distribution is permitted without any charges [26].

Self-Hosted

ACRA itself is only the Android library part, but officially references two back-end projects which are built for ACRA: *Acralyzer* was built as the official back-end services for ACRA, which is able to store crash reports as well as analyze them. This project, however, has been unmaintained for some years. *Acrarium* is the recommended alternative back-end, which is in active development, but has not reached a stable release yet. Figure 4 shows the *Acrarium* dashboard. There are a number of other back-ends (both closed source as well as open sourced) available, which are compatible with ACRA's crash reports. Therefore, ACRA can be used in applications with back-ends running in a self-hosted, open source way, as well as using a provider for hosted ACRA back-end services, which may introduce charges as well as data accumulation between multiple applications.

Effort to set-up

As ACRA offers a lot of configuration options and a variety of how crashes should be logged, the implementation of the Android library requires more steps than other solutions. There are multiple dependencies which can be included depending on which plugins or configurations are

Acrarium

Logout

Bugs

Reports

Statistics

Admin

Reports

Latest Rep...

Title

Solved

Filter

☒ Hide Solved

6

moments ago

1

1

java.lang.RuntimeException: Manually triggered non-fatal exception

Not solved

1

moments ago

1

1

java.lang.RuntimeException: Manually triggered fatal exception on main thread

Not solved

1

moments ago

1

1

kotlin.UninitializedPropertyAccessException: lateinit property configuration has not been initialized

Not solved

Figure 4: Dashboard of *Acrarium*, a back-end and dashboard service for ACRA. Screenshot from a self-hosted instance of *Acrarium*.

desired. In the simplest configuration, where crashes are solely sent to a back-end service, only one dependency needs to be included. Furthermore, how crashes should be logged, and how they should be sent and displayed needs to be configured, as described in [2]. ACRA provides the different options as different plugins that can be included and configured. Again, in the simplest configuration, this would be a simple upload to the back-end service. To perform that, the HTTP plugin needs to be configured respectively with the domain as well as basic authentication parameters of the back-end service, as shown in listing 2.1.

Listing 2.1: Sample configuration of the ACRA library

```

1 @AcraCore(
2     buildConfigClass = BuildConfig.class,
3     reportFormat = StringFormat.JSON
4 )
5 @AcraHttpSender(
6     uri = "https://example.com/acra/report",
7     httpMethod = HttpSender.Method.POST,
8     basicAuthLogin = "username",
9     basicAuthPassword = "password"
10 )
11 public class MyApplication extends Application {
12     @Override
13     protected void attachBaseContext(Context base) {
14         super.attachBaseContext(base);
15         ACRA.init(this);
16     }
17 }

```

As ACRA needs a self-hosted back-end to be set up, this needs to be done, too. As mentioned before, there are currently two officially referenced back-end services as well as a lot of other back-end services customarily developed. For this comparison, the two officially referenced back-end services have been analyzed:

Setting up Acralyzer is a bit more complicated, as there is no official containerized solution available, but the two components, the storage and the web application, need to be installed separately.

The Acrarium back-end be set up by using docker. On the GitHub page, a sample `docker-compose` file is provided, which sets up Acrarium with a respective MySQL database to store the data in.

Set of features

The Android library ACRA consists of different artifacts which provide different functionality for the library. There are different implementations on where and how to send the crash report, either via *HTTP* to a back-end service, via *E-Mail* or to provide a custom implementation to send the data to a custom destination [4]. Furthermore, in-application actions can be activated by using additional artifacts, in order to show a user-facing dialog, notification or toast message when a crash occurred [3]. There is also the possibility to include additional plugins for advanced features available, such as limiting the number of reports sent [2]. While there are various options to configure the library and how crashes should be handled, there is no possibility to add non-fatal errors to the report.

Additional Restrictions Including the Android library into the application does not introduce any additional restrictions other than the minimum SDK version, which is set as 14, which is the Android version 4.0. While this does not include all SDK versions, SDK level 14 is the lowest among competing libraries, and includes 99.7% of all active Android devices according to table 2 [41]. Furthermore, the *androidx* libraries (formerly *support libraries*), which are provided by the Android framework to support features introduced in newer Android versions for earlier Android versions declare a minimum SDK version of 14 as well [9]. Therefore it is unlikely that any modern application will declare a minimum SDK version below 14.

Summary Table 5 summarizes the analysis of ACRA in regards to the aspects defined in section 2.2.

Website	www.acra.ch
Pricing	Free
Self-hosted	Yes
Effort to set-up	Android: configuration needed; Back-end: Non-Trivial set-up needed.
Set of features	Crash reporting, Multiple forms of reporting
minSdkVersion	14

Table 5: Summary of ACRA

2.2.4 Sentry

Sentry is an application monitoring platform for a variety of different platforms and technology stacks, including Android applications. Figure 5 shows the dashboard of Sentry with the list of the latest and top application errors.

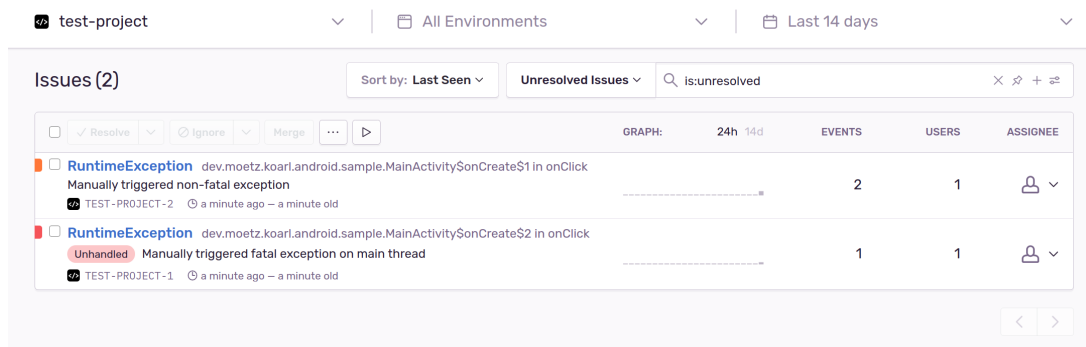


Figure 5: Dashboard of Sentry, which lists the top application crashes from the last 14 days. *Screenshot from sentry.io*

Pricing

Sentry offers three different plans as well as enterprise contracts, where only one of the plans is free of charge. In the free plan ('Developer'), the number of errors and transactions is limited to include 5000 errors and 10000 transactions. The crashes and errors are only retained for 1 month and advanced features such as email alerts, different integrations as well as custom keys are not featured in this plan [77]. The cheapest plan where those features are integrated charges \$ 26 per month as of November 2020, which also increases all quotas from the free plan and retains the reports for 90 days.

Self-Hosted

Sentry only offers to provide hosted solutions, and their back-end is not open sourced. Furthermore is the location can only be chosen between different locations in the enterprise pricing plan, whereas in other plans the data storage and processing can be done in any country Sentry maintains service provider facilities according to their privacy statement [78].

Effort to set-up

The Sentry Android library is straightforward to include, as only the dependency needs to be included, and the Domain Name Server (DNS) of the Sentry dashboard project needs to be declared in the applications manifest file. This will allow Sentry to auto-initialize itself on application start.

Also, a project needs to be created in the Sentry dashboard. This is possible free of charge in the Developer plan, and after account creation the type of applications as well as an organization name needs to be provided for the dashboard, and the dashboard is set up.

Set of features

Sentry provides a large set of features, however, a lot of them are only available in a non-free pricing plan. The Android library both automatically catches application crashes as well as has the possibility to submit non-fatal exceptions. For all events (crashes and errors) the library offers the possibility to add a interceptor, which may alter the event as well as filter events to not be sent to the dashboard. Furthermore, user attributes, which help to identify the user at which a crash / error occurred, custom keys to add additional information about the situation or the user as well as so-called breadcrumbs to be able to add steps which led to the crash / error can be set.

Additional Restrictions Sentry declares a minimum SDK version of 16, which, in the same way as *Google Firebase*, may introduce problems if the application is targeting devices running Android on a version lower than 4.1. *ACRA*, being the most liberal library in terms of the minimum SDK version, may supports 0.3% older devices mode, according to the Android version distribution table 2 [41].

Summary Table 6 summarizes the analysis of Sentry in regards to the aspects defined in section 2.2.

Website	sentry.io
Pricing	Only a limited version available for free
Self-hosted	No
Effort to set-up	Trivial
Set of features	Error & crash reporting, report filtering, custom keys & breadcrumbs
minSdkVersion	16

Table 6: Summary of Sentry

2.2.5 Summary

In table 7 a summarized comparison of the different analyzed solutions can be seen.

	Firebase	Android Vitals	ACRA	Sentry
Pricing	Free	USD 25	Free	Free (limited)
Self-hosted	No	No	Yes	No
Effort to set-up	Trivial	No set-up needed	Not trivial	Trivial
minSdkVersion	16	None	14	16
Restrictions	None	Distribution through PlayStore	None	None

Table 7: Summary of the comparison of the different existing solutions

3 Methodology

3.1 Source of requirements

The requirements have been set with two main perspectives in mind: The first being the application user, who wants errors and crashes to be reported automatically so that an error can be addressed and fixed by the developers as fast as possible, but without sharing too much personal data. The second perspective being the application developer, who wants an easy solution to get insights over errors and crashes that occurred when users used the application, so that the error can be addressed and fixed as fast as possible. From the perspective of an application developer, such a crash reporting system should easily gain access to application crashes and errors, and should be able to set up easily and without much effort. This is due to the fact that this often is seen as a *nice to have* feature in smaller projects with little budget, and therefore the setup should not be dismissed due to the effort it takes to set up such a system, as stated in section 1.1. Moreover, with the users privacy in mind, as little personable identifiable data as possible should be shared from an application, and the data being sent to a back-end service should only be stored and used in a non-privacy-invading way. Especially with different laws in place for storing and processing personal data, such as the General Data Protection Regulation (GDPR) [18] effective for European users or the California Consumer Privacy Act (CCPA) [70] effective for all application user from California.

With that in mind, another key factor in the declaration of the requirements has been the analysis of already existing systems of the systems listed in section 2.2. As those systems (among others, which have not been analyzed due to the reasons stated in section 2.1) are already used and accepted widely in the Android developer community, the set of features they share in common has been seen as an indicator for what is either needed or desirable in a crash reporting system, if they are not in conflict with other requirements the system architecture for a

privacy aware system. Such common features have been the automatic detection of application crashes as well as the possibility to log custom errors which did not lead to application crashes. The different tools often provided more features, but those two were present in all tools and were not in conflict with any other requirement.

3.2 Comparison of tools

The different tools for collection crash and error logs from Android applications are going to be compared with the following criteria, which has been set up by the given requirements. The developed system is going to be compared to all existing tools using the same criteria.

Open-Source: Is the source-code of the system available and to be inspected by anyone? This includes not only the Android library to be included in the application, but also the back-end data storage as well as the dashboard to analyze the data.

Self-Hosted: Is it possible to setup the data storage server and dashboard on a company-owned server, or is the application only available through servers provided by the system? While most systems that are open source also are able to be self-hosted, this is not a must have, and therefore this has been extracted as an extra criteria. Furthermore, a self-hosted application does not need to be open-sourced, as the application could be provided as a binary package.

Data accumulation: Is there any information about data accumulation - also between multiple applications sending crash reports - being performed by the system which is more than needed for basic crash analysis? In open-source systems the code can be checked whether this is the case, in closed source systems the privacy policy of the system will be analyzed for any statements for that. Furthermore, will the collected data be shared with other, third party companies?

Documentation: Is there documentation available for the system, which makes the setup easy as well as provides all necessary information?

Effort to set up: How much effort is needed to set the system up? This will be subjectively compared with the other systems in terms of how much time as well as technical or specific knowledge is needed to set the system up, and how complex the setup process is. This includes both the setup process of the Android library as well as the process to set up the back-end service.

Data analysis: Is there a dashboard / website available to inspect the collected data? Is it possible to extract the data to include it in another dashboard / system as well?

3.3 Approach for a new system

After the analysis of existing systems as well as after the requirements of section 4 have been set as stated in section 3.1, it was decided to develop a new crash reporting system, as stated in section 6.1. This system has been developed with privacy by design, so that every aspect of the system and the architecture is considering the privacy of the user first, so that as little data as possible about the user is shared. This includes sharing as much data as needed to be able to get enough insights into the crash but as little data as possible and if possible no data or identifier that could identify one particular user within the dataset. Furthermore, this system should be free of charge, as stated in section 4.1, and therefore the decision to make the system able to be self hosted has been done.

