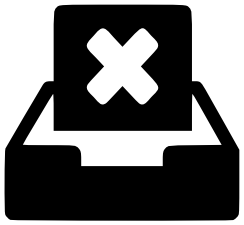
HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz



# Crashbin – Dokumentation

**Studienarbeit / Bachelorarbeit**

**Frühlingssemester 2019**

Abteilung Informatik
Hochschule für Technik Rapperswil (HSR)
`www.hsr.ch`

Author (BA): Florian Bruhin
Author (SA): Luca Tavernini

Betreuer: Prof. Stefan Richter, HSR

Datum: 14. Juni 2019

# Abstract

Automatisierte Crash-Reporting-Tools wie Sentry oder Airbrake werden von zehntausenden Software-Projekten und Firmen genutzt. Bestehende Produkte sind auf vollständige Automatisierung und oft Millionen von Crash-Events pro Monat ausgelegt. Keines der Produkte berücksichtigt die Notwendigkeit vereinfachter Kommunikation mit den Endbenutzern, die mit den Crashes konfrontiert wurden.

Opensource-Projekte sowie Projekte von Start-ups sind oft vergleichsweise klein und ziehen wenig Nutzen aus den zahlreichen Automatisierungen, die diese mächtigen Tools anbieten. Jedoch ist für diese Art von Projekten die direkte Kommunikation mit Benutzern wichtig, insbesondere wenn Probleme auftreten. Ohne entsprechende Werkzeuge besteht die Gefahr, dass Entwickler nicht von Problemen erfahren und Nutzer zu einer Alternative wechseln.

In einigen Opensource-Projekten kommunizieren Entwickler via Mail mit den von Software-Defekten betroffenen Benutzern, was schon bei mehreren Crash-Events pro Tag einen erheblichen Aufwand bedeutet. Oft werden auch Bug- und Ticket-Tracker für den Nutzer- und Kundensupport verwendet, diese sind jedoch nicht oder nur rudimentär mit den Crashreportern verknüpft.

In dieser Arbeit werden die Bedürfnisse von solchen kleineren Projekten genauer analysiert und es wird ein neuartiges Werkzeug präsentiert: Eine Mischung aus den Konzepten bestehender Crash-Reporter sowie Ticket-Tracking-Software, um die Kommunikation mit Nutzern zu erleichtern.

Automated crash reporting tools like Sentry or Airbrake are used by tens of thousands of software projects and companies. Existing products are designed for millions of crash reports per month and thus focused on a high level of automation. No existing solution allows for streamlined communication with the users affected by crashes.

Open source projects as well as projects run by start-ups are often relatively small and therefore don't benefit from the automation features offered by those tools. However, for such projects, direct communication with users is very important, especially if issues occur. Without corresponding tools, there is a risk that developers do not know about software defects, with users switching to an alternative instead.

In some open source projects, developers use emails to communicate with users after issues occur – an approach which scales up poorly, even with only a handful of reports per day. Often, bug trackers or ticket tracking tools are used for user support, but these are not integrated well with crash reporters.

In this thesis, the requirements of such smaller projects are analyzed in detail, and a new type of tool is presented: A mixture between concepts of existing crash reporters and ticket tracking software, to facilitate communication with users.

# 1. Management Summary

## 1.1. Ausgangslage

Diverse Praktiken für eine qualitativ hochwertige Software-Entwicklung sind heutzutage weit verbreitet. Trotzdem ist es oft unvermeidbar, dass durch Programmierfehler verursachte Probleme unerkannt zu Endkunden gelangen und dort Abstürze verursachen.

Gerade im Opensource-Umfeld vertrauen viele Projekte darauf, dass sich die Benutzer, bei denen Probleme auftreten, mit einem Fehlerbericht bei den Entwicklern melden. Dies erhöht jedoch den Aufwand sowohl für die Entwickler als auch die Nutzer: Letztere müssen den richtigen Ort für einen Bericht finden, und die aufgetretenen Probleme möglichst genau beschreiben. Gleichzeitig sind Entwickler oft mit Fehlerberichten konfrontiert, welche zu wenig konkrete Informationen enthalten, um den dahinterliegenden Fehler finden und beheben zu können. Es besteht die Gefahr, dass die von den Problemen betroffenen Nutzer lieber auf eine Alternative ausweichen und die Entwickler nicht von den Problemen erfahren.

Im Kontrast dazu stehen Werkzeuge zur automatisierten Verwaltung von Fehlerberichten bei Abstürzen, wie beispielsweise Sentry[1] oder Airbrake[2]. Diese sind jedoch darauf ausgelegt, automatisiert eine grosse Menge von Fehlern bzw. Events zu verarbeiten (typischerweise Tausende bis Millionen pro Monat). Keines dieser Produkte bietet die Möglichkeit, mit den Endbenutzern, bei denen Probleme aufgetreten sind, einfach in Kontakt zu treten.

Opensource-Projekte sowie Software von Start-ups bedienen oft einen vergleichsweise kleinen Nutzerkreis. Die zahlreichen Möglichkeiten zur Automatisierung, welche von bestehenden Werkzeugen angeboten werden, sind dort weniger relevant. Umso wichtiger ist der enge, proaktive und effiziente Kontakt zu den Nutzern, insbesondere nachdem Probleme aufgetreten sind.

Einige Opensource-Projekte versuchen diese Problematik zu lösen, indem sie in einem Crash-Reporter eine Mail verschicken oder dem Nutzer die nötige Information anzeigen, um manuell einen Fehlerbericht zu erstellen. Dies mag für Projekte mit wenigen Nutzern funktionieren, skaliert aber schon ab mehr als einem Report pro Tag kaum.

Ticket-Tracking-Systeme wie OTRS[3] oder Zendesk[4] bieten diverse Möglichkeiten für einfachen Kundensupport wie Textvorlagen oder Möglichkeiten zur Automatisierung, wobei trotz dieser Automatisierung die menschliche Interaktion zu den Kunden im Vordergrund steht.

Mit dieser Arbeit wollen wir die Ansätze von Crash-Reportern mit solchen von Ticket-Tracking-Systemen verbinden und so ein neuartiges Werkzeug schaffen. Dieses soll es ermöglichen, effizient mit betroffenen Benutzern zu kommunizieren.

[1] `https://sentry.io/`
[2] `https://airbrake.io/`
[3] `https://otrs.com/`
[4] `https://www.zendesk.com/`

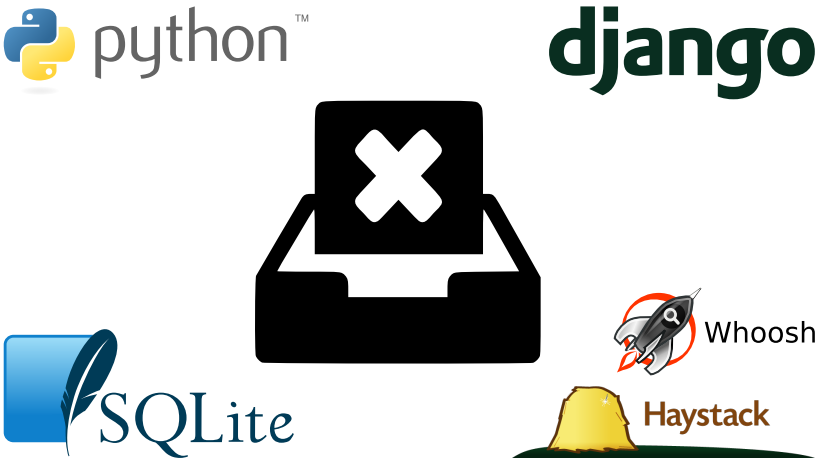



Abbildung 1.1.: Crashbin-Logo und verwendete Technologien

## 1.2. Vorgehen, Technologien

Da unser Tool plattformübergreifend funktionieren soll, haben wir eine Webapplikation entwickelt. Wir nutzen dazu Python mit dem Django-Webframework.

Trotz des Fokus auf manuelle Kommunikation soll unsere Software auch bei einem wachsenden Volumen an Fehlerberichten nutzbar sein. Um diese Flexibilität zu ermöglichen, lässt sie sich durch Plugins erweitern. Damit lässt sich das bestehende Tool durch eine schrittweise Automatisierung (wo nötig und sinnvoll) nahtlos weiter verwenden, wenn die Anzahl Nutzer oder Kunden wächst.

## 1.3. Ergebnisse

Neben einer Analyse des oben beschriebenen Business-Cases wurde als Teil dieser Arbeit ein neues Tool namens Crashbin entwickelt. Crashbin erlaubt es, die genannten Abläufe effizienter umzusetzen, als dies mit manueller Kommunikation möglich wäre.

Crashbin wurde als *Proof of Concept* implementiert und in ein bestehendes Opensource-Projekt integriert. Weiterhin werden Beispiele gezeigt, wie Crashbin in Projekte mit verschiedenen Programmiersprachen integriert werden kann.

## 1.4. Ausblick

Der Wunsch nach einem Tool wie Crashbin wuchs aus dem qutebrowser-Projekt, das von einem der Autoren dieser Arbeit (Florian Bruhin) unterhalten wird. Aufgrund der begrenzten Zeit konnten nicht alle Features implementiert werden, die für die Integration von qutebrowser benötigt werden.

Es ist geplant, das im Rahmen dieser Arbeit gestartete Projekt auch nach dem Studium weiterzuentwickeln, so dass es bald im Produktiveinsatz für qutebrowser und weitere interessierte Projekte nutzbar sein wird.

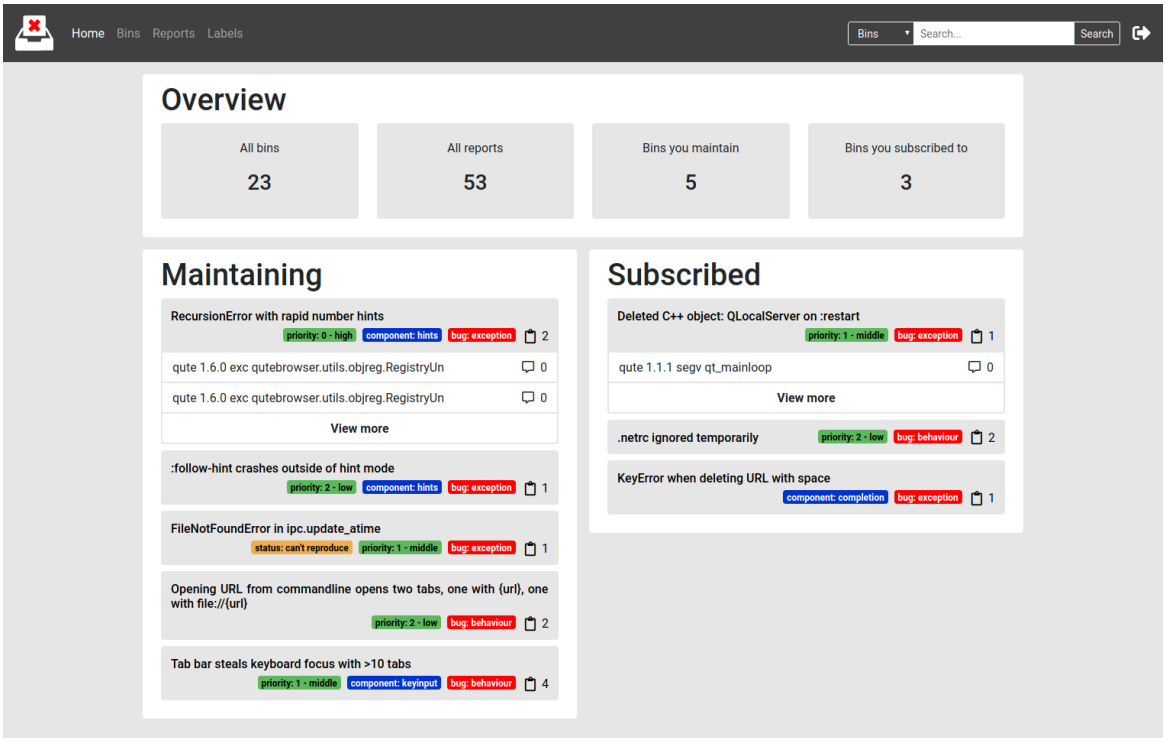



Abbildung 1.2.: Hauptansicht der Webapplikation

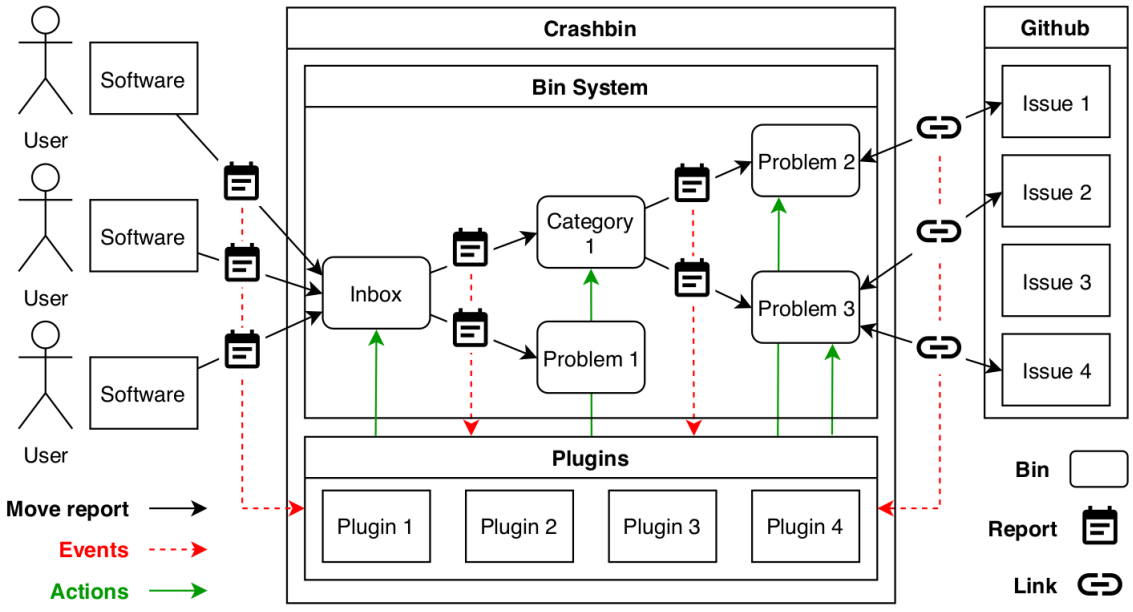



Abbildung 1.3.: Sytemübersicht

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis



# Quellcodeverzeichnis

# 2. Aufgabenstellung

## Bachelorarbeit «Crash Reporting»

### Einführung

Softwareprodukte verfügen häufig über eine Funktionalität, um Fehlverhalten zu erkennen und einen Fehlerreport automatisiert an ihren Hersteller zu senden. Bei diesem müssen sie dann ausgewertet und nachverfolgt werden. Grosse Organisationen, deren Software oft weit verwendet wird, setzen dabei mächtige Werkzeuge mit hohem Automatisierungsgrad ein. Für kleine Unternehmungen, wie beispielsweise Startups oder kleine Open-Source-Projekte, ist der Aufwand, ein solches Werkzeug aufzusetzen und zu betreiben oft unverhältnismässig hoch.

Die Beziehungen zu ihren Benutzern sind für solche kleinen Unternehmungen aber häufig ein sehr zentrales Anliegen, weil der Benutzerkreis meist nur über einen guten Ruf wächst. Deshalb werden Fehlerreports typischerweise manuell ausgewertet. Wenn der Benutzerkreis dann wächst, ist aber die Menge an Fehlerreports kaum noch vollständig vom Entwickler-Team zu verarbeiten.

### Aufgabe

Ziel dieser Bachelor-/Studienarbeit ist es, den in der Einführung beschriebenen Business-Case mit einem Software-Tool zu unterstützen. Die Studierenden sollen den Business-Case detailliert beschreiben und analysieren sowie Anforderungen an ein Tool dokumentieren, das Sie dann auch umsetzen sollen.

### Termine

Die Studienarbeit beginnt am 18.2.2019. Abgabetermin ist der 14.6.2019.

### Betreuung

Die Bachelor-/Studienarbeit wird durch Prof. Stefan Richter betreut. Jeden Dienstag von 14:10 bis 15:10 findet eine Besprechung statt, in der die Studierenden den Fortschritt der Arbeit präsentieren. Fragen und Angelegenheiten können ausserhalb dieses Termins auch per E-Mail erörtert werden.

Nach etwa der Hälfte der Zeit findet eine Zwischenpräsentation statt, an der die Studierenden den Fortschritt der Arbeit dem Betreuer und dem externen Experten sowie eventuell dem Gegenleser präsentieren.

### Bewertung

Die Arbeit wird anhand der folgenden sechs gleichgewichteten Punkte bewertet:

1. Organisation, Durchführung (Projektplanung u. Nachführung Arbeit gemäss Projektplan, Selbständigkeit, Einsatz, Zusammenarbeit mit Auftraggeber, Betreuer)

2. Bericht (Inhalt des Projektschlussberichts, Gliederung, Darstellung, Sprache der gesamten Dokumentation)
3. Problemanalyse (Vorstudie, Literaturstudium, Anforderungsspezifikation, Anforderungsanalyse, Domainanalyse)
4. Lösungsentwurf (Lösungsvarianten und deren Beurteilung, Variantenentscheid, Konzept, Entwurf)
5. Realisierung und Test
6. Bachelor-Präsentation

Für die Studienarbeit erfolgt die Bewertung zu gleichen Teilen nur für die Punkte 1 bis 5.

## Hinweise

Die folgende Seite bietet zahlreiche nützliche Informationen zum Schreiben einer wissenschaftlichen oder technischen Arbeit:

https://www.ifs.hsr.ch/index.php?id=13194&L=4metadata%2Foai_dc_1.dc

# 3. Technischer Bericht

## 3.1. Einleitung

### 3.1.1. Problemstellung / Vision

Mit dem Crashbin-Projekt wollen wir einen Server mit integrierter Weboberfläche für die Verwaltung von Fehler-Reports bei Programmabstürzen entwickeln.

Es existieren zwar bereits Lösungen für automatisierte Crash-Reports (wie beispielsweise Sentry[1] oder Airbrake[2]). Diese sind jedoch darauf ausgelegt, automatisiert eine grosse Menge von Fehlern bzw. Events zu verarbeiten (typischerweise Tausende bis Millionen pro Monat[3]).

Viele Projekte nutzen bereits ein Bug-Tracking-Tool wie etwa die "Issues"-Funktionalität von GitHub. Diverse Studien an Opensource-[4,5,6] als auch Industrie-Projekten[7] zeichnen jedoch ein klares Bild: Nutzer geben die Informationen, die von Entwicklern am dringendsten benötigt werden, nur selten an. So brauchen Entwickler primär Stack-Traces[8] und eine Anleitung zur Reproduzierung des Bugs; genau dies sind aber die Informationen, die in den Reports oft fehlen. Eine Studie,[9] die basierend auf Umfragen Vorschläge für bessere Bugtracker erarbeitet, kommt zu folgendem Schluss:

> Several reporters pointed out the need for tools that help them to collect information that they need to file bug reports. Ideally, such tools would be integrated in the software itself or its bug reporting system.