Additionally, as the Android library as part of the system is by the platform required to be developed in either Java or Kotlin, where Kotlin is the state of the art and preferred language to build applications as well as libraries, this part has been developed in Kotlin [16]. As for the back-end service, multiple options have been considered. However, due to the technology stack of Android developers being Kotlin, and Kotlin can also be used to write back-end services in, Kotlin has also been chosen to develop the back-end service in. This empowers synergies between the Android library and the back-end service, such as sharing one module which contains models as well as serializer logic, which then can be used by both parts. Furthermore, as Kotlin is directed from targeting the Java Virtual Machine (JVM) to also being able compile to native code as well, which is to be included also in iOS applications, the Android library could be developed so that all business logic code is developed once, and only the platform specific parts are developed for the respective platforms in the future [53].

3.4 Evaluation of results

The existing tools as well as the developed system will be evaluated according to the requirements set in section 4 as well as the points declared in section 3.2. While some of the points of section 3.2 can be clearly answered for each system, the question whether data accumulation is possible as well as the effort to set up will be evaluated as defined in section 3.2. The developed tool will be evaluated with the same criteria, to be able to compare the new system with the already existing systems.

4 Requirements to a privacy aware crash reporting system

In this section the requirements as well as their reasoning will be elaborated. These requirements have been defined due to the motivation stated in section 1.1. A crash reporting system as a system that will collect data produced by actions from users and on end-user devices, may contain data that lead to identification of users as well as the possibility to track users. In the case of an crash reporting system, it is a nice-to-have feature, but not required to include a user identifier or other data which connects a report to an application user with the data in order enable the developer to analyze the error. As stated in section 1.1, it is desirable for a privacy-aware data collection system to only collect as little data as needed in order to respect a users privacy. Furthermore, within the collected data, as far as possible no personally identifiable data should be contained. The general direction on how those requirements have been found is stated in section 3.1.

4.1 Free to use

The implementation of the system as well as the usage should be free of any charges. This includes licensing fees, one-time payments as well as any other payments. Many already available systems are completely free of charge, but may not be as privacy-aware as other solutions, as stated in section 2.2. Therefore, solutions which honor the privacy of user data more should also not introduce any cost, as often there is no extra budget for such a system, and if a privacy-aware system would introduce costs, then another, free system would win the choice. Furthermore, as payments may discriminate smaller projects without a budget, such tools should not introduce any fees at all.

4.2 Open Source

In order to able to proof that a system really behaves in the way it adverts, e.g. in stating that data is not being shared with third parties, the validity of that statement can either be checked by trusted authorities, e.g. third party companies inspecting the code, or the code can be made open source in order to let anyone inspect the code to see what the code is actually doing [56]. In Open Source Software also the requirements for the real use-cases are often better shown, as people who will use the software later on can actively contribute to

design choices as well as state their requirements to the software and even develop them [56]. Furthermore, as security audits of the code may be a positive indicator, such audits are often performed by a single external company, which may miss issues or threats as well as may not be the best in their field. As an example, the mobile messaging application *Threema*, which strives to provide a top secure messaging solution, has been closed source for several years with regular external reviews. While their applications source code has been reverse engineered by various security experts and validated to be secure, the code was never officially public and therefore not fully analyzable, especially the back-end systems. Threema is now planning to fully open source their applications, in order to enable anyone to independently review their security implementation [88].

4.3 Possibility to self-host the back-end service

The possibility to host the back-end service on an owned and controlled environment is a benefit for various reasons. First of all, this helps in providing the service completely free of charge, as no server infrastructure needs to be provided by the provider, which would generate costs that would need to be accounted to the system user. This could be done in licensing fees or pricing plans, or by trading the collected data to advertising companies, which would harm the set privacy principles. Furthermore, by running the back-end service on a controlled environment in combination with the open sourced software (see section 4.2), it can be assured, that the application that is running actually is built from the inspected source code. While open source itself declares that the source code of an application can be inspected, it does not guaranteed that the published source code is the actual code of the application that is run on provided servers. Moreover, the data is also being stored in a controlled environment, so it can - additionally to analyzing the applications source code - be assured that the collected data is being stored securely and not being shared outside the system. Further on, as most application need a back-end system already in place, which means that for the application there most likely is already some kind of server infrastructure set up, the back-end system can be added to that infrastructure. This has the advantage that the back-end of the crash reporting service may automatically be scaled up in the same way as the application back-end.

With the ability to self-host an application, the application can also be hosted in a cloud service, where the infrastructure and basic security principles are already set-up and do not need to be considered by the application user. Furthermore, such cloud services are often very easily scale-able and have a pay-by-use pricing model. Additionally, if the application where the crash reporting system should be added does not have a back-end service in place, low-cost cloud solutions to host the crash reporting systems back-end service can be easily set up as well as can then be used for other services as well. Furthermore, by providing the back-end service in a state of the art, well established containerized environment, vendor lock-ins can be prevented. In that way, the crash reporting system would not tie the application developer to one specific pricing model, but would only provide the software and the application developer

may choose a hosting provider by their choice.

4.4 Easy to set up

The system should not take much effort to be set-up. This includes the library that needs to be included into the Android application as well as the back-end system. This requirement has been set as most of the already existing systems require very little effort to be set up as well. Furthermore, most of the systems provide the back-end as a hosted solution, where the service does not need to be set-up, but most often just a new project or application needs to be added. Moreover, as the effort to set such a system up takes time, and time is a resource that needs to be paid by a project budget, the effort as well should cost as little as possible in terms of time. Also, as the back-end system needs to be set-up, and this project should not discriminate smaller projects or projects that do not already include a back-end service, the back-end service and therefore not necessarily a back-end developer or someone with DevOps know-how, it should not be technically sophisticated nor complicated, in order for Android developers to be able to get the system running. Therefore, the set-up and configuration of the library should be as easy as possible, especially in a default configuration, and the set-up of the back-end service should be possible without much back-end domain knowledge, as this should also be possible for developers specialized in Android development.

4.5 Configurable

Another requirement defined for a crash reporting system is for it to be configurable. As easy as the set-up should be, the crash reporting system should be customizable in how it behaves and what it does. Such customization could be as to which back-end service the data should be sent, or where and how the data should be stored locally, and how and when it should be uploaded to a back-end server. By enabling such customization, the system does not work solely as a black box to the application developer, but enables application developer to configure the system in the way it should behave, without the need to program the system themselves. Furthermore, the crash reporting system should be able to be disabled as easy as possible, to give application developer the possibility to enable crash reporting only for specific testing apps as well as to provide the option for the application user to be able to disable the data upload at all as for example an action item in the settings section of the application the library has been included.

4.6 Non-restrictive

The last defined requirements is the fact that the system and library should not introduce any restrictions in the application. This means, that when including the library, there should not be a new restriction in how the application can be used or behaves compared to when not including the library. This may include any restriction, but in the context of Android application especially two essential points, which not all libraries satisfy:

- The library should not raise the so-called *minSdkVersion* of the application. This field defines the minimum version of Android the application can be installed on. As Android is a very heterogeneous platform in terms of which versions of the operating system are still in use, as seen in table 2 [41], this minimum version should be set as low as possible, to not exclude users with older devices. The minimum SDK version of the application is defined by the `minSdkVersion` the application defines, and none of the included libraries may have a higher `minSdkVersion` defined. Therefore, if the library would declare a rather high minimum SDK version, this would introduce a new restriction to the application.
- Furthermore, even though the Play Store is currently the most often used way of publishing and distributing applications to Android devices, the Android ecosystem has always been open for other way to distribute applications [34]. While for some time, the Google Play Store still was the most common option to download apps from, in recent years some manufacturers have launched their own distribution system, and especially in China alternative application stores due to the absence of Google Play there, have evolved [92]. Therefore another matter that may not be restricted is the way of distributing the application, which means that the library should be able to report crashes and errors despite the way the application has been distributed.

5 Suggested solution

5.1 Development of the new system

A new system has been developed as none of the analyzed systems in section 2.2 has met all requirements of section 4. For the basic technology stack, Kotlin has been chosen due to several reasons. While it is a modern programming language, which has been introduced in 2011 as an alternative programming language targeting the JVM [58], it is not only open sourced [94], but also well-tested as JetBrains, the company who developed Kotlin, uses the language ever since for their widely used set of IntelliJ Integrated Development Environment (IDE) family [51]. Furthermore, in a developer survey carried out by StackOverflow in 2019, Kotlin has been voted fourth most loved programming language by a total of nearly 90.000 developers taking the survey [84]. As Android applications are running on the JVM and written in Java, the usage of Kotlin as an alternative to Java as programming language has ever been possible. After the usage of Kotlin has increased steadily in the Android developer community, Google has officially announced in May 2019 that Kotlin has been chosen to be the preferred programming language for Android applications as well as Android APIs [59]. Therefore, the Android library which needs to be developed would be developed in the same language as Android applications which enables a smoother implementation and usage of the library. While Kotlin has been more popular, the language and some accompanied libraries have made their way into back-end development as well. This was performed by using already existing systems targeting the JVM, but developing the applications using Kotlin. Therefore, by choosing Kotlin as programming language for the back-end application as well, only one language needed to be used for both applications. Furthermore, the usage of the same programming language introduces advantages like class and model sharing. Moreover, in the last years Kotlin has been expanded from only targeting the JVM to also be transpiled¹ to JavaScript or compiled to native code. This enables libraries or whole application for i.e. iOS to be developed using Kotlin, and to be able to share either the whole code-base or single modules containing e.g. business logic in Kotlin. This enables the extension of the Android library for iOS as well by rewriting as little code as possible [96].

¹ Source-to-source compilation is referred to as transpilation [5]. As JavaScript is an interpreted language, transpilation instead of compilation is performed.

5.2 Architectural overview of the system

The system needs two major parts, which are the Android library which is going to be implemented in the Android application, and the back-end service which collects and stores the data sent by the Android library, as seen in figure 6.

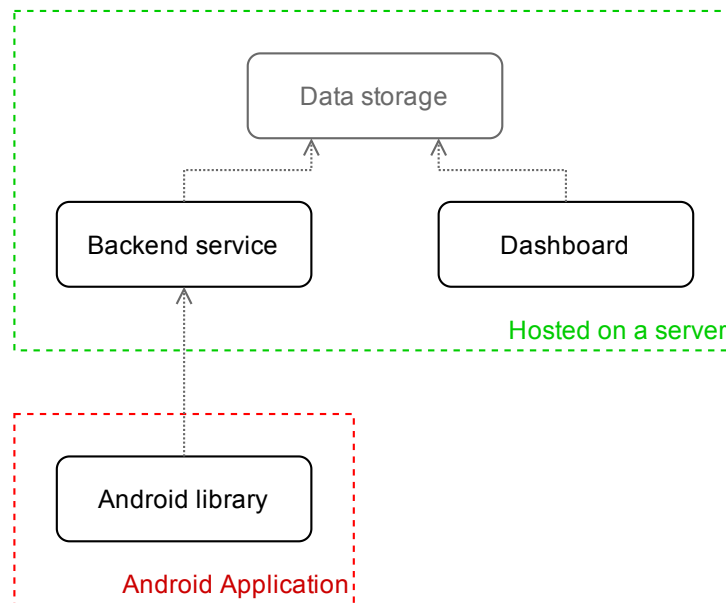


Figure 6: Diagram of the architecture of the systems involved

More specifically, the Android library will automatically catch application crashes as well as provide the possibility for the application developer to report non-fatal errors. Those reports will then be sent to the back-end service. This will happen either immediately, or on a later point if the application has been shut down or no connectivity to the back-end service is available. Therefore, the Android library also needs to implement some sort of caching logic for all data which should be sent to the back-end server. The back-end server provides an API for the crashes to be sent at, and will store them persistently in a data storage. The dashboard, which should only be available to the application developer, will then display the data stored in the data storage to the application developer in order to be able to analyze the crashes and errors. Due to the requirements of the system to be able to be self hosted (see section 4.3) and to be easy to set up (see section 4.4), the back-end service as well as the dashboard, which should be hosted on a server, is provided using a containerization software.