Automatisierte Crash-Reporter schaffen hier Abhilfe, da die benötigten Informationen grösstenteils automatisiert gesammelt werden können. Jedoch fokussieren sich bestehende Lösungen auf grosse Mengen von Reports. Daher erlauben sie es nicht, mit den Endbenutzern, die von den Crashes betroffen sind, zu kommunizieren. Dies zu tun, wäre jedoch wichtig – einerseits um eine

[1] `https://sentry.io/`

[2] `https://airbrake.io/`

[3] *Sentry Pricing*. URL: `https://sentry.io/pricing/` (besucht am 22. 04. 2019).

[4] Steven Davies und Marc Roper. „What's in a Bug Report?" In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '14. Torino, Italy: ACM, 2014, 26:1–26:10. ISBN: 978-1-4503-2774-9. DOI: 10.1145/2652524.2652541. URL: http://doi.acm.org/10.1145/2652524.2652541.

[5] Nicolas Bettenburg u. a. „What makes a good bug report?" In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM. 2008, S. 308–318.

[6] Nicolas Bettenburg u. a. „Quality of bug reports in Eclipse". In: *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*. ACM. 2007, S. 21–25.

[7] Vahid Garousi, Ebru Göçmen Ergezer und Kadir Herkilolu. „Usage, usefulness and quality of defect reports: an industrial case study". In: *Proceedings of the 20th international conference on evaluation and assessment in software engineering*. ACM. 2016, S. 39.

[8] Adrian Schroter u. a. „Do stack traces help developers fix bugs?" In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE. 2010, S. 118–121.

[9] Sascha Just, Rahul Premraj und Thomas Zimmermann. „Towards the next generation of bug tracking systems". In: *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE. 2008, S. 82–85.

Community aufzubauen und um zu verhindern, dass die Nutzer zu einer Alternative wechseln; andererseits um die Informationen zu bekommen, damit die Bugs behoben werden können.[10]

Hier setzt Crashbin an und hebt sich damit von bestehenden Lösungen ab: Es vereint die Idee eines Crash-Monitoring-Servers mit dem Konzept eines Ticket-Trackers, welche oft für effizienten Kundensupport benutzt werden.

### 3.1.2. Ziele

Die Kommunikation mit Endbenutzern, die Fehlerreports an die Entwickler senden, soll möglichst vereinfacht werden. Bestehende Lösungen (die im Abschnitt 3.2 genauer untersucht werden) erlauben es zwar, zusätzliche Informationen zu einem Fehler zu sammeln, wie beispielsweise eine Mail-Adresse. Crashbin soll einen Schritt weiter gehen und Mail-Antworten direkt aus der Crashbin-Applikation ermöglichen.

Trotz dem Fokus auf einige dutzende bis hunderte Reports pro Monat soll Crashbin skalierbar sein, sowohl in der Architektur als auch in der Benutzung. Es soll möglich sein, Abläufe wie die Triage eines Reports mit eigenen Plugins zu automatisieren.

Weiterhin soll Crashbin problemlos auf eigener Infrastruktur installierbar sein. Sind bereits Linux-Server in einer Organisation vorhanden, kann dies im Vergleich zu Cloud-Anbietern Kosten einsparen. Zusätzlich ist so sichergestellt, dass potenziell sensible Daten in Fehlerberichten nicht an Drittanbieter weitergegeben werden.

### 3.1.3. Rahmenbedingungen und Umfeld

#### Hintergrund des Projektes

Florian Bruhin ist der Autor des qutebrowser-Projekts[11]. Dabei handelt es sich um einen Webbrowser, der auf ein möglichst minimales User-Interface und eine effiziente Tastaturbedienung optimiert ist.

In qutebrowser ist seit längerer Zeit eine Funktionalität für die Meldung von Fehlern und Crashes eingebaut. Die entsprechenden Reports werden aber als Text in einem Pastebin abgelegt. Dort werden sie von Hand abgearbeitet, was jedoch schon bei mehreren Reports pro Tag nicht mehr genug skaliert – siehe dazu Abbildungen 3.2 und 3.3.

Aus dieser Situation wuchs die Idee eines Werkzeugs, um diesen Ablauf zu vereinfachen. Trotz des qutebrowser-spezifischen Hintergrunds soll Crashbin aber unabhängig davon sein: Alle wichtigen Aspekte sollen konfigurierbar sein, so dass Crashbin auch von anderen Projekten genutzt werden kann.

[10] Silvia Breu u. a. „Information Needs in Bug Reports: Improving Cooperation Between Developers and Users“. In: *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*. CSCW '10. Savannah, Georgia, USA: ACM, 2010, S. 301–310. ISBN: 978-1-60558-795-0. DOI: 10.1145/1718918.1718973. URL: http://doi.acm.org/10.1145/1718918.1718973.

[11] https://www.qutebrowser.org/

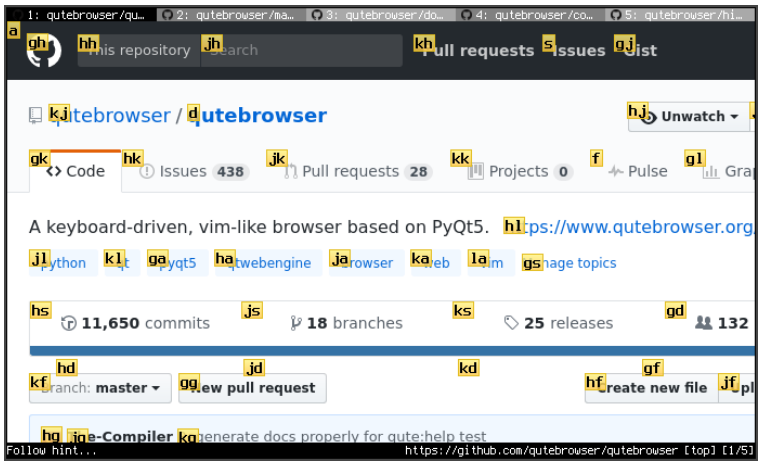



Abbildung 3.1.: Auswahl von Links mittels der Tastatur in qutebrowser

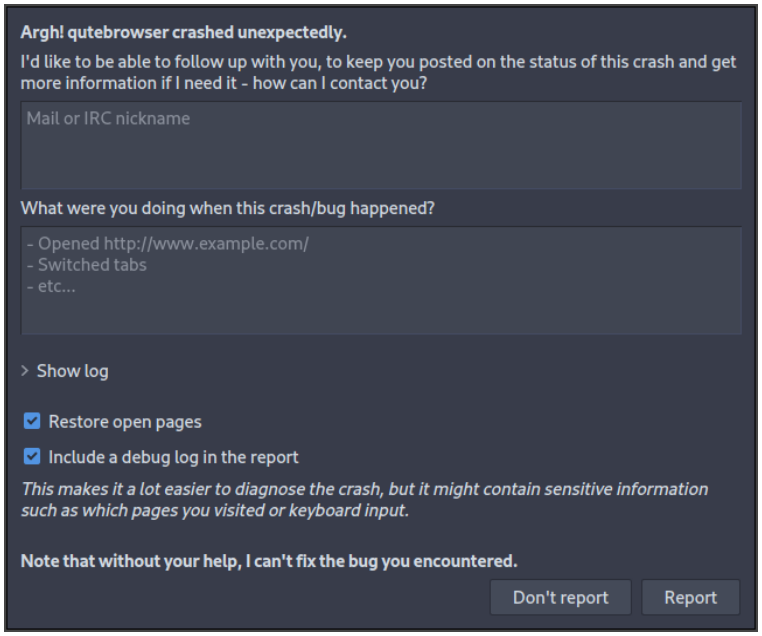



Abbildung 3.2.: Report-Dialog in qutebrowser

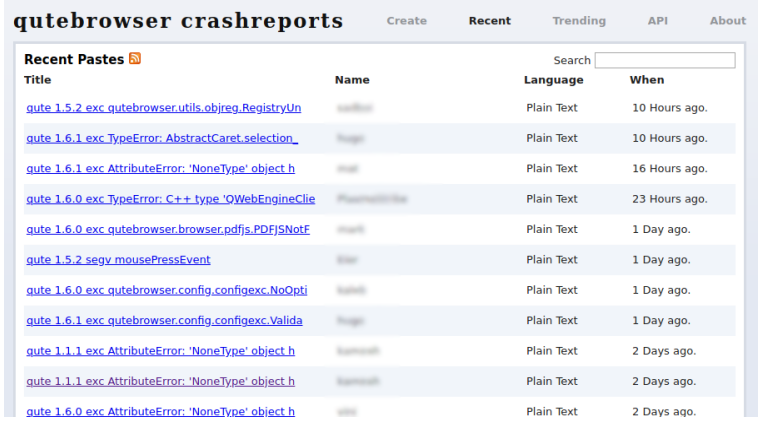



Abbildung 3.3.: Ungelesene Crash-Reports

**Zielgruppen**

Neben Opensource-Projekten wie qutebrowser sind Start-ups eine weitere Zielgruppe von Crashbin. Durch die in Abschnitt 3.1.2 genannten Ziele eignet sich hier Crashbin besser als bestehende Alternativen: Enge Beziehungen zu den Kunden und der Aufbau einer Community sind gerade für Start-ups sehr wichtig.

## 3.2. Stand der Technik – Was gibt es schon?

### 3.2.1. Crash-Monitoring

Bestehende Systeme um Abstürze zu sammeln und zu untersuchen, lassen sich grundsätzlich in zwei Kategorien einteilen:

- Tools von kommerziellen Anbietern wie in Abbildung 3.4 dargestellt.
- Tools, die Crashes von verschiedensten Applikationen in einer Linux-Distribution oder Desktopumgebung sammeln, siehe Abbildung 3.5.

Die in Abbildung 3.5 genannten Tools werden nicht weiter behandelt, da sie ein anderes Ziel erfüllen – wir sind an Lösungen interessiert, die sich in eine bestimmte Applikation integrieren lassen und von dessen Entwicklern genutzt werden.

Die Tools in Abbildung 3.4 kommen unserem Ziel näher, jedoch sind sie meist für Tausende bis Millionen von Reports/Events pro Monat konzipiert.[12] Sie sind stark auf Automatisierung ausgelegt, beispielsweise durch die automatische Erkennung und Zusammenfassung von identischen Reports.

Eine Funktionalität, die jedoch keine der Tools bietet, ist die Kommunikation mit den Personen, die von den Bugs betroffen sind. Es ist zwar möglich, zusätzliche Felder zur Erfassung einer Mail-Adresse zu konfigurieren, die Interaktion würde jedoch komplett manuell ablaufen. Siehe dazu auch die Anfragen bei den bestehenden Anbietern in Anhang F.

[12] *Sentry Pricing*.


Abbildung 3.4.: Bestehende Crash-Monitoring-Tools von kommerziellen Anbietern

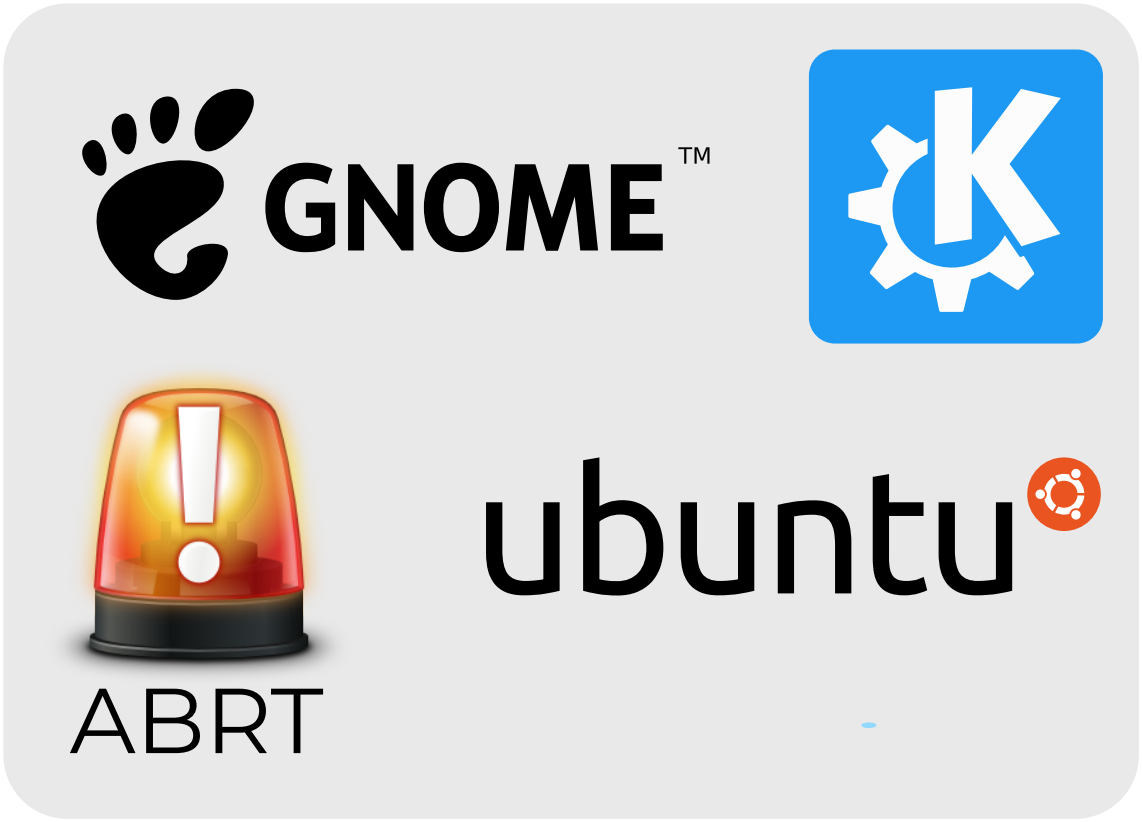



Abbildung 3.5.: Bestehende Crash-Monitoring-Tools, die systemweit agieren

Weiterhin sind fast alle dieser Tools Cloud-basiert und lassen sich nicht auf eigener Infrastruktur betreiben. Die einzigen Ausnahmen sind Sentry[13] sowie Errbit[14]. Bei Errbit handelt es sich um einen Opensource-Server, der eine kompatible API zu Airbrake (einem der genannten kommerziellen Anbietern) besitzt.

### 3.2.2. Ticket-Tracker

Für Support-Emails in Firmen wird oft ein sogenannter Ticket Tracker benutzt. Dabei handelt es sich um eine Software, die eingehende Mails von Kunden empfängt und es um ein vielfaches erleichtert, als Team auf diese zu reagieren.

Solche Tracker machen die Konversations-Historie für alle Mitarbeitenden sichtbar, bieten Textbausteine für vereinfachte Antworten, klare Zuständigkeitsbereiche für offene Tickets, und viele weitere Vorteile.

Ein bekanntes kommerzielles Produkt für diesen Zweck ist Zendesk[15]; auch Zoho, Salesforge oder Atlassian (JIRA) bieten entsprechende Lösungen an.

Da diese Art von Software sich ebenfalls stark mit unserer Vision kreuzt, haben wir folgende Projekte (alle Opensource) genauer angeschaut:

[13] https://github.com/getsentry/sentry
[14] https://github.com/errbit/errbit
[15] https://www.zendesk.com/

- OTRS: `https://otrs.com/`
- Request Tracker (RT): `https://bestpractical.com/`
- Django Helpdesk: `https://github.com/django-helpdesk/django-helpdesk`

Es wäre durchaus denkbar gewesen, ein solches Projekt (via API oder Code-Änderungen) zu erweitern. So wären wir eventuell ebenfalls bei einem Endprodukt, das sich mit unserer Vision deckt, angekommen.

Jedoch stellte sich das Konzept eines solchen Trackers als zu unflexibel für unsere Ziele heraus: Die Crashreports wären wiederum primär als unstrukturierter Text abgespeichert; eine entsprechende Kategorisierung oder auch Automatisierungen wären schwierig bis unmöglich.

Trotzdem sind diese Projekte nützlich als Inspiration für Crashbin, da solche Ticket-Tracker teilweise ähnliche Ziele wie Crashbin verfolgen:

- Organisation einer grossen Menge von Tickets, typischerweise mit Eigenschaften wie zuständige Personen (Assignees), verschiedene Kategorien/Abteilungen (Queues), Priorität, Status (Open, Closed, etc.) sowie Labels/Tags.
- Kommunikation mit Endbenutzern mittels einem integrierten Nachrichten-Editor.
- Vereinfachung von wiederkehrender Kommunikation durch Textbausteine.

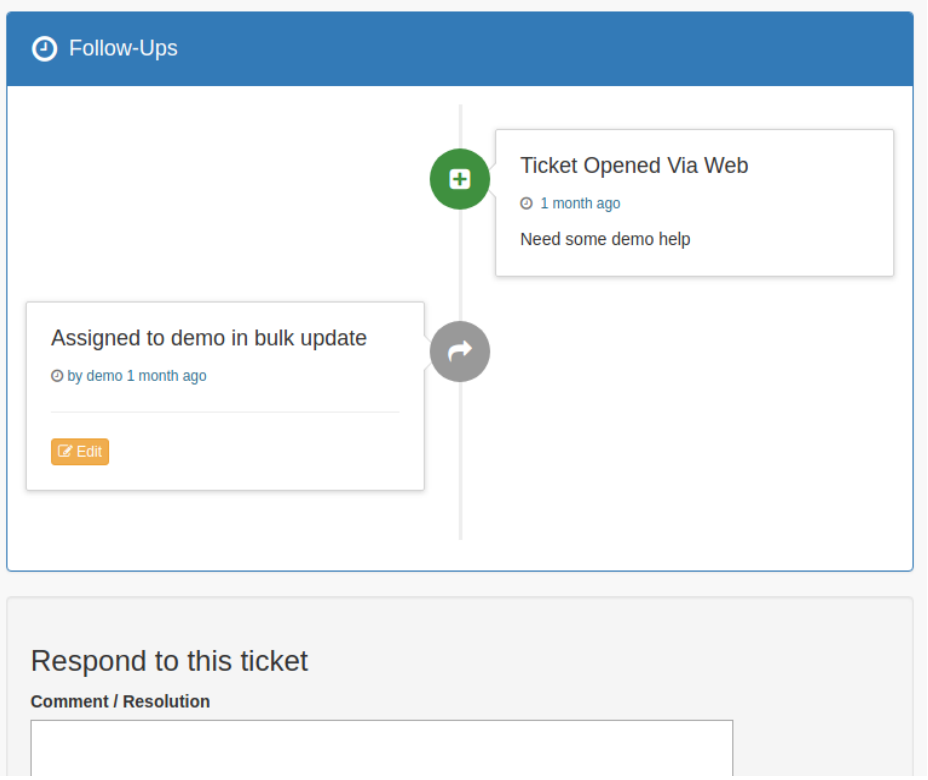



Abbildung 3.6.: Beispiel einer Konversation in einer Ticket-Tracking-Software (Django Helpdesk)

### 3.2.3. Opensource-Projekte

Um genauer zu erfahren, welche Lösungen von Opensource-Projekten für das Crash-Reporting genutzt werden, haben wir eine Diskussion in der Open Source Maintainers-Gruppe[16] von GitHub gestartet. Dabei handelt es sich um eine nicht-öffentliche Plattform für Diskussionen unter Opensource-Maintainern, bei der Florian Bruhin ein Mitglied ist.

Durch diese Diskussionen sowie weitere Recherchen haben wir die folgenden Erkenntnisse erlangt:

- Wie schon in Abschnitt 3.1.3 erwähnt, nutzt qutebrowser[17] einen Pastebin, um Crashreports zu sammeln. Florian Bruhin verschickt manuell Mails, um diese zu beantworten.
- Eric[18] (Python-IDE) verschickt Crash-Reports als Mail zum Entwickler. Dazu muss der Nutzer der IDE jedoch einen SMTP-Server für ausgehende Mails konfigurieren. In unserer Konversation mit dem Entwickler (Detlev Offenbach) erwähnte er, dass die Anzahl Reports zwar an einer Hand pro Woche abzählbar sei, jedoch viele Reports eintreffen für Fehler, die bereits behoben wurden.
- Electrum[19] (Bitcoin-Wallet) nutzt einen sehr minimalen Crashreport-Server[20], der automatisiert Issues auf GitHub öffnet[21]. In diesen Issues findet normalerweise die Kommunikation unter den Entwicklern statt, selten melden sich auch Endbenutzer zu Wort.
- OpenSUSE[22] (Linux-Distribution) nutzt eine eigene Instanz von Errbit. Keine Kommunikation mit den Endbenutzern findet statt.
- shields.io[23] (Web-Service) nutzt Sentry. Keine Kommunikation mit den Endbenutzern findet statt.
- Docker[24] (Container-Plattform) generiert für ihre Windows/macOS-Releases "Diagnostic IDs"[25], die dann für die (manuelle) Kommunikation mit den Entwicklern genutzt werden kann. Leider war es uns nicht möglich, mehr zu der von Docker verwendeten Software herauszufinden. Die Windows/macOS-Varianten von Docker sind closed-source und entsprechende Anfragen blieben unbeantwortet.
- Hashicorp[26] nutzt für ihre Produkte (Vagrant, Terraform, Packer, etc.) eine Eigenentwicklung namens Checkpoint[27]. Gemäss unserer Konversation mit einem der Entwickler ist der Workflow um Crashes zu finden bzw. zu beheben sehr improvisiert: Checkpoint nutzt Amazon AWS Athena[28] für die Crashreports, ein Commandline-Tool holt die Reports dort

[16] https://github.com/maintainers/public-resources
[17] https://www.qutebrowser.org/
[18] https://eric-ide.python-projects.org/
[19] https://github.com/spesmilo/electrum
[20] https://github.com/bauerj/crashhub
[21] https://github.com/spesmilo/electrum/issues/created_by/TARS-bot
[22] https://www.opensuse.org/
[23] https://shields.io/
[24] https://www.docker.com/
[25] https://docs.docker.com/docker-for-windows/troubleshoot/
[26] https://www.hashicorp.com/
[27] https://checkpoint.hashicorp.com/
[28] https://aws.amazon.com/athena/

ab und gruppiert diese (etwa nach der Produkt-Version). Keine Kommunikation mit den Endbenutzern findet statt.

- Grössere Projekte wie Firefox oder Chromium haben ebenfalls ihre eigenen Crash-Reporter (Socorro[29] bzw. Breakpad[30]). Diese sind für uns aber nicht weiter relevant, da das Volumen der Reports viel grösser ist, als für die Vision hinter Crashbin vorgesehen: Mozilla gab im Mai 2010 an,[31] pro Tag 2.5 Millionen Crashreports zu empfangen.

## 3.3. Eigener Lösungsansatz

### 3.3.1. Konzept für die Umsetzung

Bei der Konzeption von Crashbin war eine Analyse der Konzepte von bestehender Software sehr nützlich: Einerseits die genannten Ticket-Tracker wie OTRS oder Django Helpdesk; andererseits die bestehenden Crashmonitoring-Systeme wie Sentry oder Errbrake.

Sowohl Luca Tavernini als auch Florian Bruhin hatten bereits Zugriff auf eine OTRS-Instanz und einige Erfahrungen bei der Benutzung davon gesammelt. Um andere Produkte ebenfalls testen zu können, haben wir uns einen Account bei Sentry angelegt und lokal Instanzen von Django Helpdesk und Errbrake aufgesetzt.

Eine weitere Inspiration für die Organisation von Crashreports war die Organisation von Papierdokumenten in Büros. Mittels dieser Metapher entstand die Idee von "Bins" als Equivalent einer Papierablage, so wie sie auf vielen Schreibtischen steht.

### 3.3.2. Wichtige Elemente

#### Benutzer und Authentifizierung

Die Benutzer-Verwaltung von Crashbin ist momentan sehr einfach gehalten: Die einzige Aktion, die ohne einen Benutzer-Account verfügbar ist, ist das Abschicken eines Crash-Reports. Dies geschieht durch die Endbenutzer-Applikation, die bei einem Crash mit Crashbin interagiert.

Sollte Versand von ungewollten Reports ein Problem werden (beispielsweise von einem böswilligen Endnutzern oder automatisierten Spambots), könnte die API mittels einem API Key[32] und/oder einem Rate Limit[33] geschützt werden – es ist jedoch zu beachten, dass der API-Key von der Endbenutzer-Applikation mitgeschickt werden muss, und somit bei Opensource-Projekten auch öffentlich im Quellcode zu finden ist. Auch bei Closed-Source ist es schwierig, den API-Key von einer ungewollten Extraktion zu schützen. Es ist damit also nur ein begrenzter Schutz vor böswilligen Reports möglich.

[29] `https://github.com/mozilla-services/socorro`

[30] `https://chromium.googlesource.com/breakpad/breakpad/`

[31] Laura Thomson. *Socorro: Mozilla's Crash Reporting System*. 19. Mai 2010. URL: `https://blog.mozilla.org/webdev/2010/05/19/socorro-mozilla-crash-reports/` (besucht am 13. 06. 2019).

[32] Mirko Stocker u. a. „Interface Quality Patterns – Communicating and Improving the Quality of Microservices APIs". In: *23rd European Conference on Pattern Languages of Programs 2018*. Juli 2018. URL: `http://eprints.cs.univie.ac.at/5661/`, S. 4.

[33] Ebd., S. 8.

Alle anderen Aktionen wie das Anzeigen von Reports benötigen einen Benutzeraccount. Solche Accounts könnten beispielsweise an alle Hauptentwickler eines Projekts vergeben werden. Momentan sind alle Aktionen (wie beispielsweise das Umbenennen oder Löschen von Bins) allen Benutzern zugänglich – für grössere Firmen wäre es sinnvoll, die Benutzerverwaltung noch um Rollen bzw. unterschiedliche Berechtigungen zu erweitern ("Role-based access control"[34]).

### Reports

Bei einem Report handelt es sich um eine einzelne Meldung eines Endbenutzers. Tritt in einer Applikation ein Absturz auf (beispielsweise in der Form einer unbehandelten Exception), kann diese beispielsweise einen Dialog anzeigen, um den Benutzer zu bitten, einen solchen Crashreport zum Crashbin-Server zu schicken. Dieselbe Aktion kann aber auch manuell ausgelöst werden, je nach Applikation wäre dafür ein Menüeintrag, Kommando, etc. denkbar.

Bei einem Crashreport werden Endbenutzer nach einer Mailadresse gefragt, die dann für die Kommunikation via Crashbin benutzt wird. Zum weiteren Inhalt von Crash-Reports macht Crashbin keine Vorgaben. Je nach Applikation, die in Crashbin integriert wird, sind andere Informationen sinnvoll: Bei einem Browser ist beispielsweise die momentan besuchte Webseite interessant, währenddessen bei anderen Applikationen die Versionsnummern der Abhängigkeiten mitgeschickt werden könnten.

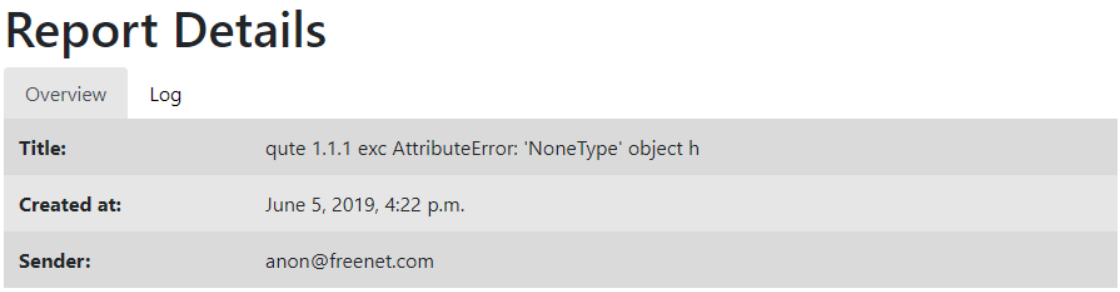



Abbildung 3.7.: Beispiel eines Reports

### Nachrichten

Jeder Report enthält eine Chronik der Interaktion mit dem betroffenen Benutzer. Es ist möglich, direkt aus Crashbin auf einen Report zu antworten. Ähnlich wie in einem Ticketsystem wird die Antwort in der Weboberfläche dargestellt (was es auch ermöglicht, die Kommunikation auf mehrere Personen aufzuteilen). Bei Antworten enthält der Benutzer jeweils eine E-Mail, aber auch interne Notizen zu einem Report sind möglich.

[34] E. Fernandez-Buglioni. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. Wiley Software Patterns Series. Wiley, 2013. ISBN: 9781119970484, S. 249.

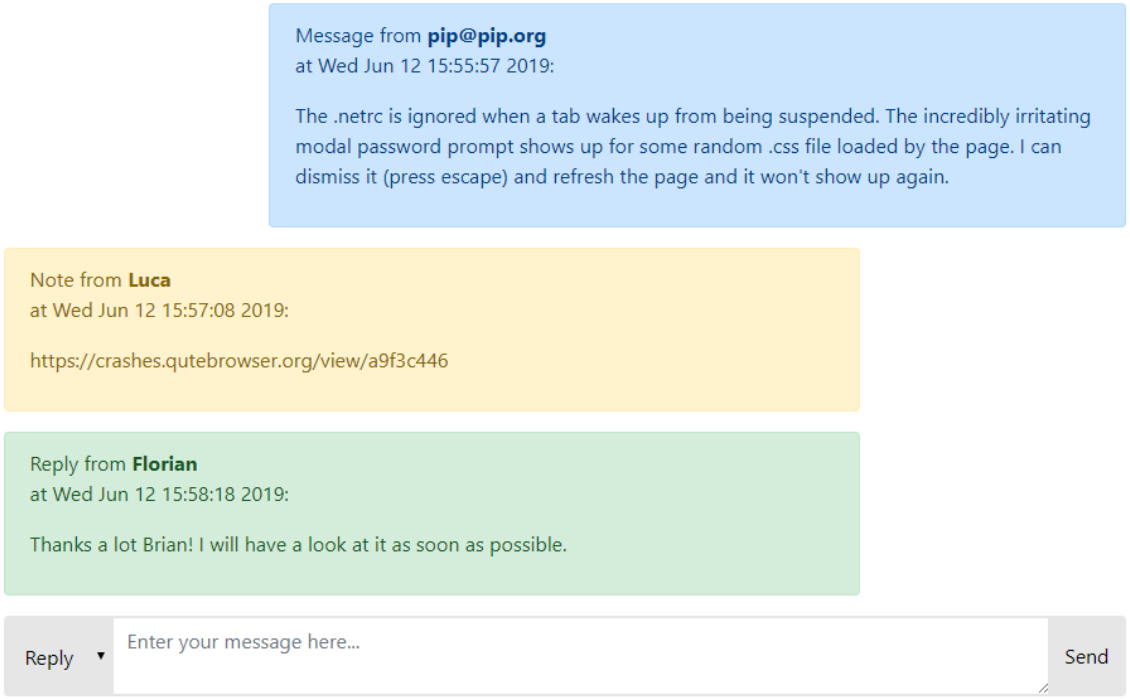



Abbildung 3.8.: Beispiel einer Konversation

### Ereignisse und Erweiterungen

Im Kontrast zu bestehenden Systemen steht bei Crashbin die manuelle, menschliche Konversation mit den Benutzern im Vordergrund. Trotzdem soll es möglich sein, gewisse Abläufe zu automatisieren, um auch mit einer steigenden Anzahl von Reports umgehen zu können.

Crashbin soll deshalb erweiterbar sein, um eine sukzessive Automatisierung wo sinnvoll und nötig zu ermöglichen. Dazu soll eine API zu Plugins exponiert werden, die es erlaubt, auf gewisse Ereignisse (wie etwa ein neuer Report) zu reagieren, und beispielsweise gewisse Reports automatisch zu kategorisieren und zu beantworten.

### Bins

Bins ("Behälter") erlauben es, Reports zu gruppieren. Ein Report ist dabei – wie bei der Methapher der Papierablage – immer in genau einem Bin enthalten.

Wird von einer Applikation ein neuer Report abgesandt, so landet dieser in einem konfigurierbaren "Inbox"-Bin, ähnlich zum Posteingang bei einem Mailclient.

Bins können (einen oder mehrere) "Maintainer" haben, also Personen, die für einen Bin zuständig bzw. verantwortlich sind. Weiterhin können Benutzer einen Bin abonnieren. In beiden Fällen werden die entsprechenden Personen mit einer E-Mail benachrichtigt, sobald ein Report in dem entsprechenden Bin abgelegt wird.

Der Unterschied zwischen Maintainern und Subscribern (Abonnenten) besteht darin, dass die Maintainer auf der Seite aufgelistet sind (da es sich dabei um die Personen handelt, die man bei

Rückfragen zum entsprechenden Bin kontaktieren sollte) – Abonnenten sind hingegen einfach an Änderungen zu einem Bin interessiert, ohne aufgelistet zu werden. Dies entspricht Plattformen wie GitHub, GitLab oder Jira, wo bei einem Issue ebenfalls "Assignees" sowie "Subscriber" (oder "watcher") existieren.

Die Nutzer von Crashbin sind grundsätzlich frei darin, wie sie Bins für die Organisation von Reports nutzen. Wir haben uns für die Implementation von gewissen Funktionen an den folgenden zwei Nutzungen für Bins orientiert:

1. Um alle Reports für einen bestimmten Bug in einer Software zu gruppieren.
2. Um alle Reports für ein bestimmtes Subsystem bzw. Komponente einer Software zu gruppieren.

Für Punkt 1 ist es wichtig zu sehen, dass Crashbin keine bestehenden Issue-Tracker, die von Projekten oder Firmen oft bereits benutzt werden, ersetzen will. Bei Opensource-Projekten ist der Bugtracker oft eine öffentliche Anlaufstelle für Problemen von Benutzern, die nicht zwingendermassen auf einen Crash zurückzuführen sind. In Firmen wird Software wie Jira oder Redmine für die Planung für Entwicklungs-Sprints genutzt, mit Funktionen, die den Umfang von Crashbin bei weitem sprengen würden.

Crashbin soll die bestehenden Arbeitsabläufe und die dahinterstehende Software lediglich ergänzen: So soll es mit entsprechenden Erweiterungen möglich sein, Crashbin beispielsweise mit GitHub zu verknüpfen. Mit einem Link innerhalb von Crashbin gelangt man so zu einem zugehörigen GitHub-Issue; auf GitHub wird mittels einem automatisiert erstellten Kommentar auf die zu einem Issue gehörenden Reports verlinkt.

Punkt 2 ist primär für grössere Projekte interessant: Die Personen, die für den Inbox-Bin zuständig sind, müssen so nicht mehr alle Bugs (bzw. Bins) einer Software im Detail kennen. Stattdessen entscheiden sie anhand der Informationen in einem Report, in welchem Subsystem das Problem auftrat und geben die Verantwortlichkeit für den Report durch Verschieben in den Subsystem-Bin an andere Personen weiter.

Werden Bins genutzt, um die Reports zu einem bestimmten Problem zu sammeln, so stellt sich die Frage, was passiert, falls in einem Report mehrere Probleme gleichzeitig ersichtlich sind. In Crashbin lässt sich dieses Problem auf zwei Arten lösen:

- Es wird ein Bin für alle Reports erstellt, die dieselbe "Signatur" von Problemen aufweisen, bei denen also dieselbe Kombination von Problemen auftaucht. Dieser wird dann mit mehreren Issues in einem Issue-Tracker (z.B. GitHub) verknüpft, anstatt den Bin mit einem einzelnen Issue zu verknüpfen.
- Es wird ebenso ein Bin für die entsprechende Problem-Signatur erstellt, jedoch werden für die einzelnen Probleme nochmals separate Bins erstellt. Die verschiedenen Bins lassen sich dann durch die "Related Bins"-Funktionalität untereinander verknüpfen.

Es wären folgende andere Lösungen denkbar gewesen, die jedoch wegen der zu hohen Komplexität bei der Benutzung verworfen wurden:

- Anstatt dass ein Report nur in einem Bin liegt, könnte ein Report mehreren Bins gleichzeitig zugeordnet werden. Dies würde jedoch das "mentale Modell" eines Benutzers von Crashbin komplexer machen, da es der Metapher einer Papierablage nicht mehr gerecht wird.

- Es könnte eine weitere Kategorisierungsebene (Report → Problem → Bin) eingeführt werden, oder das Konzept der Bins könnte komplett verworfen werden. Wir haben uns gegen diese Optionen entschieden, da sonst diverse weiterführende Konzepte (wie die zuständigen Personen für einen Bin und die entsprechenden Benachrichtigungen) zu komplex geworden wären.

- Anstatt dass es eine flache Liste von Bins gibt, könnten sich Bins ähnlich wie Ordner verschachteln lassen. Dies würde ebenfalls weiterführende Konzepte wie die Verlinkung zu einem Bugtracker oder die Zuständigkeitsbereiche von Personen unintuitiver gestalten.

Wir schätzen die Wahrscheinlichkeit, dass mehrere Probleme gleichzeitig in einem Report ersichtlich sind, relativ gering ein – zumindest in Sprachen, die Exceptions nutzen. Sollte dieser Fall eintreten, so gibt es vermutlich mehrere Endbenutzer, bei denen dieselbe Kombination von Fehlern auftritt, womit es sinnvoll ist, dafür einen eigenen Bin anzulegen. Weiterhin ist die "Related Bins"-Funktionalität flexibler: Sie lässt sich auch für andere Workflows nutzen, beispielsweise um Probleme zu verknüpfen, die verschieden aber verwandt sind.