5.3 Architectural decisions in regards to the requirements

The requirements for a privacy aware crash reporting system have been stated in section 4. To be able to guarantee those requirements, especially for means of privacy considerations, the architecture of the system needs to be prepared.

For the first requirement stated in section 4.1, which declares the system to be free to use, the back-end system has been chosen to be solely self-hostable. This means, that there is no provided solution available, and the back-end service needs to be hosted by the user of the service. This has been decided, as the system should not introduce any costs, despite the development effort, for the system developer. If a back-end service would be provided, this would introduce recurring server costs, which would need to be compensated by some of income at some point in order to keep the system running. This income could either be some sort of charged licensing, donations, advertising or sales of collected data to i.e. advertising companies. Those solutions, however, would either not meet the requirement to be free to use, or would be in conflict with other requirements, such as the requirement to be privacy aware, as defined in section 3.1.

For the the back-end service to be able to be self-hosted, as stated in section 4.3, an easy way to run the back-end system on different systems should be used. While the installation of the back-end application should be kept as simple as possible, running an application on different host systems and operating systems can introduce issues on different platforms. Issues may include conflicting dependencies from already installed software on the server, missing dependencies that need to be installed first or platform differences of different operating system distributions. While using Kotlin in a JVM environment aims to remove such issues on an application level, a Java Runtime Environment (JRE) which itself has different versions must be pre-installed as a dependency. Furthermore, as data storage an external database system is used, this would need to be installed, too. To leverage such problems, docker as a containerization technique has evolved to be a state-of-the-art solution to easily provide a runnable software in a container with all its dependencies pre-installed, which can be run on any UNIX system [93, 64, 23]. Therefore, to make the deployment of the back-end system as simple as possible with as little error-possibilities as possible, the back-end application itself is provided as a docker image. The usage of *docker* therefore leverages the requirement for the system to be easy to be set up, as stated in section 4.4. With the use of docker and the back-end service being provided as a container, the actual effort to set up the back-end service is kept to a minimum, as shown in section 5.4.2. Using docker also facilitates the requirement to be non-restrictive, as stated in 4.6, in enabling the back-end service to be run on any Linux, Windows and macOS operating system [19]. Furthermore, using *docker-compose*, the database needed for running the back-end service is also provided without no additional setup. For the data storage, the provided example `docker-compose` file uses a MySQL database, which is being connected to the back-end service. However, as stated in section 4.5, the system should also be customizable. Therefore, the back-end service is configurable to be connected to nearly any Structured Query Language (SQL) based database. More details about the supported databases are defined in section 5.4.2.

The Android library, which needs to be included by the application developers, should be easy to implement as well, as stated in section 4.4. However, while the library should be easy to be set up, it should also be configurable as stated in section 4.5. In order to meet both demands,

the type-safe builders [52] of the programming language *Kotlin* have been used, which allow easy and type-safe optional configurations.

Furthermore, the communication between the Android library and the back-end system is performed using a Representational state transfer (REST) service using JSON, which is a widely used technique to send data, and is usable from any platform and any programming language [89]. By providing a documentation for that service, it is made possible to replace either the Android library or the back-end service with a custom solution. A use-case for that would be to include the data storage in an already existing back-end for the app, but using the Android library in order to collect the error and crash data.

Moreover, the data shared between the Android library and the back-end does not contain any identifier which could connect two crashes uniquely to another. While the Android library will still send the specifications of the device the error occurred on by default, in a larger user-base this data cannot be used to reference multiple error reports to the same user. If a smaller user-base is anticipated, the library can be configured to not share the device specification, which will remove the device specification from the report and therefore from the possibility to pinpoint an error on a specific device or operating system version, but prevents referencing multiple error reports to a specific user. A full exemplary data model shared between the Android library and the back-end can be seen in listing B.2 in appendix B. There can be seen that only crashes get assigned a random Universally unique identifier (UUID), which is not reused and therefore cannot be used to reference multiple crashes to one device. By not providing any identifier or possibility in the data which could connect multiple error reports to another, the data model prevents aggregation of data from a single user by default.

5.4 Components of the system

As stated in section 5.2 and displayed in figure 6, the system needs several components. This section will analyze each component by outlining their functionality, as well as which technological stack has been used for its implementation. Figure 7 shows in more detail which technological components are in place and their technological stack that has been used.

5.4.1 Android library

The Android library is responsible to collect error and crash data, and to send that data securely to the back-end service. While the tasks for the library seems simple, the data needs to be cached locally for cases when the application has been shut down and there is no possibility to immediately send the data, or when no connection to the back-end service is possible. Furthermore, as this library is going to be included in applications and then work in the background, the library should at no circumstances lead to application crashes itself. This is especially true, as the back-end service as external service may be updated to a newer version in the future, or be shut down or unreachable for the application for any reason, which must not lead to crashes

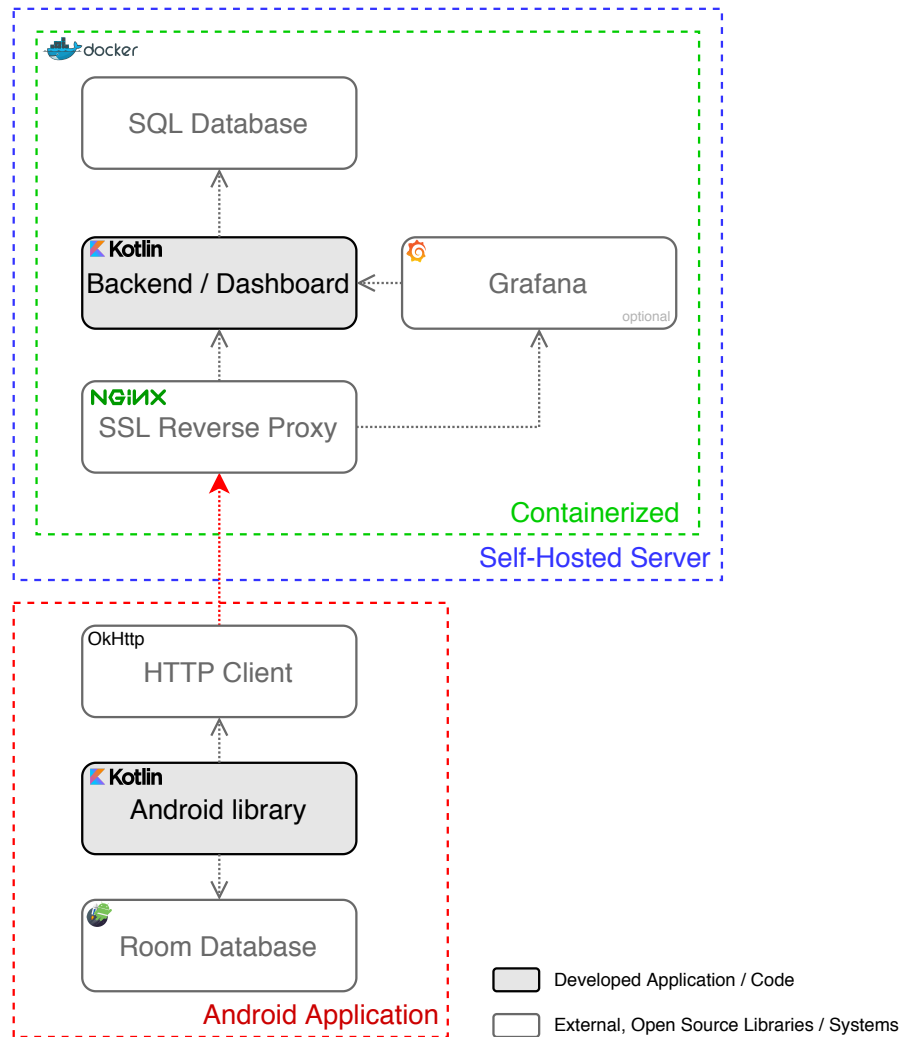


Figure 7: Schematic diagram of the developed and used components for the Android library and the back-end service.

when the library encounters error responses. This problem arose in the past, most recent and with an impact on a lot of popular applications with the Facebook iOS SDK [62], leading to sudden numerous crashes, which could not be solved by the application developers themselves.

Exception handling

The Android library should automatically report all application crashes. There are two reasons for an application to crash. Either the application has thrown an exception that has not been caught within the application, or a native library within the application has emitted an unhandled signal [7].

The JVM provide a functionality called *uncaught exception handler* for exceptions which were not caught in threads [71]. This enables the developer to add a handler which will be invoked if an exception has been thrown and not caught in a specific thread. Furthermore, a default

handler for any exception not being handled by an explicitly set handler for a given thread can be set, which will be invoked for any thread. Using this mechanism, the Android library can create a handler that will be invoked on any uncaught exception which occurred on any thread. Listing 5.1 shows a simplistic sample on how such an default uncaught exception handler can be defined. As those exceptions are the triggers for application crashes, this can be used to save the data of the crash just before the JVM will shut down and the application will crash.

Listing 5.1: Sample usage of registering an uncaught exception handler

```
1 Thread.setDefaultUncaughtExceptionHandler { thread, throwable ->
2     // This lambda will be called on an uncaught exception with the thread
3     // and throwable which was uncaught.
4 }
```

Local cache

The Android library is responsible for catching error and crash reports and to send them to the back-end service. However, as the notification about an application crash is done using the uncaught exception handler, which is being notified about an uncaught exception on the event, and just before the application actually crashes and the JVM will be shut down, it is not feasible to send the report right away to the back-end service. Furthermore, the smartphone, on which the Android application is running on, may not be connected to a network at any time, whereby sending the report is not possible. Therefore, the data needs to be cached locally to be able to send the data to the back-end service. The data, however, will only be stored on the device until it has been sent successfully to the back-end service, and will then be deleted.

For the local data storage, multiple options are available. For this library to be as small and lightweight as possible, third-party options such as *Realm* have not been considered, they often blow up the size of the library. Therefore, as data storage possibilities, the remaining options from the Android framework are the *Shared Preferences* as a key-value storage, the *File API* to store the data as files, or *SQLite* [65]. As the data to be stored are several objects, File-based and Shared Preference storage would require manual serialization and de-serialization of the data and no type safety. Therefore, the choice was to use *SQLite*, as it is a lightweight and fast relational database built into the framework, and with the use of the Android Room Persistence Library [8], which is part of the project *Android Jetpack*, the queries and schemata are automatically generated by generated code at compile-time, which makes the usage of SQLite type-safe while not introducing a new storage technique but using the built in SQLite engine.

Crash uploading

As mentioned in section 5.3, the crashes are uploaded to the back-end service using HTTP (HTTPS), and a REST service using JSON as content type. For the implementation of the upload service, again, multiple frameworks could be chosen. While Android provides

a very low level HTTP API, it is considered good practice to not use the platform provided `URLConnection`, but instead encapsulate the HTTP request in any popular third party library that is being shipped within the application [14]. Besides the fact that those libraries are often well tested, used by many other applications and open source, they encapsulate the whole HTTP request and response into an easy to use API. Furthermore, as they are shipped within the application and not within the framework itself which is part of the operating system, they are much easier to update and will contain the same behavior across different operating system versions.

As the Android library has been developed in Kotlin, HTTP libraries which are either written in Kotlin or to be integrated easily using Kotlin have been preferred. The library *OkHttp* is written in Kotlin and very well adopted in the Android developer community [14], this library as well as *Ktor*, which is the library used within the back-end service as an HTTP server framework, which provides an HTTP client as well, have been considered. Ktor is a wrapping library around real implementations of HTTP clients making the code ready to be run on Kotlin Multiplatform (KMP), and includes for android using *OkHttp* as well as the *URLConnection*. While both libraries are very easily integrated into the library using Kotlin, *OkHttp* has been chosen as it shows a significant smaller size than *Ktor-OkHttp* as shown in table 8, which is an important number in Android library development, as the library should not introduce a too large footprint for the final application. As *Ktor-Android* is built using the *URLConnection* it does not introduce a much larger overhead than *OkHttp*, but does not provide any additional features such as certificate pinning. Furthermore, *Retrofit*, another popular library for creating type-safe HTTP calls [14], has not been considered, as this library does only create a type-safe API for the developer around the usage of *OkHttp*, which would for the small use-case of the upload of the crash data would only introduce a larger footprint for the final application size.