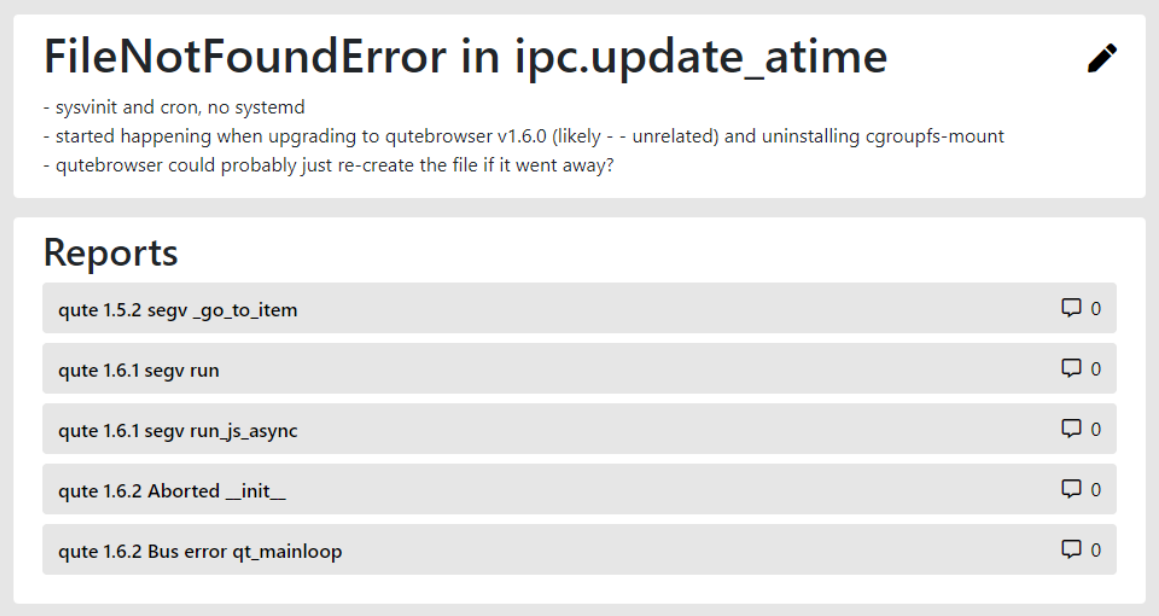



Abbildung 3.9.: Beispiel eines Bins

### Labels

Wenn viele Bins bzw. Reports vorhanden sind, sind weitere Werkzeuge nützlich, um diese sinnvoll zu gruppieren oder filtern zu können. Zu diesem Zweck unterstützt Crashbin ähnlich wie GitHub oder GitLab das Anlegen von Labels (Tags). Ein Label besitzt einen Namen, eine optionale Beschreibung, sowie eine Farbe für die einfachere visuelle Unterscheidung von verschiedenen Labels. Diese Labels können dann zu Bins und/oder Reports hinzugefügt werden, um beispielsweise alle Bins für einen bevorstehenden Bugfix-Sprint zu markieren. Auf dieselbe Art könnten etwa alle Reports markiert werden, die von einem besonders wichtigen Kunden erstellt wurden.

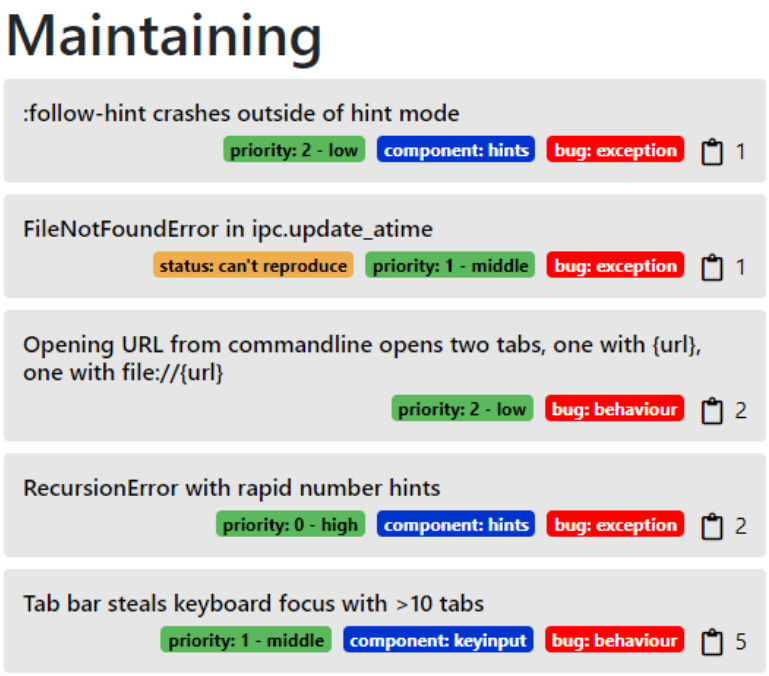



Abbildung 3.10.: Beispiel von Labels, die an Reports angehängt sind

### Konfiguration

Die Konfiguration von Crashbin findet in einer Konfigurationsdatei statt, die vom Server beim Start gelesen wird. Dort sind beispielsweise die zu verwendende Datenbank, der Name des Inbox-Bins oder eigene Stylesheets (CSS) konfigurierbar. Die Konfiguration ist in der Softwaredokumentation in Kapitel **??** genauer beschrieben.

Wird Crashbin in Zukunft um eine rollenbasierte Benutzerverwaltung erweitert (siehe Seite 19), so wäre es denkbar, dass die Konfiguration von Administratoren direkt in der Weboberfläche anpassbar ist, anstatt auf dem Server eine Konfigurationsdatei anpassen zu müssen. So lassen sich Serveradministratoren trennen von Crashbin-Administratoren, was gerade für Firmen die Konfiguration von Crashbin erleichtern würde.

# 4. Projektdokumentation

## 4.1. Anforderungsspezifikation

### 4.1.1. Use Cases

In der folgenden Abbildung ist eine Übersicht der Use Cases dargestellt, welche in diesem Kapitel genauer beschrieben werden. Aufgrund der verkürzten Elaborationsphase wurden die Use Cases nur im Casual Format ausgearbeitet. Auf das Fully Dressed Format wurde verzichtet.

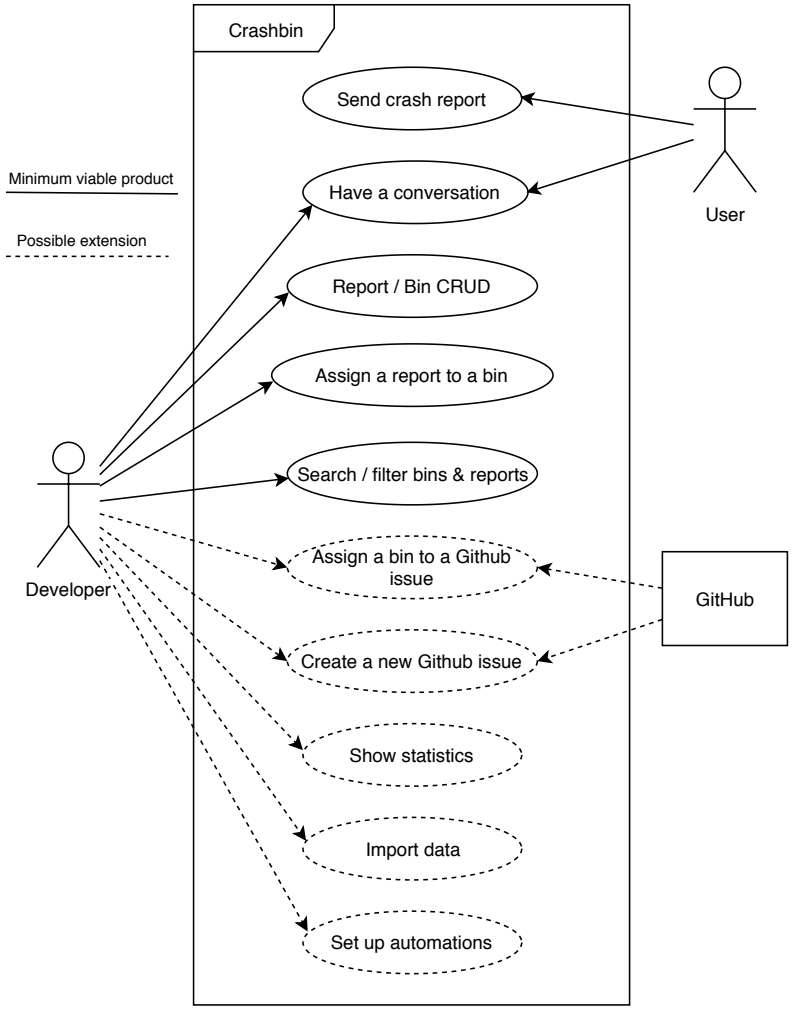



Abbildung 4.1.: Use Case-Diagramm

**Crashreport empfangen**

- Hauptszenario:
  - Ein Benutzer erhält beim Gebrauch einer Applikation einen kritischen Fehler. Über einen Crashreport-Dialog sendet er einen Crashreport an die Entwicklerin. Hierbei kann er optional seine Email-Adresse und/oder einen Kommentar hinzufügen.
- Alternative Szenarien:
  - Der Benutzer nutzt ein Kommando/Menüeintrag/etc. um manuell einen Report abzusenden, ohne dass eine Exception ausgelöst wurde.

**Crashreports verwalten**

- Hauptszenario:
  - Die Entwicklerin legt für jedes bekannte Problem einen eigenen Bin an. Erhaltene Crashreports werden fortlaufend auf diese Bins verteilt.
- Alternative Szenarien:
  - Die Entwicklerin erhält Crashreports, welche sie noch nicht einem bestimmten Problem zuordnen kann. Deshalb erstellt sie Bins für Subsysteme oder temporäre Kategorien, um eine Vorsortierung solcher Reports zu tätigen.

**Unterhaltung führen**

- Hauptszenario:
  - Die Entwicklerin erhält anhand eines Crashreports den Kommentar des Benutzers und dessen Email-Adresse. Sie analysiert den Crashreport und schickt dem Benutzer ihre Erkenntnisse.
- Alternative Szenarien:
  - Der Benutzer antwortet auf den Bericht der Entwicklerin. Die Nachricht des Benutzers wird automatisch dem schon existierenden Report zugeordnet und als Unterhaltung dargestellt. Die Entwicklerin wird über die neue Nachricht informiert.
  - Die Entwicklerin sendet nachträglich eine Nachricht über Crashbin an den Benutzer, um zusätzliche Fragen zu stellen. Die von ihr gesendete Nachricht wird automatisch dem bereits vorhandenen Report hinzugefügt und als Unterhaltung dargestellt.

**Reports suchen und filtern**

- Hauptszenario:
  - Die Entwicklerin sucht einen bestimmten Report anhand des Titels. Das System findet den Report und stellt ihn mit allen Details dar.

- Alternative Szenarien:
  - Die Entwicklerin sucht eine Menge von Reports, die ein bestimmtes Kriterium erfüllen (beispielsweise “Report ist mit Issue X verknüpft”, etc.).
  - Die Entwicklerin sucht einen Report, der ein bestimmtes Wort beinhaltet. Das System sucht anhand einer Volltext-Suche die entsprechenden Reports und stellt diese dar.

**GitHub Issue dem Ticket zuordnen**

- Hauptszenario:
  - Die Entwicklerin erkennt einen Crashreport und weiss, dass auf GitHub hierfür schon bereits ein Issue eröffnet wurde. Sie verknüpft das Ticket mit dem Issue.
- Alternative Szenarien:
  - Es existiert kein passendes Issue für den Crashreport. Die Entwicklerin kann ein neues Issue dafür erstellen.

**Statistiken einsehen**

- Hauptszenario:
  - Die Entwicklerin kann sich eine Statistik der Bins und Reports anzeigen lassen. Mögliche Anzeige: Wie viele Crashreports sind einem gewissen Issue zugeordnet, welche Version der Applikation hat welche Probleme, usw.
- Alternative Szenarien:
  - Kein alternatives Szenario vorhanden.

**Import von Daten**

- Hauptszenario:
  - Die Entwicklerin hat einen bestehenden Datensatz von Crashreports und importiert diese in das System.
- Alternative Szenarien:
  - Es existieren ältere Versionen der Applikation, welche noch nicht aktiv Crashbin unterstützen. Die Entwicklerin implementiert einen Adapter, welcher die Daten für Crashbin aufbereitet.

**Automatisierung einrichten**

- Hauptszenario:

  - Die Anzahl Tickets nimmt zu und die Entwicklerin ist nicht mehr in der Lage, all diese händisch zu verarbeiten. Sie richtet eine Automatisierung ein, welche Tickets nach vorbestimmten Kriterien automatisch verarbeitet und beantwortet.

- Alternative Szenarien:
  - Kein alternatives Szenario vorhanden.

### 4.1.2. Nicht-funktionale Anforderungen

#### Funktionalität

- Sicherheit
  - Der Zugriff auf das System erfolgt passwortgeschützt.
  - Die Webapplikation kann so eingerichtet werden, dass die Verbindung verschlüsselt wird.
  - Sollten Informationen über die Webapplikation einem Benutzer zur Verfügung gestellt werden, sind diese Informationen nicht öffentlich, sondern nur für ihn ersichtlich.
- Interoperabilität
  - Crashbin exponiert eine Schnittstelle für Plugins, um mit externen Issue-Trackern (z.B. GitHub) interagieren zu können.

#### Zuverlässigkeit

- Fehlertoleranz
  - Das System soll auch in einem fehlerhaften Zustand möglichst weiterhin Crashreports annehmen können.
- Wiederherstellbarkeit
  - Das System lässt das Erstellen und Einspielen von Backups der Daten zu.

#### Übertragbarkeit

- Installierbarkeit
  - Das System ist einfach und ohne grossen Konfigurationsaufwand installierbar.
  - Für die Installation müssen nicht zuerst Services vorinstalliert werden.

## 4.2. Analyse (Business-Modell)

### 4.2.1. Domain-Modell

In Abbildung 4.2 ist das gesamte Domain-Modell als Übersicht dargestellt. Für weitere Erläuterungen wird nachfolgend das Domain-Modell in drei Bereiche aufgeteilt.

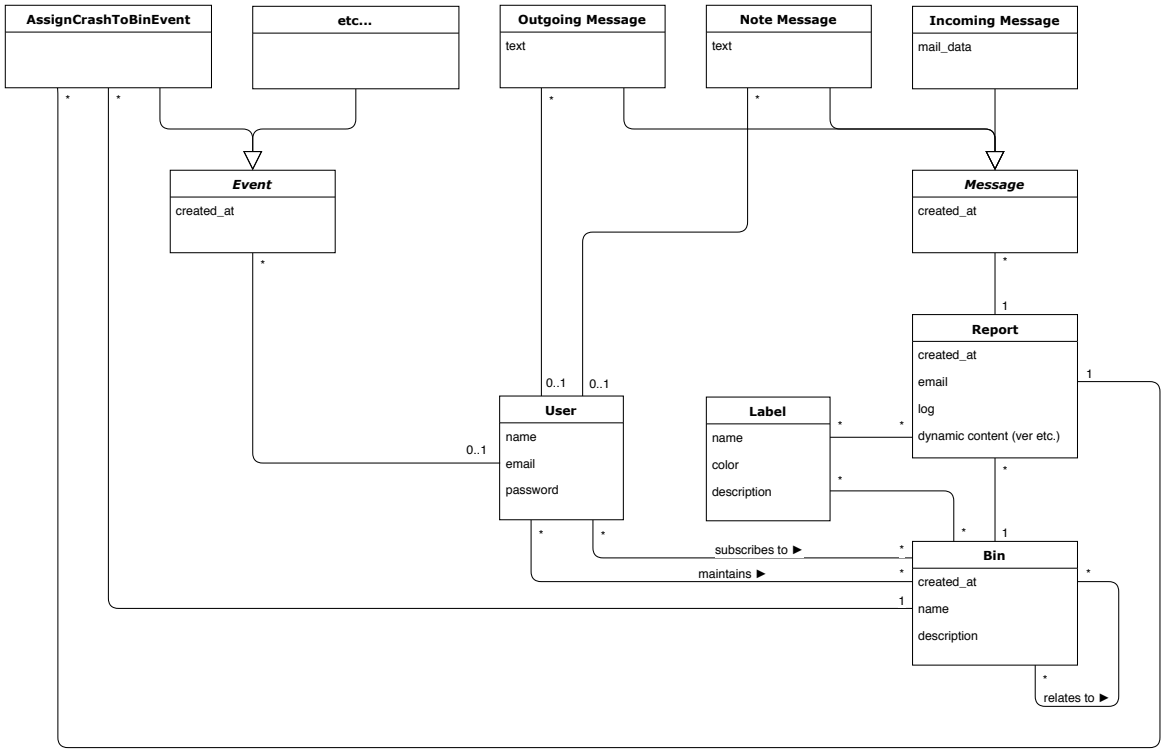



Abbildung 4.2.: Domain-Modell: Übersicht

#### Ablagesystem

Dieser Teil des Domain-Modells beinhaltet das Kernkonzept von Crashbin. Hierbei sind folgende Punkte bemerkenswert:

- Reports können für die Sortierung immer nur einem Bin zugewiesen werden.
- Labels sind universell einsetzbar und können somit für Bins wie auch Reports gebraucht werden.
- Bins können beliebig viele Verweise auf andere Bins besitzen. Diese Verweise sind symmetrisch, womit also die Verknüpfung in beide Richtungen getätigt wird.
- Anwender von Crashbin können sich bei Bins als Verantwortliche eintragen oder einen Bin abonnieren, um über zukünftige Änderungen informiert zu werden.