Library usage

As for the requirement stated in section 4.4, the library should be easy to set up. This includes that there should be very few steps easily performed by Android developers in order to include the library into an Android application. However, it is, due to the fact that the back-end system is self-hosted, not possible, that no configuration at all is needed, as the back-end service needs to be connected to the library. Therefore, in order to include the library into an Android application, two steps need to be performed. In the first step, the library needs to be included into the project. Android uses by default the build automation tool *Gradle* [11], which enables to include external libraries from *Maven* repositories by declaring its group identifier, artifact identifier as well as the version [44, 87]. This needs to be done in the *Gradle* configuration file for the Android application. Listing 5.2 shows how the android library can be implemented using *Gradle*. While *Gradle* is the most common build system to automate Android builds with and manage the dependencies, there are also other systems available, such as *Base*, *Ant*

² *minify* in the Android build system includes code shrinking, obfuscation and optimization [6]

	Ktor-OkHttp	Ktor-Android	OkHttp
	without minification		
APK size (raw)	4.4 MB	4.4 MB	3.3 MB
Number of classes	6975	6951	4848
Number of methods	49272	49151	39177
	with minification		
APK size (raw)	685.1 KB	585.2 KB	573.3 KB
Number of classes	1887	1661	1494
Number of methods	9330	8275	7542

Table 8: Comparison of the size of the APK and the compiled bytecode (Dalvik Executable (DEX) file) with the Android library either with OkHttp or with Ktor included, with and without minification². As minification removes unused code and performs code optimization, both application sizes (with and without minification) are shown here, as due to that process a former large library could have produced a smaller footprint.

or *Maven*. With those solutions, the library can either also be implemented using the *Maven* repository, or the library is also published as a Java Archive (JAR) artifact and its transitive dependencies, which can be included in the respective build automation tools.

Listing 5.2: Including the Android library (“Koarl”³) dependency into an Android application using Gradle

```

1 dependencies {
2     implementation "dev.moetz.koarl:koarl-android:0.1.0"
3 }

```

The second step is to initialize the library during runtime. This needs to be done as the domain / address of the back-end service needs to be set in the library custom for every application. Besides the configuration of the back-end service, this has deliberately been decided to force the Android application developer to explicitly enable the Android library instead of starting the library in the background automatically, as other solutions do. Doing that, the Android developer is more informed of when and which libraries are actually initialized and running in the background application. Furthermore, this enables an easy implementation of opt-in behavior instead of an opt-out behavior, if the Android developer wants the crash reporting in the Android application to be an opt-in feature in regard to the users privacy. For other systems which initialize themselves without the need to a method invocation implemented by the application developer, it is often needed to read the documentation in order to be able to change the library behavior into an opt-in behavior. Listing 5.3 shows the minimum configuration needed in order to initialize the library within an Android application. This is in most cases done in the

³Koarl is the working title of the system, which is a backronym of “**K**otlin **O**pen **S**ource **A**ndroid **C**rash **R**eporting **L**ibrary”

initialization phase of the application, which is performed in the *Application* class' `onCreate` method. However, this can be performed anywhere and anytime in the application, and only after the initialization has been invoked, the library will start to collect error and crash reports.

Listing 5.3: Initializing the Android library in the application at runtime

```
1 class MyApplication : Application() {  
2  
3     override fun onCreate() {  
4         super.onCreate()  
5  
6         Koarl.init(context = this) {  
7             // declare the base-url of the back-end service:  
8             baseUrl("https://koarl.dev/")  
9         }  
10    }  
11  
12 }
```

As stated in section 4.5, the library should not only be easy to set up, but also configurable. Listing 5.3 has shown the minimum required configuration, so the easiest way to initialize the library, as required in section 4.4. This initialization can be extended with various configuration options, which are explained in the following paragraphs and are shown in an example in listing 5.7.

Enable verbose logs To be able to debug the behavior of the application as well as be able to inspect the actual configuration that is used at runtime, the library exposes the functionality to enable debug logs, which log information about what the library is doing to be able to inspect as well as debug the behavior. In line 3 of listing 5.7 is shown how to enable or disable the debug logs by passing the respective boolean value.

Certificate pinner To secure the data transmission to the back-end service, HTTPS should be used, although it is not enforced. While using Transport Layer Security (TLS) as an encryption mechanism in using HTTPS, there are still attacks possible, e.g. Man-in-the-Middle attacks. This kind of attack can easily be prevented by pinning the SSL certificate that is being used by the server [15]. In order to enable that security measure, it is possible to define a *CertificatePinner* which contains the server certificate hash to ensure that a secure connection will only be established using the real servers certificate, which prevents man-in-the-middle attacks [80]. In listing 5.7 is shown how to declare such a certificate pinner in lines 17 to 21, where the second parameter passed in line 19 is the hash of the certificate to pin to.

Custom application name and version By default, the name of the application as well as the version of the application that are linked to an occurred error or crash are automatically retrieved from the application manifest. However, it is possible to set those values manually, in case the values should differ, as shown in listing 5.7 lines 7 and 8.

Disable device data sharing Crash and error reports contain device data by default, which include data such as the device model, manufacturer and Android version. As the sum of all those fields in a small set of users may enable to reference the device to a real user, the library provides the possibility to disable that data sharing. This can either be disabled by the application developer, or to be implemented as an opt-in or opt-out switch in the application itself. When initializing the library, this can be defined as seen in listing 5.7 in line 10.

Disable error reporting Furthermore, the library enables the possibility to disable all data collection and data sharing to the back-end server. The library can be disabled or enabled at any time within the application. Therefore, the library easily be disabled by the developer for i.e. development builds, the respective option in the builder construct can be seen in figure 5.7 in line 11. Moreover, the application can choose to offer this setting to the user, to give the application use more control over whether error reports may be shared to the back-end server. To be able to change this behavior at runtime without the need to restart the application, the library furthermore provides a method to disable this at any time within the application lifetime, without the need to re-initialize the library.

Custom local cache For caching the crashes until they have been successfully uploaded, a *Room* database is by default used. However, this database can be replaced by a custom implementation of the caching interface as seen in listing 5.7 in line 4, in order to e.g. use an already existing data storage logic. Listing 5.4 shows an empty implementation of a custom local repository, which shows that only the basic create, read and delete methods need to be implemented. All other logic for when to store, load or send crashes is still handled by the library. As *Room* does not implement any further security measures by default, which means that the data is not encrypted, and only secured by the platforms storage application storage sand-boxing, which can be disabled on rooted devices [65], a custom local repository with encrypted storage should be implemented if the crash reports should be encrypted additionally.

Listing 5.4: Defining a custom LocalRepo for the Android library

```
1 class MyLocalRepo : LocalRepo {
2
3     override suspend fun addCrash(crash: CrashUploadRequestBody.ApiCrash) {
4         // Store the given crash
5     }
6
7     override suspend fun getCrashes(): List<CrashUploadRequestBody.ApiCrash> {
8         // Return all stored crashes
9     }
10
11    override suspend fun removeCrashes(ids: List<String>) {
12        // Remove the crashes with the given ids
13    }
14
15 }
```

Custom crash upload handler The mechanism that uploads the crashes can be replaced by a custom implementation, as shown in listing 5.7 in line 5. This can be used if either another HTTP client library is already in use, or if an already existing back-end is used to store and display the crash reports, and therefore the reports should be sent in a custom way. By default, a provided *OkHttp* based upload service is used. In listing 5.5 can be seen that when using a custom crash uploader, the local caching of the crashes, as well as the retry logic is still performed by the library, and therefore the custom implementation only needs to try to upload the crash and signal back whether the upload was successful or not.

Listing 5.5: Defining a custom CrashUploader for the Android library

```
1 class MyCustomCrashUploader : CrashUploader {
2
3     override suspend fun uploadCrashes(
4         baseUrl: String ,
5         deviceData: ApiDeviceData?,
6         appData: ApiAppData ,
7         crashes: List<CrashUploadRequestBody.ApiCrash>
8     ): Boolean {
9         // Here the upload needs to be performed, and the returned boolean signals whether the
10        // upload has been successful.
11    }
12
13 }
```

Custom device state retriever The library offers a default implementation to record the current state of the device at the time of an error or crash. This logic can also be implemented in a custom way, if for example specific fields should not be recorded, as shown in listing 5.6, which gets included in listing 5.7 in line 6.

Listing 5.6: Defining a custom DeviceStateReceiver for the Android library

```
1 class MyCustomDeviceStateRetriever : DeviceStateRetriever {
2     override fun getDeviceState(): ApiDeviceState {
3         return ApiDeviceState(
4             freeMemory = /* the currently free RAM memory */,
5             totalMemory = /* the total available memory */,
6             orientation = /* the current orientation of the device (portrait or landscape) */
7         )
8     }
9
10    override fun getDeviceData(): ApiDeviceData {
11        return /* Needs to return the custom queried device data. */
12    }
13 }
```

Alter timing settings Moreover, the library enables different timings settings, such as the delay after the application startup to first send reports to the back-end server, to be customized, as shown in listing 5.7 in line 13. This can be necessary on devices with lower computing power in order to not increase the start-up time of the application and therefore the user experience [36].

Listing 5.7: Android library initialization with all possible configuration options

```
1 Koarl.init(context = this) {  
2     baseUrl("https://koarl.dev/")  
3     debugLogsEnabled(false)  
4     customLocalRepo(MyCustomLocalRepo())  
5     customCrashUploader(MyCustomCrashUploader())  
6     deviceStateRetriever(MyCustomDeviceStateRetriever())  
7     appName("some-custom-app-name")  
8     version(17, "0.1.4-debug")  
9     privacySettings {  
10         sendDeviceData(true)  
11         enableReporting(true)  
12     }  
13     timingSettings {  
14         delayAfterApplicationStartToUpload(10, TimeUnit.SECONDS)  
15         delayToRetryUpload(2, TimeUnit.MINUTES)  
16     }  
17     certificatePinner(  
18         CertificatePinner.Builder()  
19             .add("koarl.dev", "sha256/YLh1dUR9y6Kja30RrAn7JKnbQG/uEtLMkBgFF2Fuihg=")  
20             .build()  
21     )  
22 }
```

5.4.2 Back-end system

The back-end system is responsible to store and display the collected data which is sent by the Android library. The data is posted by the Android library to a REST service, which the back-end application provides. This component manages API requests and stores the data in an SQL database. In order to support SSL connection backed by a well established and tested implementation, all HTTP requests are passed through a reverse proxy provided by *NGINX* [25], which is responsible for SSL connection handling. A reverse proxy is a service which acts as a proxy for incoming connections to an application server, which in contrast to a client-side proxy does not need to be configured at the client. This allows that only the reverse proxy application is visible and directly accessible from the internet, which proxies the requests on to the real applications. This is usually done in order to have a separate, single server handling incoming connections and therefore shielding application servers from attacks performed on the HTTP stack level [24]. Furthermore, SSL handshakes are usually performed at a reverse proxy, where a strong, tested and secure implementation of the SSL protocol is implemented, and to take the load of handling SSL handshakes from the application servers.

As the system should be as easy to be set-up as possible, the system has been containerized using *docker*, which is a highly adopted and easily manageable containerization technology [23, 57, 93]. Furthermore, the back-end application also provides dashboard website, which allows developers to analyze the reported data. Furthermore, the back-end application also exposes a data source to be loaded within a *Grafana* application, in order to be able to show statistics about an applications stability.

As declared in section 4.4, the back-end system should require as little effort as possible to be set up.

By using *docker compose*, the handling of multiple docker images can be eased with a single configuration file. This is necessary, as the back-end application requires an SQL database to be available to store the data in. Furthermore, the reverse proxy as well as the optional Grafana dashboard are separate docker images to handle.

As shown in figure 7, the back-end service consists of several components, each of them being a separate application. By using *docker compose*, a “tool for defining and running multi-container Docker applications” [22], the different docker container can be configured and started easily. The usage of docker-compose leverages the easy set-up for multiple containers as well as the communication between them. The configuration is done in a *YAML* file, where a default configuration is provided to be used, which can be seen in listing B.1 in appendix B, where all configuration is already done and only the passwords need to be set to new values. The configuration can be adapted, but can be used in a production setting as it is. If a different SQL based database should be used, the respective service in the configuration needs to be modified, as well as the environment variable for the back-end application. Once the configuration is set up, the service can be started with only one command as shown in listing 5.8 after installing *docker* [21] as well as *docker compose* [20] on the host.

Listing 5.8: Single command needed to start the back-end service

```
1 docker-compose up -d
```

Back-end application & dashboard

The back-end application is the main part of the back-end, responsible for exposing a REST API for the Android library, persisting the data in the SQL database as well as serving the dashboard website as well as the *Grafana* data source. As stated in section 5.1, both the Android library as well as the back-end application have been developed using the programming language *Kotlin* in order to leverage serialization logic and model sharing across both applications. Furthermore, by developing the back-end application in Kotlin, no further programming language needs to be introduced. The *Ktor* framework has been used, as it leverages many Kotlin features, but does not introduce a new network stack, but can be used with different already established networking stacks such as the *Jetty* or *Netty* web server [81, 82].

For the API, a REST architecture with JSON as content type has been chosen. This is a very common set-up for communications for mobile applications. *XML* as an alternative content type introduces data overhead, and was therefore dismissed [66]. Moreover, *GraphQL* as a more flexible communication protocol than REST services was analyzed, but dismissed, as the advantages such as the flexible response bodies was not used, and therefore would not provide advantages over simple REST services [46].

The API stores the data in a provided SQL database. Due to the usage of the *Exposed SQL Framework* [54] from JetBrains, an object-relational mapping (ORM) tool. This enables

the application itself to run against a typed, domain specific class schema, whereas the ORM will translate the transactions into SQL statements for different SQL databases, including *H2*, *MySQL*, *Oracle*, *PostgreSQL* and *SQLite* [54]. In the provided configuration a *MySQL* database is used. If another database should be used, the respective `database.type` in line 13 of the configuration in listing B.1 needs to be adapted to the respective database type, as well as the connection parameters need to be adapted for the different database.

To enable developers to analyze the error and crash reports, a dashboard has been integrated into the back-end application in *Ktor*. The Ktor framework provides a Domain-Specific Language (DSL) to build HTML websites, with which web pages with dynamic content can be created [83]. The dashboard was developed to be simplistic, and is serving HTML pages with the content, which can be seen in figures 8 and 9. Besides that, all dashboard content is also available via a custom REST API, which enables the data to be used by other applications as well. Figure 10 shows the overview of the provided documentation page for the REST API.

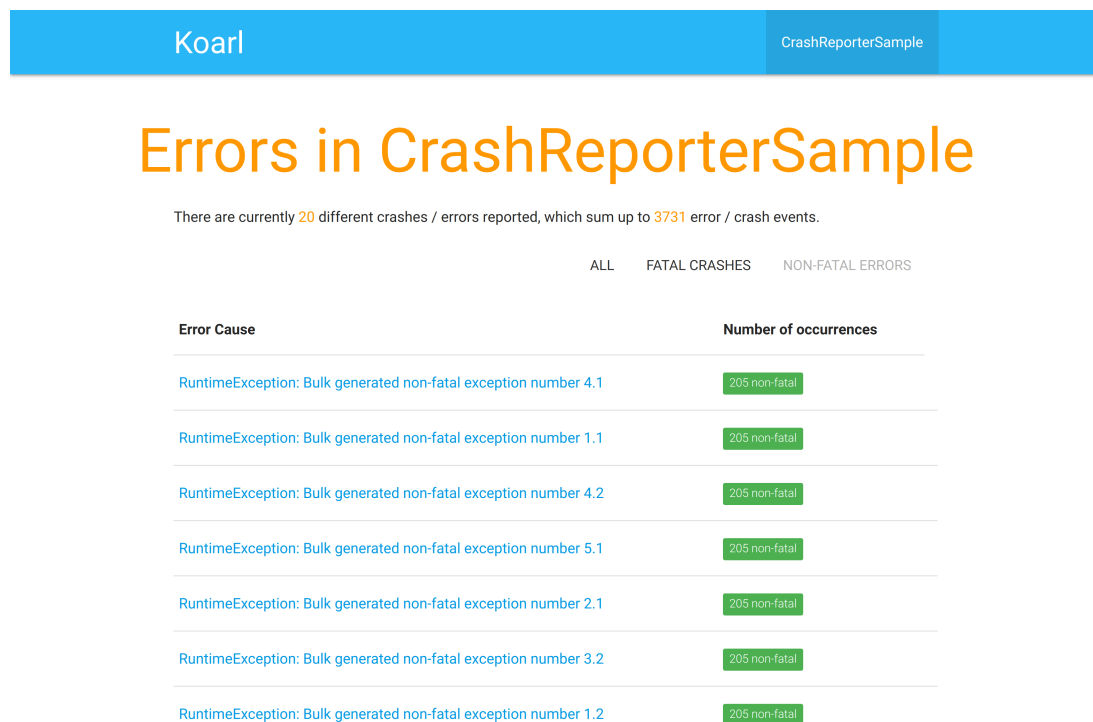


Figure 8: Screenshot of the list of non-fatal crashes in the dashboard for a given application

Back-end functionality

The back-end does not only store the reported data, but processes it to be able to display it in groups of similar errors and crashes. Errors are considered similar, if they share the same exception and stack trace, including their causes. This enables developers to easily detect how often a specific error occurred, and therefore prioritize the analysis and bug fixing according to that.

CrashReporterSample

Stack-Trace:

```

RuntimeException: Bulk generated non-fatal exception number 4.1
    at dev.moetz.koarl.android.sample.MainActivity$onCreate$3$threads$1$1.invoke(41)
    at dev.moetz.koarl.android.sample.MainActivity$onCreate$3$threads$1$1.invoke(9)
    at kotlin.concurrent.ThreadsKt$thread$thread$1.run(30)
    
```

Sessions / Crashes:

< 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 ... 21 >

Timestamp	OS version	Device name	Device brand	App version
2020-07-06T00:21:52.705Z	30	walleye	google	0.0.9 (1)
2020-07-06T00:21:49.123Z	29	walleye	google	0.0.9 (1)
2020-07-06T00:21:54.294Z	30	walleye	google	0.0.9 (1)
2020-07-06T00:21:54.800Z	29	walleye	google	0.0.9 (1)
2020-07-06T00:21:51.581Z	30	walleye	google	0.0.9 (1)

Figure 9: Screenshot of a crash-group, its stack trace and the list of occurrences in the dashboard

Furthermore, Android applications often are – and should be – obfuscated, which means that their application code should not contain real class-, member- and method names, but are replaced with random names [6]. This does not only shrink the final application size, but also makes it harder to reverse engineer the application [72]. However, as all method- and class names are replaced by meaningless names [68], the developers would not gain any helpful information of the stack trace, where all names of the classes and methods have been replaced with different names. Therefore, it is required by the back-end service to be able to show the original class- and method names. To do so, the obfuscation process of *ProGuard* (and its successor, *R8* [76]) produces a file, which contains the mapping of the original values and the replaced values of the package, class and method names. Furthermore, *ReTrace*, a tool to translate an obfuscated stack-trace with its original values is provided to be able to de-obfuscate stack-traces [6, 69]. This tool has been implemented in the back-end, and once the mapping-file for an application is uploaded in the dashboard, all stack-traces for this application will automatically be de-obfuscated and therefore analyzable by the developer.

Grafana

The back-end system was designed to be able to be included in already existing infrastructure and systems. Therefore, an integration into a widely used, open source monitoring and dashboard tool was considered. *Grafana* is, according to a 2019 survey [60], the most used metric monitoring tool used by *DevOps* engineers. Therefore, the back-end exposes a *Grafana* data source in order to be able to use the data collected in the back-end in *Grafana* dashboard [79]. This data includes the number of crashes for applications or specific versions of applications.

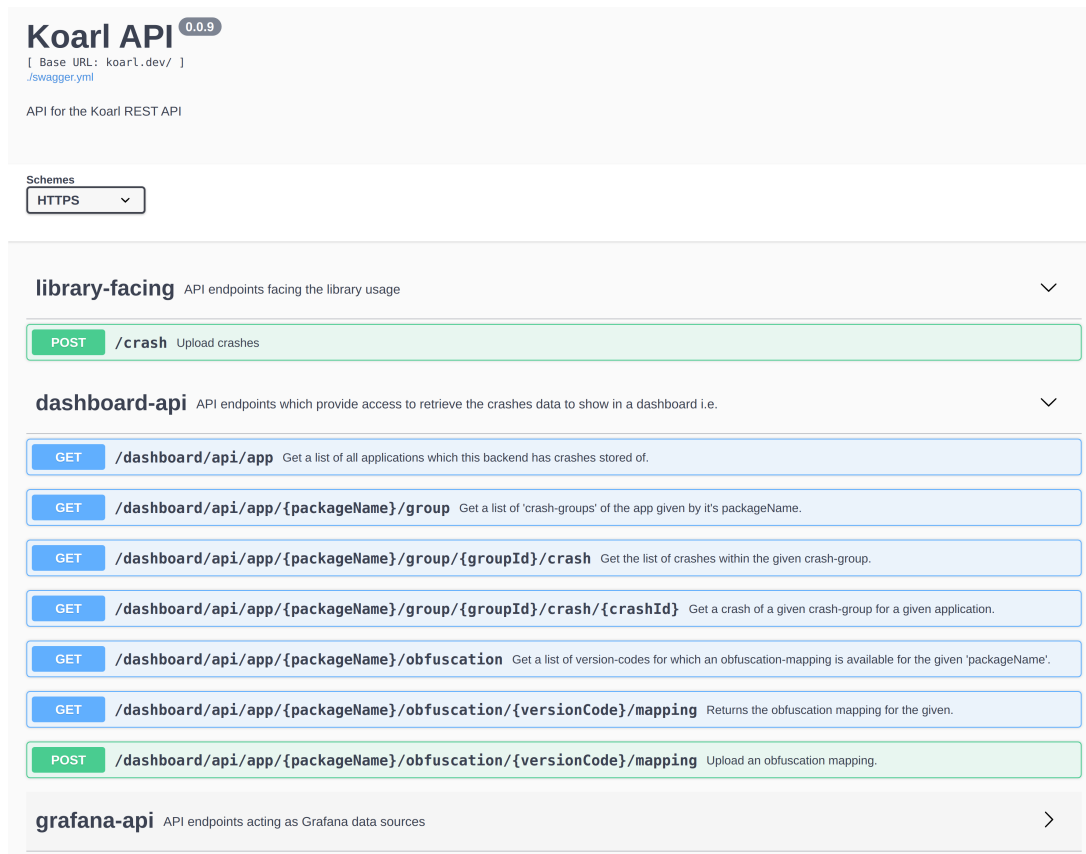


Figure 10: Screenshot of the provided API documentation to query the dashboard data

Many already available solutions, such as the systems introduced in section 2.2, offer additional metrics, which include analytical data, which this library in terms of user privacy is not collecting. Furthermore, nearly all existing solutions offer a metric called *Daily Active Users* or *Monthly Active Users*. However, collecting this metric would require the library to send so-called heartbeat events on every application start, and the back-end server would need to be able to identify multiple requests originating from one client, and therefore would need to introduce some identification of one device across multiple requests. As this introducing such an identifiable id in the system would be in contrast with the privacy by design as stated in section 1.1, this metric has not been made available. The data source is enabled by default, but can be disabled in the back-end. Moreover, a *Grafana* dashboard instance is included in the *docker compose* configuration, which can be disabled by removing the respective configuration.

Database

As a default database in the *docker compose* setup, a *MySQL* service is used. As mentioned above, this can be exchanged with any SQL database provided by *Exposed* [54]. In an earlier stage, *MongoDB* as a *NoSQL* database has been used, as a document-based *NoSQL* database seemed to fit the use-case of many writes with the same data structure perfectly.

However, the implementation using *MongoDB* showed very high latency of several seconds at accumulated read-requests, which were needed for the queries performed for the dashboard. While there was the possibility to take those delays in the dashboard into account, or implement some kind of caching logic for the dashboard, a switch to relational, SQL based databases has been done. While the read-queries for the dashboard showed no significant delay when using *MySQL*, the write-queries, which are in this back-end the performance-critical ones, showed no delays.

The database is run as a separate docker container, which started up and connected to the back-end service using *docker compose*, and is configured by environment variables. In that way, the database can be run independent to the back-end service, and can be replaced by any supported SQL database.