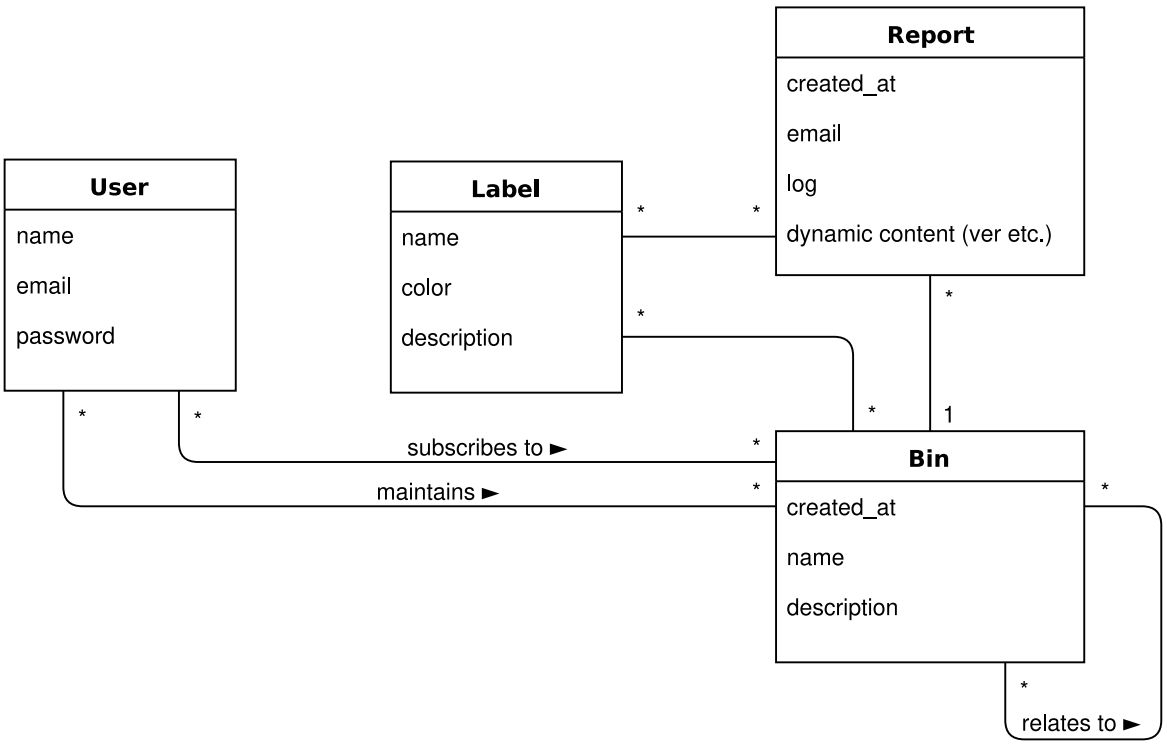



Abbildung 4.3.: Domain-Modell: Kernkonzept

### Messaging

In diesem Teil des Domain-Modells geht es um die Kommunikation zwischen Crashbin und dem User:

- Messages können immer nur zu einem Report gehören.
- Es gibt drei verschiedene Arten von Messages:
  - Incoming Messages: Nachrichten, welche vom User an Crashbin gesendet werden.
  - Note Messages: Notizen, welche von den Entwicklern als zusätzliche Information abgespeichert werden können.
  - Outgoing Messages: Antworten von den Entwicklern an den User.
- Weitere Arten von Messages sind denkbar: Beispielsweise System-Messages, welche über Events informieren (z.B. Name wurde geändert, Report wurde zu neuem Bin hinzugefügt, usw.).
- Note- und Outgoing Messages sind zusätzlich noch mit einem User verknüpft, damit ersichtlich ist, wer diese Nachrichten verfasst hat. Bei Mails besitzt der externe Absender keinen Crashbin-Account.

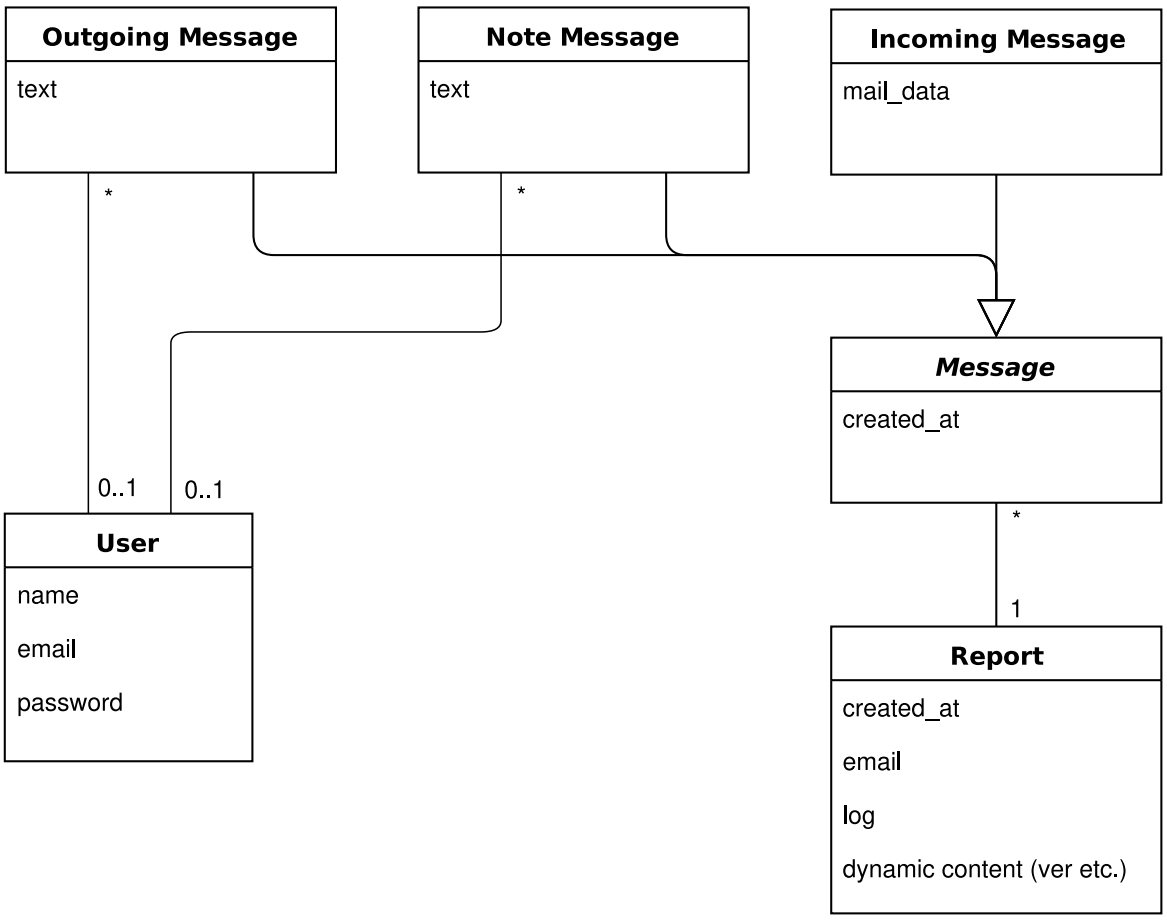



Abbildung 4.4.: Domain-Modell: Messaging

### Events

Der letzte Teil des Domain-Modells beschäftigt sich mit den Events. Hier wird als Beispiel nur ein konkreter Event dargestellt. Es sind jedoch analog dazu andere Events vorstellbar. Bei diesem Abschnitt sind folgende Punkte nennenswert:

- Events können optional einen User zugeordnet haben. Wenn ein User angegeben ist, ist der Event von einem Entwickler ausgelöst worden. Wenn kein User vorhanden ist, ist der Event von Crashbin selbst ausgelöst worden (z.B. wenn ein neuer Report eintrifft).
- Je nach Funktion und Zweck des Events ergeben sich dann andere Verknüpfungen zu den jeweiligen Klassen.

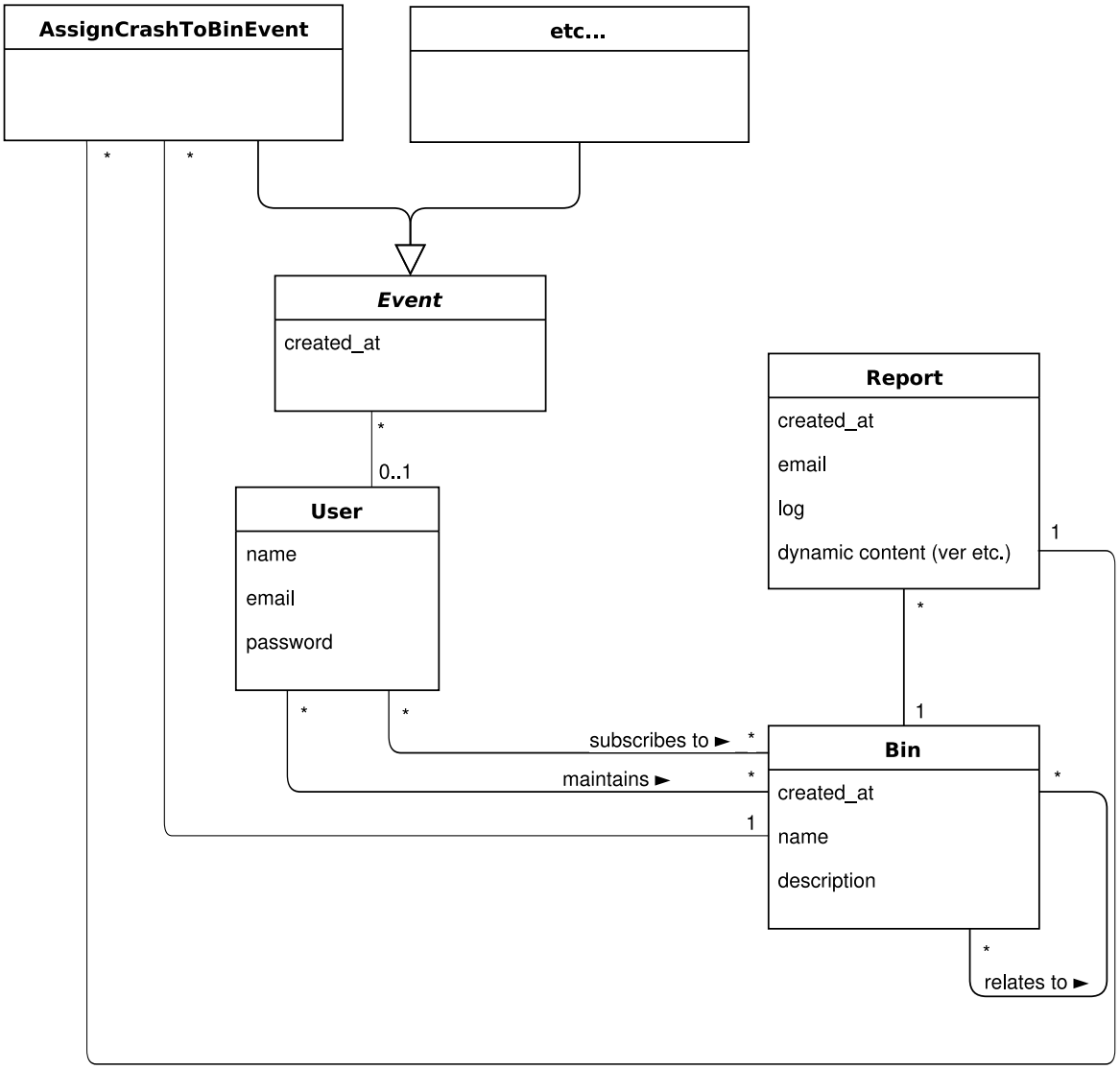



Abbildung 4.5.: Domain-Modell: Events

## 4.3. Design

### 4.3.1. Architektur und Abhängigkeiten

#### Grundsätzliche Überlegungen

Bei der Auswahl von Technologien und insbesondere Abhängigkeiten waren folgende Überlegungen interessant:

**Einfache Entwicklung** Inwiefern erleichert oder erschwert die Wahl unsere Entwicklung? Wir haben relativ wenig Zeit für die Entwicklung eines Prototyps und wollen vermeiden, das Rad neu zu erfinden.

**Nutzererlebnis** Inwiefern verbessert oder verschlimmert die Wahl das Nutzererlebnis? Inwiefern beeinflusst sie die gewünschten nicht-funktionalen Anforderungen (Abschnitt 4.1.2)?

**Verbreitung** Wie weit sind die infrage kommenden Projekte verbreitet? Auch bei Abhängigkeiten sollten diese von mindestens einigen Dutzend anderen Projekten benutzt werden. So ist eher sichergestellt, dass die benutzten Projekte eine gewisse Stabilität und Reife besitzen. Um dies abschätzen zu können, nutzen wir Statistiken von PyPI (Python Package Index) sowie GitHub, siehe Anhang C.

**Qualität** Befolgen die genutzten Projekte "best practices"? Sind eine übersichtliche Dokumentation sowie automatisierte Tests vorhanden?

**Aktivität** Werden die Projekte immer noch aktiv weiterentwickelt? Existiert eine gewisse Community dahinter, oder hängen die Projekte von einem einzelnen Maintainer ab?

**Exit-Strategien** Falls die genutzten Projekte in Zukunft Probleme verursachen sollten oder nicht mehr weiterentwickelt würden: Wie schwierig wäre es dann, diese gegen ein anderes Projekt auszutauschen oder die entsprechende Funktionalität selbst zu entwickeln?

### Webapplikation

Wir haben uns bei Crashbin für die Entwicklung einer Webapplikation entschieden.

Grundsätzlich wäre auch eine Client-Server-Anwendung denkbar gewesen: Eine Serverkomponente als Backend für die Sammlung und Speicherung von Reports, sowie eine Desktopapplikation (Cross-Platform) für die Verwaltung, welche mit dem Server kommuniziert.

Webapplikationen sind für ähnliche Projekte sehr weit verbreitet (siehe Abschnitt 3.2). Weiterhin sind sie problemlos von jeglichen Betriebssystemen (inklusive Android/iOS) benutzbar, ohne dass auf den Endgeräten neue Software installiert werden muss. Der Aufwand für die Implementation wird dadurch ebenfalls vereinfacht, da wir eine Serverkomponente mit integrierter Web-Oberfläche entwickeln können, anstatt zwei verschiedene Applikationen zu entwickeln.

### Programmiersprache

Es existieren verschiedenste Frameworks, um in diversen Programmiersprachen Webapplikationen zu erstellen, darunter:

- Python (Django/Flask/etc.)
- C# (ASP.NET)
- Java (Spring)
- JavaScript (Express/Node.JS)
- PHP (Laravel)
- Ruby (Rails)

C#, Java, PHP und Ruby waren keine Option, da wir wenig Interesse und keine Erfahrung mit diesen Technologien haben.

JavaScript wäre eine Option, da wir in den WED-Vorlesungen etwas Erfahrung damit gesammelt haben. Wir haben uns jedoch schnell für Python entschieden, da Florian Bruhin damit schon viel Erfahrung mitbringt und Luca Tavernini schon seit längerer Zeit mehr Python-Kenntnisse erwerben wollte.

Weiterhin ist Python die momentan am schnellsten wachsende Programmiersprache und bietet uns eine Auswahl von weit verbreiteten Web-Frameworks.[1]

### Web-Framework

Es existieren diverse Web-Frameworks für Python, weshalb wir vor einer genaueren Evaluation auf die bekanntesten davon beschränken wollen. Diese Beschränkung macht Sinn, da so eher sichergestellt ist, dass die verwendeten Technologien noch länger fortbestehen. Weiterhin ist es bei weit verbreiteten Frameworks einfacher, bei Problemen Hilfe, gute Dokumentation und andere wichtige Ressourcen zu finden.

In einer Umfrage[2] der PSF (Python Software Foundation) und Jetbrains nutzten 47% der rund 18'000 Teilnehmenden Flask; 45% Django. Andere Web-Frameworks fristen eher ein Nischendasein: Nach Django folgt das Tornado-Framework, welches nur 6% der Teilnehmenden nutzen; alle weiteren Antwortmöglichkeiten (Pyramid, web2py, Bottle, CherryPy, Falcon, Hug, Turbogears) erreichen unter 5%.

Die Umfrage schliesst mit dem "Key Takeaway":

> Flask and Django are the most popular frameworks among web developers. Having equal shares (around 45% each), they leave other Python web frameworks far behind.

Hervorzuheben ist, dass die Nutzung von Flask aus unbekannten Gründen gegenüber dem Vorjahr um 15 Prozentpunkte gestiegen ist:

> Surprisingly, compared to the previous year, Flask usage has grown by 15 percentage points among the respondents of our survey and as such, this year Flask has become the most popular web framework. Django was selected by 45% of respondents (41% in 2017).

Auch eine Umfrage[3] von Stack Overflow kommt zu einem ähnlichen Ergebnis: Dort nutzten 13% der rund 64'000 Teilnehmenden Django; 12.1% Flask. Die insgesamt geringeren Zahlen erklären sich dadurch, dass diese Umfrage sich an Programmierer in allen Programmiersprachen richtet, während sich die PSF-Umfrage auf Python fokussiert.

Wenn man sich die Philosophie hinter Django und Flask genauer anschaut wird klar, dass sich diese grundlegend unterscheiden. Flask bezeichnet sich selbst als ein "Micro-Framework" und definiert[4] diesen Begriff wie folgt:

---

[1] Stack Overflow. *Stack Overflow Developer Survey Results 2019*. Apr. 2019. URL: `https://insights.stackoverflow.com/survey/2019` (besucht am 04. 06. 2019).

[2] Python Software Foundation und JetBrains. *Python Developers Survey 2018 Results*. Feb. 2019. URL: `https://www.jetbrains.com/research/python-developers-survey-2018/` (besucht am 04. 06. 2019).

[3] Stack Overflow, *Stack Overflow Developer Survey Results 2019*.

[4] *Foreword - Flask 1.0.2 Documentation*. URL: `http://flask.pocoo.org/docs/1.0/foreword/` (besucht am 04. 06. 2019).

> "Micro" does not mean that your whole web application has to fit into a single Python file (although it certainly can), nor does it mean that Flask is lacking in functionality. The "micro" in microframework means Flask aims to keep the core simple but extensible. Flask won't make many decisions for you, such as what database to use. Those decisions that it does make, such as what templating engine to use, are easy to change. Everything else is up to you, so that Flask can be everything you need and nothing you don't.
>
> By default, Flask does not include a database abstraction layer, form validation or anything else where different libraries already exist that can handle that. Instead, Flask supports extensions to add such functionality to your application as if it was implemented in Flask itself. Numerous extensions provide database integration, form validation, upload handling, various open authentication technologies, and more. Flask may be "micro", but its ready for production use on a variety of needs.

Django verfolgt eher den Ansatz, diverse Funktionalität mitzubringen und den Entwicklern möglichst viel Arbeit abzunehmen – aus der Dokumentation:[5]

> - Django was designed to help developers take applications from concept to completion as quickly as possible.
>
> - Django includes dozens of extras you can use to handle common Web development tasks. Django takes care of user authentication, content administration, site maps, RSS feeds, and many more tasks – right out of the box.
>
> - Because Django was developed in a fast-paced newsroom environment, it was designed to make common Web-development tasks fast and easy.

Beide Ansätze haben ihre Vor- und Nachteile – sowohl generell, als auch für unseren spezifischen Anwendungsfall. Durch die im nächsten Abschnitt geschilderte Datenbank-Architektur hatten wir erst Flask bevorzugt, da es mehr Flexibilität bei der Wahl der Datenbank-Technologie bietet. Als wir jedoch auf das *django-haystack*-Projekt[6] stiessen, welches eine flexible Volltextsuche mit Django integriert, tendierte unsere Wahl wieder zu Django.

Wir haben uns schliesslich für Django entschieden – primär wegen django-haystack, aber auch, da unsere Applikation gut zu den von Django bereitgestellten Werkzeuge passt: Es war absehbar, dass wir diverse Funktionalität benötigen, welche in Django fix eingebaut ist, bei Flask aber zusätzliche Bibliotheken voraussetzt.

Im Englischen könnte man Django als *opinionated* bezeichnen: Viele Entscheidungen werden einem bereits vom Framework abgenommen. Dies führt zu gewissen Einbussen bei der Flexibilität, kann gleichzeitig aber auch sehr zeitsparend sein. Auch die sehr knappe Implementations-Zeit von etwa 7 Wochen sprach daher für diesen Entscheid.