Reverse Proxy

An *NGINX* reverse proxy is part of the containers within the *docker compose* setup. This reverse proxy was added due as the communication to the back-end should be done in a secure manner. Therefore, HTTPS should be used, preferably with certificate pinning. This could be implemented within the Kotlin back-end server and its web stack, but then the application would already be part of the application that is facing towards the internet. Moreover, as with the optional Grafana dashboard there are multiple systems involved to be accessible from outside, the reverse proxy acts as a facade for those services. Therefore, hiding the services behind a reverse proxy contains the SSL termination be administrated and performed at one place, the reverse proxy. Furthermore, the separate systems behind the proxy do not need to care about SSL handling at all, as all connections behind the proxy will be performed using HTTP. There are multiple technologies available to be used as a reverse proxy, the two most often used ones are *Apache* and *NGINX*. *Apache* is still the most used web-server by the number of sites running on *Apache* with a decreasing share of 36.1% of all scanned sites by October 2020, *NGINX* currently serves 32.5% of the scanned sites by *W3Techs* [91]. Due to the simpler configuration of the *NGINX* reverse proxy as well as its increasing share of usage, *NGINX* has been selected.

An additional advantage in having the SSL termination externally from the back-end service is, that if this back-end service is to be added into an already existing server set-up, which already contains SSL handling as well as a reverse proxy, the *NGINX* docker container can just be omitted in the set-up and the already existing solution can be used without any additional configuration being needed in the back-end application.

In order to be able to server content using HTTPS, an SSL certificate is needed. While they can be purchased, with the use of Let's Encrypt certificates can be safely and without any costs be requested automatically as well as renewed automatically [49]. This certificate is then used by the *NGINX* reverse proxy to be able to establish HTTPS connections.

6 Reflection

In this section a reflection of the state of the art systems analyzed in section 2.2 will be performed. In section 4 it was defined how a privacy-aware and easy to use crash reporting system should be designed, where the requirements have been applied to already existing systems in section 2.2, which will be reflected in section 6.1 below. As none of the analyzed systems met all requirements, a new system has been developed, which will be discussed in section 6.2. Furthermore, the applicability of the proposed solution will be reflected.

6.1 Applicable, existing systems

In section 2.2, the state of the art in regards to already existing solutions has been stated. While Google Firebase, Android Vitals and Sentry were easy to set up, they all only provided hosted solutions with closed source back-ends, where the data storage techniques as well as the absence of data analysis for other purposes than the crash reporting for the eligible developers could not be verified. ACRA was the only analyzed solution which provided a self-hosted, open source back-end without data sharing to other systems, however, the set-up of this system is not straightforward and might lead to developer not choosing this privacy aware option due to the larger effort to set up.

Due to privacy considerations as mentioned in section 1.1 as well as section 3.1, a solution which does not share any data with external companies should be preferred. Furthermore, open source solutions, especially when the back-end implementation can be analyzed and inspected should be favored over closed-source back-end solutions, which additionally often are not able to be hosted on premise, whereby data sharing cannot be controlled at all.

Of the analyzed systems, only one, ACRA, could be proven to not accumulate user data and to not share data with third party entities and therefore respect the users privacy. The other systems may or may not share data, but this could not be verified due to closed source applications and missing control of the back-end application as well as non-clear statements in their privacy statements.

6.2 Applicability of the developed system

A new system which aims to meet all set requirements has been developed and published open source on GitHub: github.com/FlowMo7/Koarl [67]. The developed system meets all requirements stated in section 4:

- The system is free to use and does not introduce any licensing cost at all, which satisfies the requirement to be free to use in section 4.1.
- It is published with the *Apache 2.0* license, which enables freedom of use, modification and distribution of the system [26]. Additionally, the system and its source-code is available on GitHub [67], and can be inspected, build and extended by everyone, which satisfies the requirement for the system to be open source as stated in section 4.2.
- The back-end system is available to be self-hosted by using docker as containerization technology, which enables quick and easy set-up, which satisfies the requirement in section 4.3.
- The effort to set the system up includes the deployment of the back-end system using docker, which can be easily done, and the initialization of the library in the Android application with the URL to the back-end application. While there are systems available with less configuration to be performed in the Android application, for systems being able to be self-hosted, this is the minimum required effort to be able to initialize a self-hosted back-end application, which should satisfy the requirement to be easy to set up in section 4.4.
- Furthermore, the Android library provides different configuration options as shown in listing 5.7, and the back-end system can be configured in various ways, such as defining the SQL database to be used, whether Grafana should be included, and the option to expose a REST API, the data can be used in existing monitoring applications as well, which satisfies the requirement to be configurable as stated in section 4.5.
- The usage of the Android library does not introduce any restrictions, and declares a minimum SDK version of 10, which is lower than any competing libraries, and therefore satisfies the requirement to be non-restrictive as defined in section 4.6.

Besides the fact that all set requirements are set, the developed system also contains negative aspects. Due to the fact that the system and data collection has been integrated with privacy by design in mind, some features, such as an application version adoption rate or a daily active users or number of crash-free sessions have not been implemented. Such metrics would have required to link multiple crashes together to one specific, identifiable device, or would require the library to report the event of opening and closing the app, which would expose the user behavior in the data, and would require an identifier which identifies those events of one user uniquely. Furthermore, in comparison with already available systems, the provided back-end website is minimal, while the possibility to export the data using the *Grafana* data source or the REST API enables to integrate the data into possibly already existing monitoring solutions.

7 Future work

While the existing solutions have been analyzed on their applicability from a developers point of view, their shared data and library implementations have not been analyzed due to the source-code often not being public, and the data traffic being encrypted. Further analysis on the data collected and sent to the back-end applications of the existing systems would be admirable in order to gain more insights of which potentially sensitive data may be shared. Furthermore, the implications of their data usage in regards to applicable laws, such as the GDPR, and their impact on the developer as well as the application user should be analyzed. While the developed solution is ready to be used, it is still in active development, to i.e. provide a configuration which automatically handles TLS certificates using *Let's Encrypt* as well. In terms of the developed solution, the dashboard should be refactored to provide a better usability for the developers analyzing the crashes. Furthermore, the system could be adapted to provide the functionality to collect and report errors and crashes for iOS applications as well. By using KMP, components of the already existing Android library could be reused.

A Statistics

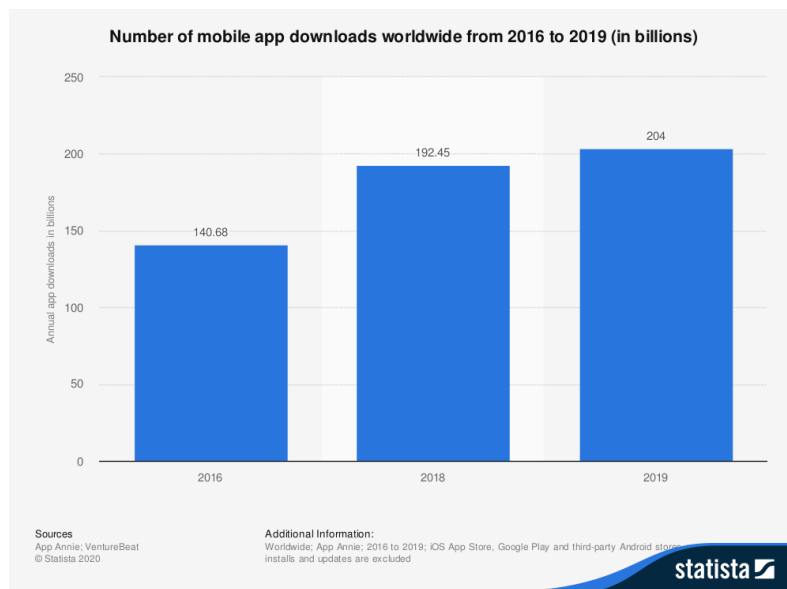


Figure 11: Number of mobile app downloads worldwide from 2016 to 2019 [90]

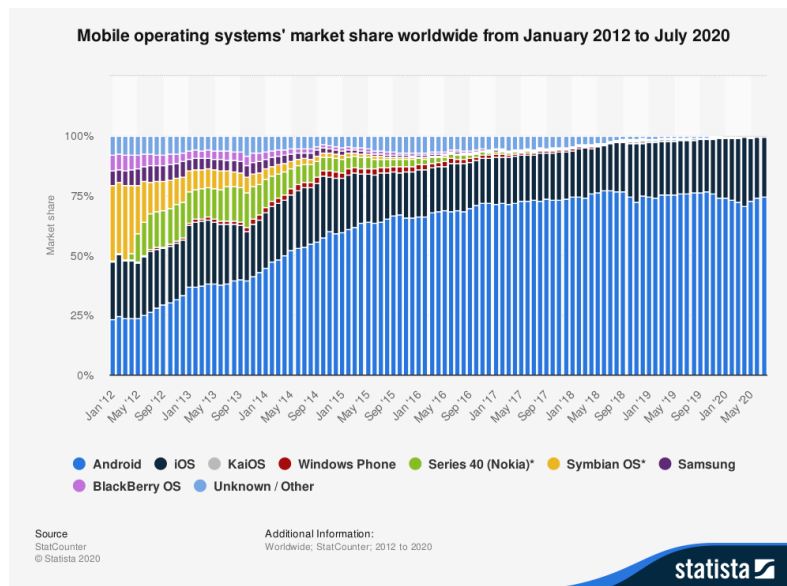


Figure 12: Mobile operating systems' market share worldwide from January 2012 to July 2020 [85]

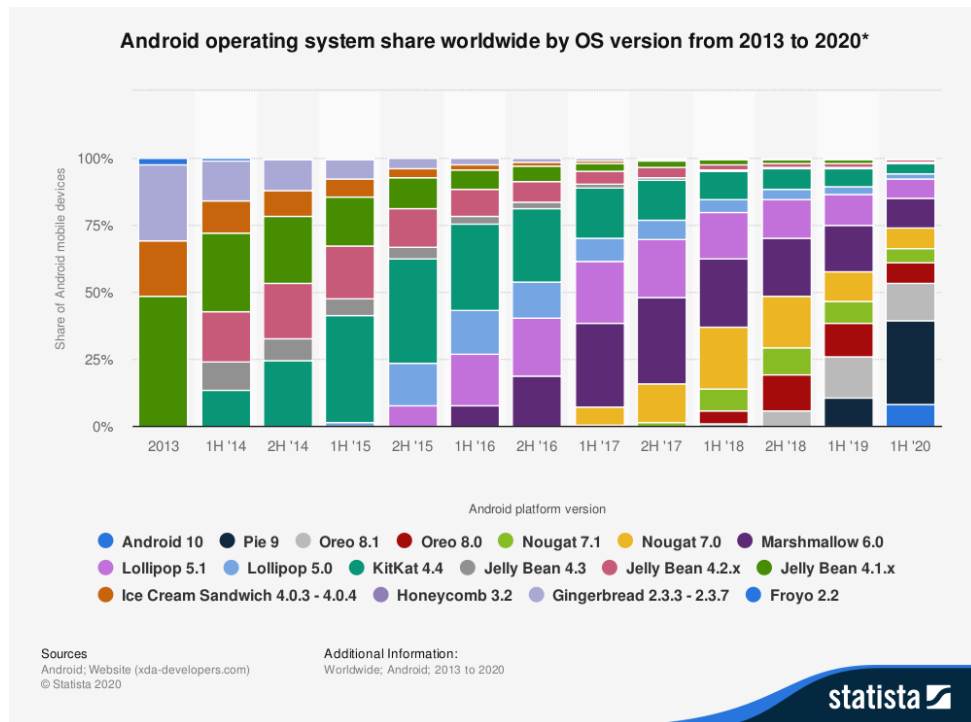


Figure 13: Android operating system share worldwide by OS version from 2013 to 2020 [12]