### Datenbank

Bei der Wahl einer geeigneten Datenbank-Technologie bzw. -Struktur für Crashbin waren folgende Limitationen relevant:

[5] *Django overview*. URL: `https://www.djangoproject.com/start/overview/` (besucht am 04. 06. 2019).
[6] `https://github.com/django-haystack/django-haystack`

1. Wie in Abschnitt 4.1.2 erwähnt, ist es ein Ziel von Crashbin, möglichst einfach installierbar zu sein. Für kleinere Instanzen von Crashbin soll kein externer Datenbankserver benötigt werden.

2. Sollte eine leichtgewichtige Datenbank wie sqlite[7] bei einer steigenden Anzahl Reports/Usern nicht mehr ausreichen, so soll es möglich sein, Crashbin mit bestehenden, von Django unterstützten Datenbank-Servern (wie PostgreSQL, MySQL/MariaDB oder Oracle) zu verknüpfen. Wir wollen vermeiden, diese Auswahl unnötig einzuschränken, damit Crashbin auch bei grösseren Deployments einfach installier- und integrierbar bleibt.

3. Der Inhalt und die Struktur eines Crashreports sind je nach Applikation sehr unterschiedlich. Zwei verschiedene Applikationen nutzen vermutlich andere Bibliotheken, deren Versionsnummern in den Reports (durchsuchbar) festgehalten werden sollen. Wir brauchen also dynamische Spalten bzw. Datenstrukturen, was typischerweise mit SQL (und damit Django) nicht vorgesehen ist.

4. Debug-Logs in Crashreports können sehr lang werden, Crashbin soll dafür jedoch eine Volltextsuche anbieten, damit Reports einfach auffindbar sind. Eine Performance-Messung mit den bestehenden Reports für das qutebrowser-Projekt zeigt, dass eine simple SQL-Suche zu langsam ist: Eine Suche durch die rund 8000 Reports dauert 15-20 Sekunden[8].

Basierend auf diesen Überlegungen haben wir folgende Grundsatzentscheidungen getroffen:

- Um die einfachen Fälle abzudecken (kein Volltext-Index notwendig, keine dynamischen Spalten), nutzen wir die mit Django mitgelieferte Datenbank-Funktionalität, welche auf SQL aufbaut. So können wir von diversen Django-Funktionen wie die eingebaute Userverwaltung und den vielseitigen Datenbank-Support profitieren (Punkte 1 und 2). Alle Daten ausser die Crashreports selbst lassen sich so speichern.

- Wir evaluieren mögliche Lösungen für die obengenannten Probleme in der Design-Phase, beschränken uns aus Zeitgründen bei der Implementation jedoch auf SQL als Datenbankmodell. Daraus folgt, dass keine effiziente Volltextsuche durch Reports möglich ist. Weiterhin lassen sich keine dynamischen Spalten hinzufügen, sondern Crashreports bestehen (ausser der Mail-Adresse des Reporters) aus einem unstrukturierten Freitext.

Für Punkt 3 (dynamisch strukturierte Daten in Reports) sind folgende Lösungen denkbar:

- Dynamische Anpassung der Django-Models und der darunterliegenden SQL-Datenbank, wenn sich die gewünschten Felder in einem Report ändern. Es existieren zwar Projekte wie django-dynamic-models[9] oder dcolumn[10] zu diesem Zweck, diese werden jedoch kaum genutzt[11]. Bei einem solch tiefen Eingriff in die Funktionsweise von Django scheint diese Lösung eher heikel.

- Anlegen einer Datenbanktabelle mit einer Key- und einer Value-Spalte, welche so Key-/Value-Paare abspeichert und mit den entsprechenden Reports verknüpft. Bei der Anzeige von Reports können diese Daten mit einem SQL `JOIN` dazugeholt werden; bei einer

[7] `https://sqlite.org/`

[8] Gemessen auf einem Linux-Serversystem mit einer Intel Xeon E5504 CPU (4 Kerne + Hyperthreading, 2 GHz), HGST Ultrastar 7K4000 Festplatten (7200RPM), 16 GB RAM, MariaDB 10.3.14.

[9] `https://github.com/rvinzent/django-dynamic-models`

[10] `https://github.com/cnobile2012/dcolumn`

[11] Siehe Anhang C

Suche wird auf dieselbe Art der passende Report gefunden. Diese Lösung hat den Vorteil, bestehende Datenbank-Abstraktionen von Django problemlos nutzen zu können.

- Für die Reports zusätzlich eine NoSQL-Datenbank nutzen, in der dynamische JSON-artige Objekte gespeichert werden können. Aufgrund der Punkte 1 und 2 bietet sich hier tinydb[12] mit tinymongo[13] an: Damit würde Crashbin weiterhin ohne externen Datenbankserver laufen, da TinyDB ähnlich wie sqlite auf Files basiert. Wird mehr Performance benötigt, so kann dank tinymongo und pymongo[14] nahtlos auf MongoDB[15] umgestiegen werden, eine performante NoSQL-Datenbank als eigene Serverapplikation. TinyDB und MongoDB sind weit verbreitet[16], jedoch ist fraglich, inwiefern sie sich mit Django integrieren lassen. Tinymongo wird von relativ wenigen Projekten benutzt, jedoch handelt es sich dabei nur um einen kleinen Wrapper um TinyDB/PyMongo, der einfach auszutauschen wäre. Mit Quokka[17] gibt es ein Content-Management-System, das denselben Ansatz mit TinyDB/MongoDB verfolgt, jedoch auf Flask basiert.

Für Punkt 4 (effiziente Volltextsuche von Debug-Logs) haben wir folgende Lösungen in Betracht gezogen:

- Django bietet zwar eingebaute Unterstützung für Volltextsuchen, allerdings nur für PostgreSQL-Datenbanken[18]. Diese Lösung würde also Punkte 1 und 2 oben verletzen.
- Mit Haystack[19] existiert eine Django-Library, welche genau dieses Problem löst. Haystack lässt sich mit verschiedenen "Backends" verwenden und unterstützt dabei auch Whoosh[20], was eine solche Suche ohne externen Server erlaubt (siehe Punkt 1). Ist eine grössere Performance gewünscht, so kann stattdessen Solr[21], ElasticSearch[22] oder Xapian[23] als externer Service eingebunden werden.

### Weitere Python-Bibliotheken

Neben Django selbst verwenden wir in Crashbin folgende Python-Bibliotheken:

- *attrs*[24] um auf einfache Weise deklarativ Value-Klassen zu definieren. Von attrs inspirierte Funktionalität ist in Python 3.7 als `dataclasses` eingebaut, wir wollen jedoch Python 3.6 ebenfalls unterstützen.
- *django-colorful*[25] um für Labels ein `RGBColorField` verwenden zu können, welches eine entsprechende Farbauswahl im Browser zulässt. Als Alternative wäre *django-colorfield*[26]

[12] `https://github.com/msiemens/tinydb`
[13] `https://github.com/schapman1974/tinymongo`
[14] `https://pypi.org/project/pymongo/`
[15] `https://www.mongodb.com/`
[16] Siehe Anhang C
[17] `https://github.com/rochacbruno/quokka`
[18] `https://docs.djangoproject.com/en/2.2/topics/db/search/`
[19] `http://haystacksearch.org/`
[20] `https://whoosh.readthedocs.io/`
[21] `http://lucene.apache.org/solr/`
[22] `https://www.elastic.co/`
[23] `https://xapian.org/`
[24] `http://www.attrs.org/`
[25] `https://github.com/charettes/django-colorful`
[26] `https://github.com/jaredly/django-colorfield`

möglich gewesen – wir haben uns für django-colorful entschieden, da es eine Auswahl von vordefinierten "Lieblingsfarben" in der Konfiguration zulässt.

- *django-crispy-forms*[27], damit die von Django generierten HTML-Formulare den Bootstrap-Style benutzen.
- *django-mailbox*[28] als Bibliothek, um eingehende Mails zu empfangen.
- *Django REST framework*[29] für die Implementierung von HTTP REST APIs.

### 4.3.2. UI-Design

#### Konzept

Bei der Erstellung eines UI-Konzeptes ist es wichtig, Sinn und Zweck der Software sowie die Benutzergruppen zu berücksichtigen. Da es sich bei Crashbin um ein Arbeitstool für Entwickler handelt, sollte die Oberfläche möglichst simpel und effizient gestaltet sein. Aus diesem Grundgedanke sind folgende Überlegungen entstanden:

- Struktur: Sollte sich am Kernkonzept von Crashbin (Metapher mit Papierablage) und dessen Aufbau orientieren. Ist das Konzept bekannt, sollte somit auch die Navigation durch Crashbin intuitiv erfolgen.
- Orientierung: Bei der Benutzung von Crashbin sollte möglichst jederzeit ersichtlich sein, wo man sich in der Applikation befindet. Grundsätzlich hängt der Detaillierungsgrad von der Komplexität der Navigationsstruktur ab. Je mehr Ansichten vorhanden sind, desto genauer muss die Struktur angezeigt werden.
- Navigation: Sollte ermöglichen, dass von jedem Ort in Crashbin in die wichtigsten und meistverwendeten Ansichten gewechselt werden kann. So soll ein flüssiger Workflow garantiert werden.
- Verschachtelungstiefe: Ein hilfreiches Merkmal für die Ermittlung der Komplexität einer Applikation ist die Verschachtelungstiefe. Je mehr Ansichten durchlaufen werden müssen um eine bestimmte Operation auszuführen, desto mehr Zeit wird für die Suche von solchen Funktionen benötigt. Grundsätzlich gilt: Je weniger desto einfacher.
- Farbkonzept: Farben können bei Benutzeroberflächen auf verschiedene Arten eingesetzt werden. Auf Webseiten werden Farben oft für eine ansprechende visuelle Präsentation gebraucht. Bei Arbeitstools führt dies jedoch oft zu Ablenkungen und verschleiert wichtige Funktionen. Um dies zu verhindern, können Farben auch funktionell für die schnelle Erkennung wichtiger Bestandteile in der Ansicht eingesetzt werden.
- Layout: Die richtige Anordnung der verschiedenen Funktionsbausteinen in einer Ansicht trägt dazu bei, dass ein optimaler Workflow möglich ist. Wenn jedoch jede Ansicht ein komplett anderes Layout verwendet, erschwert dies das Zurechtfinden in der Applikation. Es sollten also möglichst ähnliche und gut gewählte Layouts wiederverwendet werden.

[27] `https://django-crispy-forms.readthedocs.io/`
[28] `https://django-mailbox.readthedocs.io/`
[29] `https://www.django-rest-framework.org/`

Anhand der oben genannten Gedanken haben wir für unser UI-Design folgende Grundsätze festgelegt:

- Struktur: Crashbin soll vier Hauptbereiche haben: Home, Bins, Reports und Labels. Bins, Reports und Labels sind wichtige Hauptbestandteile des Crashbin-Konzeptes, weshalb jedes Element für die Verwaltung eine eigene Hauptansicht bekommen soll. Die Home-Ansicht dient als Dashboard, wo wichtige Angaben zur aktuellen Situation übersichtlich dargestellt werden sollen.

- Orientierung: Aufgrund der niedrigen Verschachtelungstiefe entschieden wir uns, dass für die Orientierung des Benutzers nicht immer der genaue Pfad angezeigt werden soll. Hierfür reicht es aus, wenn die jeweilige Hauptkategorie (Home, Bins, Reports, Labels) angezeigt wird. Dies soll in der Menüleiste optisch hervorgehoben werden.

- Navigation: Anhand der Menüleiste kann zu jedem Zeitpunkt in eine der Hauptansichten gewechselt werden. Aufgrund der niedrigen Verschachtelungstiefe kann somit jede Funktion mit wenigen Klicks schnell erreicht werden. Zusätzlich sind wir der Meinung, dass die Suche von Bins, Reports und Labels zu den am meisten genutzten Funktionen von Crashbin gehört. Aus diesem Grund soll ein Suchfeld mit Bereichsauswahl immer in der Menüleiste angezeigt werden.

- Verschachtelungstiefe: Wir sind der Meinung, dass wie die Überschriften bei einer Dokumentation die Verschachtelungstiefe von vier Ebenen nicht überschritten werden sollte. Dies soll auch für Crashbin gelten.

- Farbkonzept: Crashbin soll auf aufwendige und ablenkende Darstellungen verzichten. Die Benutzeroberfläche ist deshalb hauptsächlich mit Grautönen zu designen. Wichtige Informationen, sowie Hauptfunktionen werden mit wenigen Akzentfarben hervorgehoben. Für die Labels kann der User die Farben selbst bestimmen. Hierbei soll ihm eine Auswahl der gebräuchlichsten Farben vorgeschlagen werden.

- Layout: Listen, Details sowie Settings sollen bei den verschiedenen Hauptelementen (Bins, Reports, Labels) möglichst gleich aussehen. Natürlich wird es inhaltliche Unterschiede geben, jedoch soll die grobe Anordnung gleich bleiben und ähnliche Funktionen sich im gleichen Bereich befinden.

Aufgrund der Benutzergruppen gehen wir davon aus, dass Crashbin hauptsächlich am Arbeitsplatz oder mobil auf einem Laptop gebraucht wird. Die Benutzung auf Smartphones soll möglich, aber nicht der Hauptfokus sein. Somit soll Crashbin zwar ein fluides Design, jedoch aufgrund der kleineren Bildschirmgrösse keine inhaltlichen Einschränkungen haben, damit die Ansicht nicht zu überladen wird.

### Wireframe

Wireframes werden benutzt, um einen frühen konzeptionellen Entwurf einer Webseite darzustellen. Hierbei geht es in erster Linie um die Konzeption und nicht um das Design. So wird sichergestellt, dass alle geforderten Funktionen vorhanden sind und diese in einem sinnvollen Layout angeordnet werden. Somit wurde anhand unserem Konzept für die wichtigsten Ansichten ein Wireframe erstellt. Nachfolgend ist als Beispiel das Wireframe der Bin-Detailansicht zu sehen. Alle erstellten Wireframes sind im Anhang E.1 zu finden.

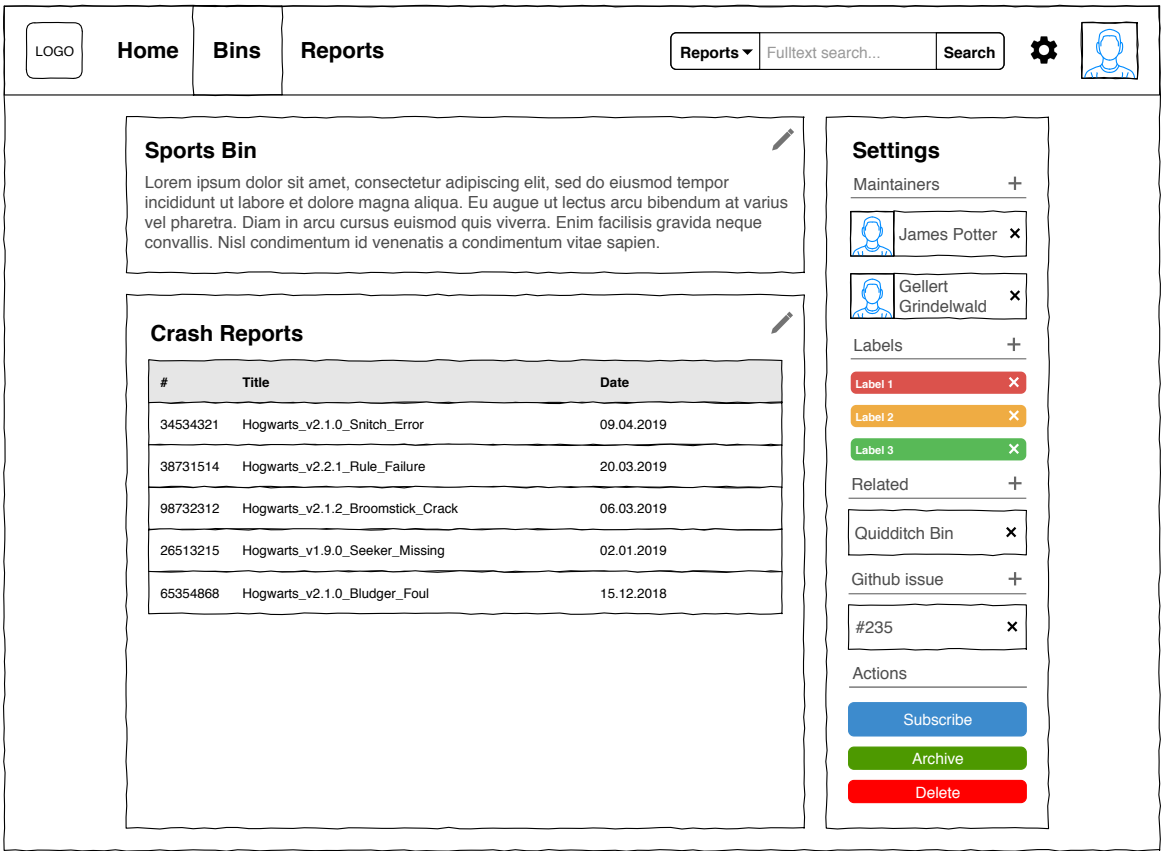



Abbildung 4.6.: Wireframe der Bin-Detailansicht

### Verwendete Frameworks

Bei der Auswahl des CSS-Frameworks wurde wie bei der Wahl des Web-Frameworks darauf geachtet, dass es sich um ein viel genutztes, gut dokumentiertes und gewartetes Framework handelt. Für eine erste Auswahl beschränkten wir uns auf uns bekannte CSS-Frameworks:

- Bootstrap: Für Anfänger und diejenigen, die ein robustes Frontend-Framework bevorzugen.
- Semantic-UI: Für Anfänger und diejenigen, die ein leichtes, wendiges Framework suchen.
- Foundation: Für Entwickler, die über einen angemessenen Erfahrungsschatz verfügen und sich in erster Linie mit der Entwicklung schneller, attraktiver und reaktionsschneller Websites befassen.
- Materialize: Für weniger erfahrene Entwickler, die eine Anleitung zu den Materialdesign-spezifikationen von Google benötigen.
- Material UI: Für Entwickler, die React verstehen und Erfahrung mit React haben und einen einfachen Weg benötigen, um die Richtlinien zur Materialgestaltung einzuhalten.
- Pure: Für Entwickler, die sich auf die Erstellung reaktionsschneller mobiler Websites konzentrieren.

- Skeleton: Für jemanden, der ein kleineres Projekt erstellt, das nicht alle Style-Komponenten eines grösseren Frameworks benötigt.
- UIKit: Für ziemlich erfahrene Entwickler aufgrund des derzeitigen Mangels an verfügbaren Ressourcen. Ansonsten ist es sowohl für einfache als auch für komplexe Projekte geeignet.
- Milligram: Für Entwickler, die ein kleines Projekt erstellen, das nicht viele Styling-Komponenten benötigt und ein CSS Flexbox Grid-System verwenden möchten.
- Susy: Für Jeden, der einzigartige oder spezielle Layoutanforderungen hat.

Anhand diversen Online Umfragen[30,31] zeigt sich schnell, dass in den vergangenen Jahren bis heute Bootstrap mit 30-40% am meisten genutzt wird. Bootstrap zeichnet sich besonders dadurch aus, dass es Responsive Web Design unterstützt und eine umfassende Dokumentation bietet. Aufgrund dessen wurde dieses Framework ausgewählt. Zusätzlich ist noch zu erwähnen, dass zugunsten einer schnellen Entwicklung auf den CSS-Präprozessor Sass verzichtet wurde. Sollte Crashbin mit weiteren Ansichten erweitert und ausgebaut werden, ist ein Refactoring mit Sass empfehlenswert.

## 4.4. Implementation und Tests

### 4.4.1. Django-Architektur

Bei der Implementation von Crashbin verfolgen wir die vom Django-Framework vorgegebene Architektur, welche in Abbildung 4.7 gezeigt wird.

Wird vom Browser eine Anfrage zur Crashbin-Applikation abgeschickt, so entscheidet Django aufgrund von "URL patterns", welche View für die Generierung der Seite zuständig ist. Einige Anfragen und Antworten werden ausserdem von Django's Middleware-Komponenten[32] abgeändert.

Eine View nutzt dann typischerweise weitere Django-Komponenten, um die gewünschte Seite zusammenzustellen und zurück an den Browser zu senden:

- Django Models um auf die Datenbank zuzugreifen. Django agiert hier als Object-Relational Mapper (ORM), kümmert sich also weitgehend automatisch um die Interaktion mit der konfigurierten SQL-Datenbank.
- HTML Templates, um den Inhalt einer Seite zu generieren. Dabei handelt es sich um normale HTML-Dateien, welche Platzhalter wie `{{ report.title }}` nutzen. Die Views reichen die dafür benötigten Daten an das Template weiter.

[30] Ashley Nolan. *The Front-End Tooling Survey 2018 - Results*. 25. Juli 2018. URL: `https://ashleynolan.co.uk/blog/frontend-tooling-survey-2018-results#css-frameworks` (besucht am 12. 06. 2019).

[31] Louis Lazaris. *The Results of The Ultimate CSS Survey*. 16. März 2016. URL: `https://www.sitepoint.com/results-ultimate-css-survey/` (besucht am 12. 06. 2019).

[32] Eine Art Plugin-System, bei dem jede Middleware eine Anfrage oder Antwort verändern kann, bevor diese weitergereicht wird. So existiert beispielsweise die `SecurityMiddleware`, die diverse HTTP-Header hinzufügt, um gewisse moderne Sicherheitsfunktionen in Browsern zu aktivieren.

- Django Forms, um Formulare basierend auf den Models zu generieren (etwa um einen Bin zu editieren). Django kümmert sich automatisch um eine entsprechende Validierung der Benutzereingaben.

### 4.4.2. Dateistruktur

In den Tabellen 4.1 und 4.2 wird die Dateistruktur des Crashbin Git-Repositories aufgezeigt. Dabei wurden folgende Dateien ausgelassen:

- Leere `__init__.py`-Dateien. Diese werden von Python benötigt, um einen Ordner als Python-Package zu markieren.
- Django-Migrationen, die automatisch von Django generiert wurden: `crashbin_app/migrations/`
- Die genauen Inhalte von Frameworks von Drittanbietern: `crashbin_app/static/css/bootstrap/`, `crashbin_app/static/fontawesome/`, `crashbin_app/static/js/`.

Um beim Projektstart ein für Django passendes Grundgerüst zu erstellen, wurde `django-admin startproject` benutzt. Das `django-admin`-Tool wird als Teil von Django mitgeliefert.

### 4.4.3. URLs, Views und Templates

Tabelle 4.3 zeigt die von Crashbin definierten URLs sowie die damit verknüpften Views und Templates. In Tabelle 4.4 sind die einzelnen Views genauer beschrieben.

Wo sinnvoll wurden Views und Templates mehrfach verwendet, um duplizierten Code zu vermeiden. So sind beispielsweise die Formulare für Bins und Labels sehr ähnlich und nutzen beide dasselbe `form.html`-Template.

Views, welche als "nur POST" gekennzeichnet sind, werden nur für HTTP POST-Requests genutzt, also nur, um Daten vom Browser zu empfangen. Deshalb ist dort kein Template vorhanden.

Einige URL-Bereiche (alle URLs unter `accounts/`, `admin/` und `api/`) werden an andere Komponenten bzw. Bibliotheken delegiert. Die Platzhalter `<id>` sowie `<name>` bezeichnen eine eindeutige Objekt-ID bzw. der Name der zu ändernden Einstellung (`maintainer`, `label`, `related` oder `bin`).

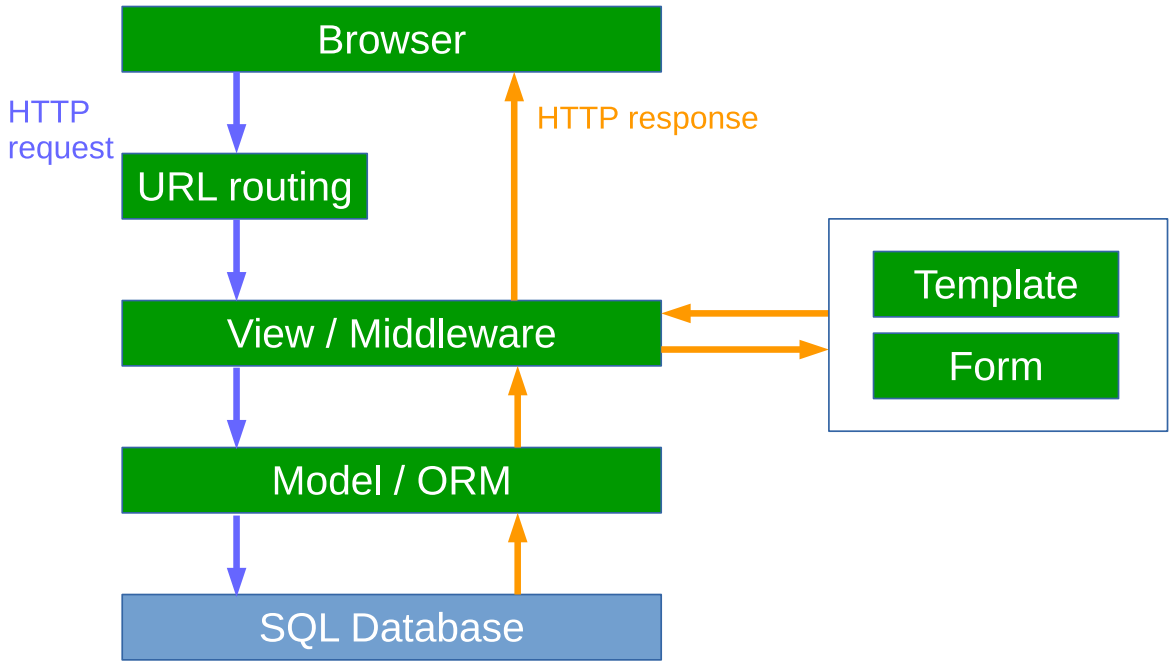



Abbildung 4.7.: Architektur von Django, Public-Domain, Fleschenberg René. Juni 2017. URL: `https://fleschenberg.net/django-architecture-diagram/` (besucht am 07. 06. 2019)

| Datei | Beschreibung |
|---|---|
| .coveragerc | Konfigurationsdateien für die entsprechenden Entwicklungswerkzeuge, siehe Abschnitte 4.4.4 und 4.4.6. |
| .flake8 | |
| .pylintrc | |
| .travis.yml | |
| mypy.ini | |
| pytest.ini | |
| tox.ini | |
| crashbin_settings_example.py | Beispielkonfiguration für Crashbin |
| manage.py | Von Django bereitgestelltes Skript für die Entwicklung (Development-Server starten, etc.). |
| LICENSE | Eine Kopie der MIT-Lizenz: `https://choosealicense.com/licenses/mit/`. |
| requirements.txt | Alle Abhängigkeiten (rekursiv, inkl. genauen Versionen), damit die von uns genutzte Umgebung reproduzierbar ist. |
| setup.py | setuptools-Datei, um Crashbin als Python-Package zu installieren. |
| crashbin_app/ | Die Django-Applikation für Crashbin. Die Dateien in diesem Ordner sind in Tabelle 4.2 aufgelistet. |
| crashbin_site/ | Eine Django-Site, die mehrere Applikationen beinhalten kann. |
| crashbin_site/settings.py | Diverse Django-Einstellungen. Einige davon können mit einer `crashbin_settings.py` Datei überschrieben werden, andere (wie die zu verwendenden Django-Libraries) sind fix für Crashbin angepasst. |
| crashbin_site/urls.py | URL-Routing-Definitionen. Definiert, dass `/admin` auf das Django-Admin-Webinterface zeigen soll, alle anderen URLs von der Crashbin-Applikation behandelt werden. |
| crashbin_site/wsgi.py | Von Django bereitgestellt, wird für das Deployment mit einem Webserver via WSGI benötigt. |
| tests/ | Mit pytest verfasste automatisierte Tests. |

Tabelle 4.1.: Dateistruktur des Crashbin-Repositories

| Datei | Beschreibung |
|---|---|
| admin.py | Definition von Model-Klassen, die im Django-Admin-Interface verfügbar sein sollen. |
| apps.py | Django AppConfig-Klasse für Crashbin. Wird benötigt für die Initialisierung beim Start des Servers. |
| forms.py | Definition von Formularen und den darin enthaltenen Feldern. |
| models.py | Definition aller Klassen, die von Django in der Datenbank abgespeichert werden. |
| signals.py | Definition von Signalen, die für Plugins verfügbar sind. Plugins können Callbacks mit solchen Signalen verbinden (siehe die Softwaredokumentation in Kapitel 5). |
| urls.py | URL-Routing-Definitionen, mit denen URLs zu Views gebunden werden. |
| utils.py | Diverse Klassen und Funktionen, die von anderem Code benötigt werden. |
| api/ | Implementation der HTTP API, die von Crashbin selbst (via JavaScript) und externen Applikationen genutzt werden kann. |
| . . . /serializers.py | Definition von Serializern, die Daten aus den Models als JSON aufbereiten. |
| . . . /urls.py | URL-Routing-Definitionen für alle URLs unter `/api`. |
| . . . /views.py | API Views, die Serializer mit den entsprechenden Daten verknüpfen. |
| migrations/ | Django "Migrations", die bei Änderungen an der Datenbankstruktur benötigt werden, um bestehende Datenbanken anzupassen. |
| static/ | Statische Web-Ressourcen. |
| . . . /css/main.css | CSS Stylesheet für Crashbin. |
| . . . /css/bootstrap/ | Das Bootstrap CSS-Framework. |
| . . . /fontawesome/ | Die FontAwesome Icon-Bibliothek. |
| . . . /img/ | Das Crashbin-Logo für die Webapplikation. |
| . . . /js/ | JavaScript-Bibliotheken für Bootstrap (jQuery, Popper). |
| templates/ | HTML-Templates, die von den Views genutzt werden. |
| templatetags/ | Eigene Tags wie `{% bin_list %}`, die innerhalb von HTML-Templates genutzt werden. |
| . . . /components.py | HTML-Komponenten, die von mehreren Templates gebraucht werden. |
| . . . /utils.py | Diverser Python-Code für Funktionen, die nicht direkt in Templates implementierbar sind. |
| views/ | Views für die verschiedenen Unterseiten, die via `urls.py` mit URLs verknüpft werden, und die Templates benutzen. |

Tabelle 4.2.: Dateistruktur unterhalb von crashbin_app/

| URL | View | Template |
|---|---|---|
| / | misc_views.home | home.html |
| search/ | misc_views.search_dispatch | nur POST |
| accounts/* | Django User-Accounts | |
| admin/* | Django Admin-Interface | |
| api/* | Django REST framework | |
| bins/ | bin_views.bin_list | bins.html |
| bin/new/ | bin_views.bin_new_edit | form.html |
| bin/<id>/ | bin_views.bin_detail | bin_detail.html |
| bin/<id>/edit/ | bin_views.bin_new_edit | form.html |
| bin/<id>/subscribe/ | bin_views.bin_subscribe | nur POST |
| bin/<id>/settings/<name>/ | setting_views.settings | nur POST |
| labels/ | label_views.label_list | labels.html |
| labels/new/ | label_views.label_new_edit | form.html |
| labels/<id>/edit/ | label_views.label_new_edit | form.html |
| reports/ | report_views.report_list | reports.html |
| report/<id>/ | report_views.report_detail | report_detail.html |
| report/<id>/reply/ | report_views.report_reply | nur POST |
| report/<id>/settings/<name>/ | setting_views.settings | nur POST |

Tabelle 4.3.: URLs und dazugehörige Django-Views/Templates

| View | Beschreibung | Screenshot |
|---|---|---|
| `misc_views.home` | Hauptansicht mit Dashboard. | E.6 (S. 98) |
| `misc_views.search_dispatch` | Wird für die Suche in der Navigation benötigt, leitet auf `bin_list` oder `report_list` weiter. | |
| `setting_views.settings` | Wird via JavaScript aufgerufen, um Einstellungen (Labels, etc.) für Bins/Reports zu setzen. | E.14 (S. 102) |
| `bin_views.bin_list` | Liste aller Bins. | E.7 (S. 98) |
| `bin_views.bin_new_edit` | Formular um einen neuen Bin anzulegen, oder einen bestehenden Bin zu editieren. | E.9 (S. 99) |
| `bin_views.bin_detail` | Detailansicht eines Bins. | E.8 (S. 99) |
| `bin_views.bin_subscribe` | Wird via JavaScript aufgerufen, um einen Bin zu abonnieren. | |
| `label_views.label_list` | Liste aller Labels. | E.12 (S. 101) |
| `label_views.label_new_edit` | Formular um ein neues Label anzulegen, oder ein bestehendes Label zu editieren. | E.13 (S. 101) |
| `report_views.report_list` | Liste aller Reports. | E.10 (S. 100) |
| `report_views.report_detail` | Detailansicht eines Reports. | E.11 (S. 100) |
| `report_views.report_reply` | Wird aufgerufen, wenn eine neue Nachricht zu einem Bin abgesendet/gespeichert werden soll. | |

Tabelle 4.4.: Beschreibung der Views

### 4.4.4. Automatische Testverfahren

#### Wahl des Testframeworks

Für Python existieren folgende Testframeworks, welche auch in der Python-Dokumentation verzeichnet sind[33]:

- unittest.py (PyUnit): Wird mit Python mitgeliefert. Die Syntax für Assertions ist dabei von Java bzw. xUnit inspiriert und im Vergleich zu pytest/nose sehr schwerfällig.
- nose[34]: Wird laut der Webseite nicht mehr weiterentwickelt: *Nose has been in maintenance mode for the past several years and will likely cease without a new person/team to take over maintainership. New projects should consider using Nose2, py.test, or just plain unittest/unittest2.*
- nose2[35]: Offizieller Nachfolger von nose. Obwohl nose2 schon seit 2010 in Entwicklung ist, ist es im `setup.py` weiterhin als Beta markiert und wird von vergleichsweise wenigen Projekten genutzt (siehe Anhang C).
- pytest[36]: Sehr reifes Projekt (2004 als Teil von PyPy[37] enstanden). Trotz seines Alters wird pytest sehr aktiv weiterentwickelt von einer Gruppe aus ca. 20 Core-Maintainern[38].

Für unittest.py liefert Django selbst ein `django.test`-Modul, um Django-Projekte einfacher testen zu können. Eine solche Integration ist aber auch für alle genannten Frameworks in Form von externen Plugins verfügbar.

Nose bzw. nose2 scheiden wegen ihrer unklarer Zukunft aus. Unittest.py bietet gegenüber pytest fast nur einen Vorteil: Es wird mit Python mitgeliefert, während pytest als externe Abhängigkeit nachinstalliert werden muss. Dies stellt jedoch kein Problem dar, gerade durch die weite Verbreitung von pytest.

Weiterhin ist Florian Bruhin einer der Maintainer von pytest. Er hat so einen kurzen Draht zum Projektteam für den unwahrscheinlichen Fall, dass damit Probleme auftauchen sollten. Da er schon in verschiedenen Firmen Kurse gegeben hat sowie als "technical reviewer" in einem Buch[39] zum Thema mitgearbeitet hat, besitzt er ein umfassendes Vorwissen zu pytest.

Aus diesen Gründen entschieden wir uns für pytest, um unsere Tests zu implementieren.

Wir nutzen zusätzlich folgende pytest-Plugins:

- pytest-django[40] für die Integration mit Django
- pytest-cov[41] für die Integration von code coverage (siehe Seite 49)

[33] `https://docs.python.org/3/library/unittest.html`
[34] `https://nose.readthedocs.io/`
[35] `https://nose2.io/`
[36] `https://pytest.org/`
[37] eine alternative Python-Implementation mit einem JIT, siehe `http://pypy.org/`
[38] `https://github.com/orgs/pytest-dev/teams/core/members`
[39] B. Okken. *Python Testing with pytest: Simple, Rapid, Effective, and Scalable*. Pragmatic Bookshelf, 2017. ISBN: 9781680504408.
[40] `https://pytest-django.readthedocs.io/`
[41] `https://github.com/pytest-dev/pytest-cov`

- pytest-mock[42] für die Integration von `unittest.mock`[43]

### Arten von Tests

Grundsätzlich sind für Django-Projekte drei Arten von Tests vorstellbar, welche der weit verbreiteten "Testing Pyramid"[44] entsprechen:

- End-to-end Tests (UI-Tests, Systemtests), die mittels Tools wie Selenium[45] einen Web-Browser fernsteuern und so einen Nutzer imitieren.
- Integration Tests (Service Tests, Subcutaneous Tests), die mit einer Bibliothek wie requests[46] HTTP-Requests gegen einen Webserver ausführen.
- Unit Tests, welche die Funktionen (Model-Methoden, Views, Utilities, etc.) direkt aufrufen.

Wir entschieden uns, uns vorerst nur auf Unit Tests zu beschränken. Einige Abläufe sind so nicht komplett testbar (sondern nur zerlegt in einzelne Komponenten), aber aufgrund der knappen Zeit erschien uns dies als eine sinnvolle Einschränkung. Sollten Fehler existieren, die so nicht entdeckt werden, sollten diese durch manuelle Tests oder während den Usability-Tests gefunden werden.

### Code Coverage

Eine hohe Testcoverage bedeutet nicht unbedingt, dass eine Codebasis auch wirklich gut getestet ist – gerade da bei Django viele Dinge deklarativ definiert werden, kann es vorkommen, dass Codezeilen abgedeckt und trotzdem ungetestet sind.

Trotzdem ist die Coverage ein nützliches Werkzeug, um zu erfahren, wo noch keine ausreichenden Tests existieren. Wir nutzen dafür das Tool coverage.py[47] sowie das pytest-cov[48]-Plugin.

Standardmässig wird von coverage.py nur die Line-Coverage gemessen, wir haben zusätzlich die Messung der Branch-Coverage aktiviert. Wird beispielsweise folgende Funktion definiert:

```
1 def fn(x):
2     if x:
3         y = 10
4     return y
```

und mittels `fn(1)` getestet, so sind bei normaler Line-Coverage alle Zeilen abgedeckt. Trotzdem erzeugt die Funktion mit `fn(0)` einen `UnboundLocalError`, da das `if` nicht ausgeführt wird. Mit aktivierter Branch-Coverage wird das `if` stattdessen als "partially covered" markiert, da dieser Pfad in den Tests nicht berücksichtigt wird.