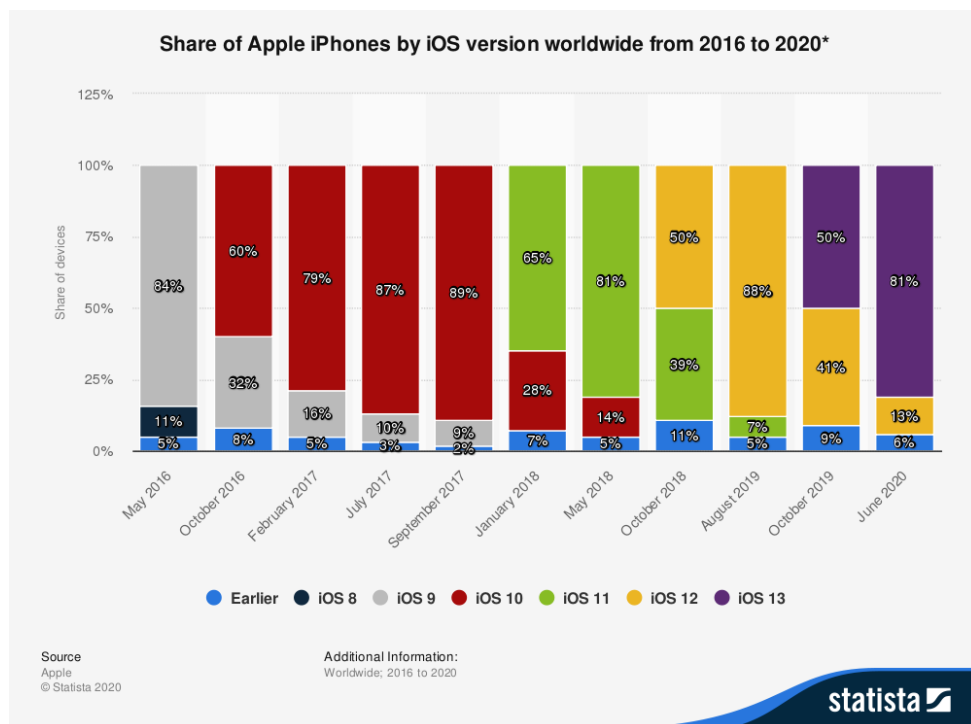


Figure 14: Share of Apple iPhones by iOS version worldwide from 2016 to 2020 [13]

B Koarl Configuration & Payload

Listing B.1: Provided docker-compose.yml file, which contains the default configuration needed to install and start the back-end service

```
1 version: "3"
2 services:
3   backend:
4     image: "koarl/koarl-backend:latest"
5     restart: "unless-stopped"
6     networks:
7       koarl_network:
8         aliases:
9           - backend
10    depends_on:
11      - database
12    environment:
13      - database.type=mysql
14      - database.host=database
15      - database.port=3306
16      - database.database_name=koarl
17      - database.user=koarl-database-user
18      - database.password=[GENERATED_DATABASE_PASSWORD_HERE]
19      - grafana.basicauth.user=grafana-koarl-backend-user
20      - grafana.basicauth.password=[GENERATED_GRAFANA_BASIC_AUTH_PASSWORD_HERE]
21      - swagger.enable=true
22
23   database:
24     image: "mysql:8"
25     restart: "unless-stopped"
26     environment:
27       - MYSQL_DATABASE=koarl
28       - MYSQL_USER=koarl-database-user
29       - MYSQL_PASSWORD=[GENERATED_DATABASE_PASSWORD_HERE]
30       - MYSQL_ROOT_PASSWORD=[DEFINE_ROOT_PASSWORD_HERE]
31     networks:
32       koarl_network:
33         aliases:
34           - database
35     volumes:
36       - ./database:/var/lib/mysql:rw
37   nginx:
38     container_name: "koarl"
39     hostname: "koarl"
40     image: "nginx:stable"
41     restart: always
42     ports:
43       - "80:80"
44       - "443:443"
45     networks:
46       - koarl_network
47     depends_on:
48       - backend
49     volumes:
50       - ./proxy/config:/etc/nginx/conf.d/:ro
51       - ./logs:/logs:rw
```

```

52  # Optional:
53  grafana:
54      image: "grafana/grafana:latest"
55      container_name: "koarl-grafana"
56      hostname: "koarl-grafana"
57      restart: always
58      networks:
59          - koarl_network
60      depends_on:
61          - backend
62      volumes:
63          - grafana:/var/lib/grafana:rw
64      environment:
65          - GF_SERVER_DOMAIN=koarl.example.com
66          - GF_SERVER_ROOT_URL=%(protocol)s://%(domain)s:%(http_port)s/grafana/
67          - GF_SERVER_SERVE_FROM_SUB_PATH=true
68          - GF_SECURITY_ADMIN_PASSWORD=[GENERATED_GRAFANA_ADMIN_PASSWORD_HERE]
69          - GF_INSTALL_PLUGINS=grafana-clock-panel,grafana-simple-json-datasource
70
71  volumes:
72      grafana:
73
74  networks:
75      koarl_network:
76          driver: bridge

```

Listing B.2: Example JSON payload sent from the Android library to the back-end server

```
1 {
2   "deviceData": {
3     "deviceName": "generic_x86",
4     "manufacturer": "Google",
5     "brand": "google",
6     "model": "Android SDK built for x86",
7     "buildId": "OSM1.180201.007",
8     "operationSystemVersion": 28
9   },
10  "appData": {
11    "packageName": "dev.moetz.koarl.sample",
12    "appName": "Koarl Sample Application",
13    "appVersionCode": 237,
14    "appVersionName": "1.6.17"
15  },
16  "crashes": [
17    {
18      "uuid": "c70d5137-0856-4695-bc44-607289e0135b",
19      "isFatal": true,
20      "dateTime": "2020-11-29T024300.497UTC",
21      "throwable": {
22        "name": "IllegalStateException",
23        "message": "Something, somewhere, went terribly wrong",
24        "localizedMessage": "Something, somewhere, went terribly wrong",
25        "stackTrace": [
26          {
27            "fileName": "MainActivity.kt",
28            "lineNumber": 17,
29            "className": "MainActivity",
30            "methodName": "onCreate",
31            "isNativeMethod": false
32          }
33        ],
34        "cause": {
35          "name": "IllegalStateException",
36          "message": "Something, somewhere, went terribly wrong",
37          "localizedMessage": "Something, somewhere, went terribly wrong",
38          "stackTrace": [
39            {
40              "fileName": "MainActivity.kt",
41              "lineNumber": 17,
42              "className": "MainActivity",
43              "methodName": "onCreate",
44              "isNativeMethod": false
45            }
46          ]
47        }
48      }
49    ]
50  }
51 }
```

Bibliography

- [1] A. Cavoukian. Privacy by Design - The 7 Foundational Principles. <https://www.ipc.on.ca/wp-content/uploads/Resources/7foundationalprinciples.pdf>, January 2011. Accessed: 01.03.2020.
- [2] ACRA Contributors. Basic Setup - ACRA: Application Cras Reports for Android. <https://github.com/ACRA/acra/wiki/BasicSetup>. Accessed: 06.10.2020.
- [3] ACRA Contributors. Interactions - ACRA: Application Cras Reports for Android. <https://github.com/ACRA/acra/wiki/interactions>. Accessed: 13.10.2020.
- [4] ACRA Contributors. Report Destinations - ACRA: Application Cras Reports for Android. <https://github.com/ACRA/acra/wiki/report-destinations>. Accessed: 13.10.2020.
- [5] B. F. Andrés and M. Pérez. Transpiler-based architecture for multi-platform web applications. In *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6, 2017.
- [6] Android Developer. Shrink, obfuscate, and optimize your app. <https://developer.android.com/studio/build/shrink-code>. Accessed: 06.10.2020.
- [7] Android Developers. Detect and diagnose crashes. <https://developer.android.com/games/optimize/crash>. Accessed: 27.09.2020.
- [8] Android Developers. Room Persistence Library. <https://developer.android.com/topic/libraries/architecture/room>. Accessed: 27.09.2020.
- [9] Android Developers. Support library. <https://developer.android.com/topic/libraries/support-library>. Accessed: 14.10.2020.
- [10] Android Developers. Android vitals. <https://developers.android.com/topic/performance/vitals>, 2020. Accessed: 01.03.2020.
- [11] Android Developers. Configure your build. <https://developer.android.com/studio/build>, 2020. Accessed: 28.09.2020.
- [12] Android, Website (xda-developers.com). Android operating system share worldwide by OS version from 2013 to 2020. *Statista*. <https://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/>, April 2020. Accessed: 25.11.2020.

- [13] Apple. Share of Apple iPhones by iOS version worldwide from 2016 to 2020. *Statista*. <https://www.statista.com/statistics/565270/apple-devices-ios-version-share-worldwide/>, June 2020. Accessed: 25.11.2020.
- [14] A. Belkhir, M. Abdellatif, R. Tighilt, N. Moha, Y. G. Guéhéneuc, and É. Beaudry. An observational study on the state of rest api uses in android mobile applications. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 66–75, 2019.
- [15] M. Bhor and D. Karia. Certificate pinning for android applications. In *2017 International Conference on Inventive Systems and Control (ICISC)*, pages 1–4, 2017.
- [16] C. Haase. Google I/O 2019: Empowering developers to build the best experiences on Android + Play. <https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html>, 2019. Accessed: 22.09.2020.
- [17] M. Chibba and A. Cavoukian. Privacy, consumer trust and big data: Privacy by design and the 3 c's. In *2015 ITU Kaleidoscope: Trust in the Information Society (K-2015)*, pages 1–5, Dec 2015.
- [18] Council of European Union. Council regulation (EU) no 2016/679. <https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679&from=EN>, April 2016. Accessed: 11.10.2020.
- [19] Docker Inc. Docker frequently asked questions (FAQ). <https://docs.docker.com/engine/faq/>. Accessed: 27.09.2020.
- [20] Docker Inc. Install Docker Compose. <https://docs.docker.com/compose/install/>. Accessed: 05.10.2020.
- [21] Docker Inc. Install Docker Engine. <https://docs.docker.com/engine/install/>. Accessed: 05.10.2020.
- [22] Docker Inc. Overview of Docker Compose. <https://docs.docker.com/compose/>. Accessed: 05.10.2020.
- [23] Docker Inc. Empowering App Development for Developers | Docker. <https://www.docker.com/>, 2020. Accessed: 27.09.2020.
- [24] H. Dong, J. Wang, and J. Yang. Effective request distributing in distributed ssl reverse proxies. In *2016 International Symposium on Computer, Consumer and Control (IS3C)*, pages 412–416, 2016.
- [25] Inc F5. NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy. <https://www.nginx.com/>. Accessed: 29.11.2020.

- [26] Apache Software Foundation. Apache License, Version 2.0. <https://www.apache.org/licenses/LICENSE-2.0>, January 2004. Accessed: 14.10.2020.
- [27] Roberto Gallo, Henrique Kawakami, Ricardo Dahab, Luiz Navarro, Patricia Hongo, Glaucio Junquera, Kaio Karam, and Luander Ribeiro. Security and system architecture: comparison of android customizations. In *8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 06 2015.
- [28] M. Gomez, R. Rouvoy, B. Adams, and L. Seinturier. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 88–99. IEEE, 2016.
- [29] Google. Firebase Crashlytics and Firebase App Distribution Terms of Service. <https://firebase.google.com/terms/crashlytics>, September 2019. Accessed: 17.09.2020.
- [30] Google. Privacy and Security in Firebase. <https://firebase.google.com/terms/crashlytics>, September 2019. Accessed: 17.09.2020.
- [31] Google. How to use the Play Console. <https://support.google.com/googleplay/android-developer/answer/6112435>, 2020. Accessed: 17.09.2020.
- [32] Google. Monitor your app’s technical performance with Android vitals. <https://support.google.com/googleplay/android-developer/answer/7385505>, 2020. Accessed: 17.09.2020.
- [33] Inc. Google. Android – Certified. https://www.android.com/intl/en_in/certified/. Accessed: 11.10.2020.
- [34] Google Developers. Alternative distribution options. <https://developer.android.com/distribute/marketing-tools/alternative-distribution>. Accessed: 23.09.2020.
- [35] Google Developers. Firebase Crashlytics and Firebase App Distribution Terms of Service. <https://firebase.google.com/docs/android/setup>. Accessed: 05.10.2020.
- [36] Google Developers. App startup time. <https://developer.android.com/topic/performance/vitals/launch-time>, 2020. Accessed: 03.10.2020.
- [37] Google Developers. Firebase. <https://firebase.google.com/>, 2020. Accessed: 26.03.2020.
- [38] Google Developers. Firebase Crashlytics. <https://firebase.google.com/docs/crashlytics>, 2020. Accessed: 01.03.2020.
- [39] Google Developers. Firebase Pricing. <https://firebase.google.com/pricing>, 2020. Accessed: 01.03.2020.

- [40] Google, Inc. Android 10 Compatibility Definition. <https://source.android.com/compatibility/10/android-10-cdd>. Accessed: 02.03.2020.
- [41] Google, Inc. Distribution Dashboard. <https://developer.android.com/about/dashboards/index.html>. Accessed: 08.03.2020.
- [42] Google, Inc. Support different platform versions. <https://developer.android.com/training/basics/supporting-devices/platforms>. Accessed: 08.03.2020.
- [43] Google Inc. privacy Policy - Privacy & Terms. <https://policies.google.com/privacy>, 2020. Accessed: 05.10.2020.
- [44] Gradle Inc. Declaring dependencies. https://docs.gradle.org/current/userguide/declaring_dependencies.html, 2020. Accessed: 28.09.2020.
- [45] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia. Understanding android fragmentation with topic analysis of vendor-specific bugs. In *2012 19th Working Conference on Reverse Engineering*, pages 83–92, 2012.
- [46] D. A. Hartina, A. Lawi, and B. L. E. Panggabean. Performance analysis of graphql and restful in sim lp2m of the hasanuddin university. In *2018 2nd East Indonesia Conference on Computer and Information Technology (EIConCIT)*, pages 237–240, 2018.
- [47] I. Malchev. Faster Adoption with Project Treble. <https://android-developers.googleblog.com/2018/05/faster-adoption-with-project-treble.html>, May 2018. Accessed: 08.03.2020.
- [48] A. K. Iannillo, R. Natella, D. Cotroneo, and C. Nita-Rotaru. Chizpurple: A gray-box android fuzzer for vendor service customizations. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–11, 2017.
- [49] Internet Security Research Group (ISRG). How it Works - Let's Encrypt. <https://letsencrypt.org/how-it-works/>. Accessed: 05.10.2020.
- [50] J. Isaak and M. J. Hanna. User data privacy: Facebook, cambridge analytica, and privacy protection. *Computer*, 51(8):56–59, August 2018.
- [51] D. Jemerov. Why JetBrains needs Kotlin. <https://blog.jetbrains.com/kotlin/2011/08/why-jetbrains-needs-kotlin/>, August 2011. Accessed: 25.03.2020.
- [52] JetBrains. Type-Safe Builders. <https://kotlinlang.org/docs/reference/type-safe-builders.html>. Accessed: 27.09.2020.
- [53] JetBrains. Multiplatform programming. <https://kotlinlang.org/docs/reference/multiplatform.html>, 2020. Accessed: 22.09.2020.
- [54] JetBrains and Kotlin Exposed Contributors. JetBrains/Exposed: Kotlin SQL Framework. <https://github.com/JetBrains/Exposed>, 2020. Accessed: 04.10.2020.

- [55] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 15–24, 2013.
- [56] A. Khanjani and R. Sulaiman. The aspects of choosing open source versus closed source. In *2011 IEEE Symposium on Computers Informatics*, pages 646–649, March 2011.
- [57] Á. Kovács. Comparison of different linux containers. In *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*, pages 47–51, 2017.
- [58] P. Krill. JetBrains readies JVM-based language. <https://www.infoworld.com/article/2622405/jetbrains-readies-jvm-based-language.html>, July 2011. Accessed: 25.03.2020.
- [59] F. Lardinois. Kotlin is now Google’s preferred language for Android app development. <https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/>, May 2019. Accessed: 25.03.2020.
- [60] Logs.io. The DevOps Pulse - 2019. <https://logz.io/devops-pulse-2019/>. Accessed: 10.10.2020.
- [61] M. Mahmoudi and S. Nadi. The android update problem: An empirical study. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 220–230, May 2018.
- [62] B. Mayo. For the second time this year, the Facebook SDK is causing apps like TikTok and Spotify to crash on launch. <https://9to5mac.com/2020/07/10/app-crash-facebook-sdk/>, July 2020. Accessed: 27.09.2020.
- [63] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press, München, 2004.
- [64] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.
- [65] F. Mötz. Analysis of different data-storage technologies within android applications, 2018.
- [66] F. Mötz. Using graphql with android applications, 2018.
- [67] F. Mötz. Koarl. <https://github.com/FlowMo7/Koarl>, 2020. Accessed: 30.11.2020.
- [68] Guardsquare nv. ProGuard manual | Introduction. <https://www.guardsquare.com/en/products/proguard/manual/introduction>. Accessed: 06.10.2020.
- [69] Guardsquare nv. ProGuard manual | ReTrace. <https://www.guardsquare.com/en/products/proguard/manual/retrace>. Accessed: 06.10.2020.

- [70] State of California Department of Justice Office of the Attorney General. California Consumer Privacy Act (CCPA). <https://oag.ca.gov/privacy/ccpa>. Accessed: 11.10.2020.
- [71] Oracle. Thread.UncaughtExceptionHandler (Java Platform SE 7). <https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.UncaughtExceptionHandler.html>, 2020. Accessed: 27.09.2020.
- [72] J. Pan and S. Ma. Advertisement removal of android applications by reverse engineering. In *2017 International Conference on Computing, Networking and Communications (ICNC)*, pages 695–700, 2017.
- [73] R. Paret. Fabric is Joining Google. <https://fabric.io/blog/fabric-joins-google>, 2020. Accessed: 26.03.2020.
- [74] Qualitrix. What is the Acceptable Mobile App Crash Rate? <https://qualitrix.com/blog/app-crash-percentage/>, January 2019. Accessed: 01.03.2020.
- [75] A. Schröter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 118–121, 2010.
- [76] L. Sei. R8, the new code shrinker from Google, is available in Android studio 3.3 beta. <https://android-developers.googleblog.com/2018/11/r8-new-code-shrinker-from-google-is.html>, November 2018. Accessed: 06.10.2020.
- [77] Sentry. Plans and Pricing | Sentry. <https://sentry.io/pricing/>, 2020. Accessed: 19.09.2020.
- [78] Sentry. Privacy Policy. <https://sentry.io/privacy/>, November 2020. Accessed: 27.11.2020.
- [79] simpod. JSON plugin for Grafana. <https://grafana.com/grafana/plugins/simpod-json-datasource>. Accessed: 10.10.2020.
- [80] Square, Inc. CertificatePinner - OkHttp. <https://square.github.io/okhttp/4.x/okhttp/okhttp3/-certificate-pinner/>, 2020. Accessed: 03.10.2020.
- [81] JetBrains s.r.o. Custom Engines - Help | Ktor 1.4.0. <https://ktor.io/docs/advanced-engines.html>. Accessed: 06.10.2020.
- [82] JetBrains s.r.o. First App - Help | Ktor 1.4.0. <https://ktor.io/docs/guides-application.html>. Accessed: 06.10.2020.
- [83] JetBrains s.r.o. HTML DSL - Help | Ktor 1.4.0. <https://ktor.io/docs/html-dsl.html>. Accessed: 06.10.2020.
- [84] Stack Overflow. Stack Overflow Developer Survey 2019. <https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted>, April 2019. Accessed: 26.03.2020.

- [85] StatCounter. Number of mobile app downloads worldwide from 2016 to 2019. *Statista*. <https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/>, August 2020. Accessed: 25.11.2020.
- [86] J. Taplin. *Move Fast and Break Things: How Facebook, Google, and Amazon Have Cornered Culture and What It Means For All Of Us*. Pan Macmillan, 2017.
- [87] The Apache Software Foundation. Maven - Glossary. <https://maven.apache.org/glossary.html>, 2020. Accessed: 28.09.2020.
- [88] Threema. Threema Goes Open Source, Welcomes New Partner. <https://threema.ch/en/blog/posts/open-source-and-new-partner>. Accessed: 22.09.2020.
- [89] E. Thu and T. Aung. Developing mobile application framework by using restful web service with json parser. In *Genetic and Evolutionary Computing: Proceedings of the Ninth International Conference on Genetic and Evolutionary Computing*, volume 388, 08 2015.
- [90] App Annie; VentureBeat. Number of mobile app downloads worldwide from 2016 to 2019. *Statista*. <https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/>, January 2020. Accessed: 25.11.2020.
- [91] W3Techs. Comparison of the usage statistics of Apache vs. Nginx for websites. <https://w3techs.com/technologies/comparison/ws-apache,ws-nginx>, 2020. Accessed: 04.10.2020.
- [92] Y. Wang, Y. Y. Ng, H. Zhou, Y. Dong, and Z. Ji. The improvement of the trustworthiness of android app stores in china. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 357–360, 2016.
- [93] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu. Emerging trends, techniques and open issues of containerization: A review. *IEEE Access*, 7:152443–152472, 2019.
- [94] J. K. Waters. Kotlin Goes Open Source. <https://adtmag.com/articles/2012/02/22/kotlin-goes-open-source.aspx>, February 2012. Accessed: 25.03.2020.
- [95] L. Wei, Y. Liu, and S. Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 226–237, 2016.
- [96] B. Whitley. Kotlin Multiplatform for iOS Developers. <https://www.infoq.com/articles/kotlin-multiplatform-ios-developers/>, April 2019. Accessed: 25.03.2020.

List of Figures

Figure 1	Dashboard of Firebase Crashlytics, which lists the top application crashes from the last seven days. <i>Screenshot from console.firebase.google.com</i>	10
Figure 2	Set-up instructions from the Firebase console. <i>Screenshot from console.firebase.google.com</i>	11
Figure 3	Crashes and ANRs listed in the Google Play Console. <i>Screenshot from play.google.com/console</i>	13
Figure 4	Dashboard of <i>Acrarium</i> , a back-end and dashboard service for ACRA. <i>Screenshot from a self-hosted instance of Acrarium.</i>	16
Figure 5	Dashboard of Sentry, which lists the top application crashes from the last 14 days. <i>Screenshot from sentry.io</i>	18
Figure 6	Diagram of the architecture of the systems involved	28
Figure 7	Schematic diagram of the developed and used components for the Android library and the back-end service.	31
Figure 8	Screenshot of the list of non-fatal crashes in the dashboard for a given application	40
Figure 9	Screenshot of a crash-group, its stack trace and the list of occurrences in the dashboard	41
Figure 10	Screenshot of the provided API documentation to query the dashboard data . .	42
Figure 11	Number of mobile app downloads worldwide from 2016 to 2019 [90]	47
Figure 12	Mobile operating systems' market share worldwide from January 2012 to July 2020 [85]	47
Figure 13	Android operating system share worldwide by OS version from 2013 to 2020 [12]	48
Figure 14	Share of Apple iPhones by iOS version worldwide from 2016 to 2020 [13] . . .	48

List of Tables

Table 1	The 7 Foundational Principles of Privacy by Design [1]	4
Table 2	Relative number of devices running a given version of the Android platform. Data collected during a 7-day period ending on May 7, 2019 by Google [41]. Any versions with less than 0.1% distribution are not shown.	5
Table 3	Summary of Google Firebase Crashlytics	12
Table 4	Summary of Android Vitals	15
Table 5	Summary of ACRA	17
Table 6	Summary of Sentry	19
Table 7	Summary of the comparison of the different existing solutions	20
Table 8	Comparison of the size of the APK and the compiled bytecode (DEX file) with the Android library either with OkHttp or with Ktor included, with and without minification.	34

List of Listings

Listing 2.1	Sample configuration of the ACRA library	16
Listing 5.1	Sample usage of registering an uncaught exception handler	32
Listing 5.2	Including the Android library (“Koarl” ¹) dependency into an Android application using Gradle	34
Listing 5.3	Initializing the Android library in the application at runtime	35
Listing 5.4	Defining a custom LocalRepo for the Android library	36
Listing 5.5	Defining a custom CrashUploader for the Android library	37
Listing 5.6	Defining a custom DeviceStateReceiver for the Android library	37
Listing 5.7	Android library initialization with all possible configuration options	38
Listing 5.8	Single command needed to start the back-end service	39
Listing B.1	Provided docker-compose.yml file, which contains the default configuration needed to install and start the back-end service	49
Listing B.2	Example JSON payload sent from the Android library to the back-end server .	51

List of Abbreviations

ANR	Application Not Responding
API	Application Programming Interface
CCPA	California Consumer Privacy Act
DEX	Dalvik Executable
DNS	Domain Name Server
DSL	Domain-Specific Language
GDPR	General Data Protection Regulation
IDE	Integrated Development Environment
JAR	Java Archive
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
JRE	Java Runtime Environment
KMP	Kotlin Multiplatform
ORM	object-relational mapping
REST	Representational state transfer
SDK	Software Development Kit
SQL	Structured Query Language
TLS	Transport Layer Security
UUID	Universally unique identifier