[42] `https://github.com/pytest-dev/pytest-mock/`
[43] `https://docs.python.org/3/library/unittest.mock.html`
[44] Siehe etwa `https://martinfowler.com/bliki/TestPyramid.html`, ursprünglich bekannt geworden durch M. Cohn. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Signature Series (Cohn). Pearson Education, 2009. ISBN: 9780321660565.
[45] `https://www.seleniumhq.org/`
[46] `http://python-requests.org/`
[47] `https://coverage.readthedocs.io/`
[48] `https://github.com/pytest-dev/pytest-cov`

Um mitzuverfolgen, wie sich die Coverage nach Code-Änderungen verändert, nutzen wir den Service von `https://codecov.io/`, der Coverage-Tools mit GitHub integriert.


Abbildung 4.8.: Beispiel eines Kommentars von Codecov.io

Wir haben darauf verzichtet, uns auf eine bestimmte minimal-Coverage festzulegen. Stattdessen haben wir festgelegt, dass wir Tests überall da implementieren, wo dies ohne grossen Aufwand möglich ist. Daraus resultierte eine Coverage von über 99% mit 126 Tests, nur einzelne Zeilen waren nicht mühelos automatisiert testbar. Ein Log der Ausgabe ist in Anhang D sichtbar.

### 4.4.5. Manuelle Testverfahren

#### Manuelle Tests

Manuelle Tests sollen ergänzend zu den automatischen Testverfahren möglichst systemübergreifend angewendet werden, damit Schnittstellen und schwierig testbare Bereiche überprüft werden können. Hierfür wurde die Benutzeroberfläche während der Entwicklung fortlaufend manuell von den Entwicklern selbst bei Codereviews getestet. Für die Tests wurden Szenarien von den Use Cases durchgespielt. Nach Feature-Freeze ergänzten wir Crashbin zusätzlich mit Beispieldaten von qutebrowser. Dies ermöglichte uns Crashbin mit einer grösseren und realistischen Datenmenge auszutesten. Hierbei sind uns folgende Mängel aufgefallen:

1. Wenn bei Dropdowns der Button nicht mehr im Fokus des Mauszeigers war, jedoch die Auswahlmöglichkeiten noch angezeigt wurden, wechselte sich die Textfarbe der Auswahl auf weiss und war wegen dem weissen Hintergrund nicht mehr erkennbar.

2. Aufgrund Änderungen im HTML funktionierte die Filterfunktion bei den Settings nicht mehr.

3. Bei der Listendarstellung von Bins und Reports wurden die Elemente standardmässig nach Erstellungsdatum sortiert. Dies verunmöglichte es, schnell in der Liste ein bestimmtes Element zu finden. Eine alphabetische Sortierung ergibt hier mehr Sinn.

4. In Listendarstellungen wurden vom ersten Element immer die Details angezeigt. Dies sollte den User darauf hinweisen, dass mehr Informationen zu den Elementen angezeigt werden können. Das erwies sich jedoch als störend und unnötig, da man in den meisten Fällen nicht am ersten Element interessiert ist.

5. Bei der Inbox wurden zu viele Einstellungsoptionen angezeigt, womit das Bearbeiten, Archivieren und Löschen möglich war. Crashbin braucht jedoch zu jederzeit eine Inbox, damit eintreffende Reports darin abgelegt werden können.

6. Die Anordnung der Buttons war nicht in der ganzen Applikation konsequent gleich. Dies führte zu ungewollten Aktionen, da man aus Versehen die falschen Buttons anwählte.

7. Es können nicht mehrere Reports auf einmal einem Bin zugeordnet werden. Bei grösseren Datenmengen ist das einzelne Hinzufügen von Reports sehr zeitaufwendig. Zusätzlich wäre ein Shortcut für die Bin-Zuordnung in der Listendarstellung sehr hilfreich.

8. Die Darstellung des Debug-Logs ist aufgrund der vielen Informationen zu unübersichtlich. Diese sollten besser strukturiert werden.

Die Punkte 1-6 erschienen uns als wichtig und mittels eines vernünftigen Arbeitsaufwand korrigierbar. Somit wurden diese Mängel während der Testingphase fortlaufend behoben. Die letzten zwei Punkte wurden aus folgenden Gründen nicht korrigiert:

- Punkt 7: Durch das Zuordnen von mehreren Reports wird für jeden Report eine einzelne Notification ausgelöst. Da diese jedoch alle den gleichen Bin betreffen, ist eine Bündelung dieser Notifications sinnvoll. Diese Änderung erfordert einen erhöhten Aufwand und wurde von uns niedrig priorisiert, da unserer Meinung nach in den meisten Fällen nur ein Report zugeordnet werden muss. Auf den Shortcut wurde aus zeitlichen Gründen verzichtet.

- Punkt 8: Der Log eines Reports ist sehr individuell und kann je nach Software komplett anders aussehen. Um allen Anwendern gerecht zu werden, müsste die Darstellung des Logs konfigurierbar sein. Diese Änderung erfordert auch einen erheblichen Mehraufwand und wurde wegen des knappen Zeitplans niedrig priorisiert.

### Usability Tests

Für die Usability Tests wurde eine Datenbank mit realen Beispieldaten von qutebrowser vorbereitet. Diese umfasst 8 Users, 23 Bins, 53 Reports und 33 Labels. Die Tests wurden bei 6 Probanden durchgeführt, da sich bei dieser Anzahl das beste Nutzungs/Aufwandverhältnis ergibt.[49]

Vor dem UX-Test wurde dem Probanden erklärt, wobei es sich um Crashbin handelt und was dessen Konzept (Papierablagesystem) ist. Hierbei wurde bei der Erklärung komplett auf das von den Entwicklern definierte Vokabular verzichtet. Für die Erklärung und Aufgabenstellung

[49] Jakob Nielsen und Thomas K. Landauer. „A Mathematical Model of the Finding of Usability Problems“. In: *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*. CHI '93. Amsterdam, The Netherlands: ACM, 1993, S. 206–213. ISBN: 0-89791-575-5. DOI: 10.1145/169059.169166. URL: http://doi.acm.org/10.1145/169059.169166.

wurden nur Deutsche Synonyme verwendet. Dies soll verhindern, dass die Probanden ohne zu Überlegen in der Ansicht nur nach den geforderten Keywords suchen und darauf klicken. Zusätzlich sind alle Probanden dazu aufgefordert worden, während des Tests bei unbekannten Englischen Begriffen nachzufragen.

Bei den zu lösenden Aufgaben wurde darauf geachtet, dass die gewünschten Aktionen nicht nur in einer Ansicht stattfinden, sondern die Probanden dazu gezwungen sind, gelegentlich das Menü zu wechseln. Dies macht ersichtlich, ob immer über die Navigationsleiste die Ansicht gewechselt wurde, oder ob sie auch die vorhandenen Shortcuts nutzten. Die gestellten Aufgaben lauteten wie folgt:

1. Neuen Bin erstellen
2. Maintainer, Label, Related Bin definieren
3. Erstellten Bin abonnieren
4. Alle abonnierten Bins anzeigen lassen
5. Erstellten Bin wieder löschen
6. Log eines bestimmten Reports anzeigen lassen
7. Diesen Report einem anderen Bin zuordnen
8. Notiz erstellen
9. Eine Antwort senden
10. Zugewiesener Bin archivieren
11. Bei einem bestimmten Label die Farbe ändern
12. Neues Label erstellen
13. Erstelltes Label einem Bin zuordnen

Sämtliche Probanden haben alle Aufgaben erfolgreich und in einem angemessenen Zeitrahmen erledigt. Auf Anfrage wurden noch folgende Verbesserungsvorschläge angegeben:

- Listenelemente müssen direkt geöffnet werden können. Zur Zeit benötigt dies zwei Klicks.
- Das Suchfeld in der Navigationsliste braucht eine neue Kategorie “All”, welche standardmässig ausgewählt ist. Mit dieser Option soll in allen Elementen (Bin, Report und Label) gesucht werden. Die Suche soll jedoch weniger tiefgründig erfolgen als wenn nach einem spezifischen Element gesucht werden soll.
- Das Suchfeld in der Navigationsliste soll die angewählte Option optisch hervorheben, damit diese nicht übersehen wird.
- Die Übersicht im Home-Menü beinhaltet unwichtige Informationen. Stattdessen sollen die neusten Reports angezeigt werden. Noch besser: Die zu anzeigenden Informationen sollen konfigurierbar sein.
- In der Navigationsleiste sollte sich ein Shortcut für die Inbox befinden.

- Nur die verantwortlichen Maintainer können ihren Bin mit den dazugehörigen Reports bearbeiten und löschen.
- Autocompletion für die Suchfelder.
- Notizen im Report soll man Bearbeiten und Löschen können.
- Notizen sollen Checkboxes unterstützen.
- Bei den Messages soll die Überschrift "History" stehen.

### 4.4.6. Code-Qualität und -Style

Zusätzlich zu den Tests nutzen wir diverse weitere Tools, um eine hohe Code-Qualität zu gewährleisten:

#### pylint

Das *pylint*[50]-Projekt kombiniert verschiedene Funktionalitäten in einem Werkzeug, welche die Code-Qualität erhöhen. Es hilft bei der Einhaltung von Code-Guidelines (Zeilenlänge, Variablennamen, unbenutzte Imports etc.), kann aber auch Fehler finden und enthält verschiedene Metriken (Anzahl Klassen, Argumente, Members etc.). Für die Integration mit Django nutzen wir das pylint-django[51] plugin.

Pylint ist standardmässig so konfiguriert, dass sehr viele Checks eingeschaltet sind, die aber oft viel zu strikt sind[52]. Deshalb machen wir in der Konfiguration von pylint einige Anpassungen:

- `missing-docstring` wird deaktiviert, da manche kleinere Funktionen aus unserer Sicht keinen Docstring benötigen.
- `locally-disabled` und `suppressed-message` werden deaktiviert, da es sich dabei nur um Info-Ausgaben handelt.
- `too-many-anchestors` wird deaktiviert, da die Vererbungstiefe von einigen Klassen von Django vorgegeben ist und wir darauf keinen Einfluss haben.
- `too-few-public-methods` wird deaktiviert, da in Django vieles deklarativ passiert, und eine Klasse daher nicht zwingend Methoden haben muss, um nützlich/sinnvoll zu sein.
- `cyclic-import` wird deaktiviert, weil gewisse zyklische Imports mit Django fast nicht vermieden werden können.
- `bad-continuation` wird deaktiviert, weil es mit black (siehe unten) kollidiert.
- Konstanten müssen nicht zwingend upper-case Namen haben, damit etwa Plugin-Signale `new_report` und nicht `NEW_REPORT` heissen.

[50] `https://pylint.org/`
[51] `https://github.com/PyCQA/pylint-django`
[52] siehe die pylint-Diskussion "Getting sane", `https://github.com/PyCQA/pylint/issues/746`

- Argumente haben keine Mindestlänge, so dass etwa `pk` ("primary key") als Argument erlaubt ist.

### flake8

*flake8*[53] ist ein weiteres bekanntes Tool. Der Fokus liegt hier eher auf einer schnellen Analyse für "offensichtliche" Probleme, flake8 macht im Gegensatz zu pylint keine tiefergehende statische Code-Analyse. Es gibt zwar einige Überschneidungen mit pylint, trotzdem macht es Sinn, beide Tools zu verwenden, da flake8 doch einige nützliche Checks hat, die in pylint nicht vorhanden sind. Ausserdem kann es durch die schnelle Ausführung gut in der Entwicklungsumgebung eingebunden werden, um sofortiges Feedback zu kriegen. Für die Integration mit Django nutzen wir das flake8-django[54] plugin.

In der Konfiguration von flake8 schalten wir E203 (*Whitespace before ':'*) und E501 (*Line too long*) aus, da diese mit black kollidieren (siehe nächster Abschnitt).

### black

Bei *black*[55] handelt es sich um einen auto-formatter für Python. Black orientiert sich an den PEP8-Guidelines[56], die ursprünglich für Python selbst geschrieben wurden, aber von fast allen Python-Projekten beachtet werden. Von Black formatierter Code besteht ebenfalls weitgehend die Style-Checks von pylint und flake8, da diese ebenfalls auf PEP8 basieren. Es gibt wenige Ausnahmen, bei denen sich die Tools in ihrer Interpretation von PEP8 uneinig sind – für diese Fälle empfiehlt die black-Dokumentation, zwei pylint/flake8-Checks in dessen Konfiguration zu deaktivieren, was wir so umgesetzt haben. Zwar ist Black noch relativ neu (und ist als Beta markiert), doch viele grosse Python-Projekte (darunter auch Django oder pytest) haben es in den letzten Monaten adoptiert.

### Type annotations

Mittels sogenannter *type annotations* (auch bekannt als *gradual typing*) wird die Typensicherheit im Code geprüft. Ähnlich zu Typescript erzwingt die Sprache selber keine expliziten Typendeklarationen (*dynamic typing*), jedoch sind diese valide Python-Syntax und werden einfach ignoriert. Dadurch ist kein separater "pre-compiler" wie bei Typescript nötig, jedoch ein externes Tool zur Überprüfung der Typen.

```
def calc_primes(start: int) -> List[int]:
    primes: List[int] = []
    ...
```

Listing 4.1: Beispiel für Typenannotationen

[53] `http://flake8.pycqa.org/`
[54] `https://github.com/rocioar/flake8-django`
[55] `https://github.com/python/black/`
[56] `https://www.python.org/dev/peps/pep-0008/`

Wir benutzen dazu das Tool *mypy*[57]. Ursprünglich im Rahmen einer Doktorarbeit an der Universität Cambridge entwickelt, ist es mittlerweile ein etabliertes Werkzeug und wird durch Firmen wie Dropbox benutzt und weiter entwickelt.

Als Alternativen zu mypy existieren *pyright*[58] (Microsoft), *pyre*[59] (Facebook) und *pytype*[60] (Google). Diese scheinen jedoch stark auf die Bedürfnisse der jeweiligen Firmen ausgerichtet zu sein, während hinter mypy eine grössere Community steht. Ein Test der Alternativen mit Crashbin führte zu zahlreichen false-positives sowie internen Tool-Fehlern für Fälle, die mypy korrekt behandelte.

Für eine komplette Überprüfung einer Codebasis braucht mypy möglichst vollständige Typeninformationen für die genutzten Bibliotheken. Für die Python Standard-Library sind diese in Form des typeshed-Projektes[61] in mypy enthalten; für einige Bibliotheken existieren separate Projekte, welche diese Information in Form von `.pyi`-Dateien ("stubs") mitliefern, ähnlich einer Header-Datei bei C-Projekten.

Wir nutzen *django-stubs*[62] sowie *djangorestframework-stubs*[63] um Typeninformationen zu den von uns benutzten Bibliotheken zu erhalten. Für django-mailbox bzw. -colorful existieren keine solchen Stubs, weshalb wir mypy so konfiguriert haben, dass diese Module ignoriert werden.

In der Konfiguration für mypy aktivieren wir diverse Optionen, damit mypy striktere Checks durchführt:

- `disallow_subclassing_any` damit eine Vererbung vom `Any`-Typ (der aussagt, dass ein Typ nicht näher bekannt ist) verboten ist.
- `disallow_untyped_calls` und `disallow_untyped_defs` damit sichergestellt wird, dass die gesamte Codebasis von Crashbin Typenannotationen besitzt, und auch neuer Code mit solchen versehen wird.
- `disallow_incomplete_defs` damit bei allen Argumente von Funktionen Typen vorhanden sein müssen.
- `check_untyped_defs` damit auch Definitionen ohne Typannotationen geprüft werden. Dies wäre aufgrund von `disallow_untyped_defs` nicht unbedingt nötig, macht die Ausgabe aber vollständiger während der Entwicklung von Code, bei dem solche Annotationen noch fehlen.
- `disallow_untyped_decorators` damit keine Python-Decorators[64] die Typinformationen von Funktionen verwerfen.
- `warn_redudant_cast` damit unnötige Aufrufe von `typing.cast(...)` erkannt werden.
- `warn_unused_ignores` damit unnötige `# type: ignore`-Kommentare erkannt werden.
- `strict_equality` damit vor ==-Vergleichen von inkompatiblen Typen gewarnt wird.

[57] `http://mypy-lang.org/`
[58] `https://github.com/Microsoft/pyright`
[59] `https://pyre-check.org/`
[60] `https://google.github.io/pytype/`
[61] `https://github.com/python/typeshed/`
[62] `https://github.com/mkurnikov/django-stubs`
[63] `https://github.com/mkurnikov/djangorestframework-stubs`
[64] `https://docs.python.org/3/glossary.html#term-decorator`

Einige Optionen, die mypy standardmässig mit der `--strict`-Option verwenden würde, haben wir jedoch nicht aktiviert:

- `warn_unused_configs` würde eigentlich bei nicht benötigten dateispezifischen Optionen im Config-File eine Warnung generieren, generiert jedoch wegen eines Mypy-Bugs[65] false-positives.
- `disallow_any_generics` würde Annotationen wie `Sequence` (statt einer genaueren Angabe wie `Sequence[str]`) verbieten, produziert jedoch false-positives, welche wir aus Zeitgründen nicht genauer untersuchen konnten.
- `no_implicit_optional` verlangt, dass in Fällen wie `def f(x: str = None)` (also ein Argument mit `None` als default-Wert) stattdessen `def f(x: Optional[str] = None)` geschrieben wird. Wir erachten die erste Form als kompakter und die Intention ist klar ersichtlich, weshalb wir auf diese Option verzichten.
- `warn_return_any` würde es Funktionen verbieten, einen unbekannten Typ zurückzugeben. Wir müssen jedoch Objekte zurückgeben, die von Django-Mailbox stammen. Da für Django-Mailbox keine Typinformationen vorhanden sind, haben wir diese Option nicht aktiviert.

### 4.4.7. Continuous Integration (CI)

#### Task-Automatisierung mit tox

Wir nutzen tox[66] um die verwendeten Tools (pytest, mypy, black, etc.) auf einfache Weise ausführen zu können. In der tox-Konfiguration lassen sich "environments" (Umgebungen) definieren; für jede Umgebung werden Python-Abhängigkeiten sowie das auszuführende Kommando angegeben.

Wird tox ausgeführt, so erstellt es für jede Umgebung ein Python "virtual environment" mittels *virtualenv*[67]. Dabei handelt es sich um eine lokale, isolierte Kopie der Python-Installation, so dass installierte Abhängigkeiten andere Projekte nicht beeinflussen. Danach werden Crashbin und weitere Abhängigkeiten installiert und die definierten Kommandos ausgeführt.

Für Crashbin haben wir die folgenden Tox-Umgebungen definiert:

**pyXY** führt die Testsuite (siehe Abschnitt 4.4.4) mit pytest und der entsprechenden Python-Version aus (z.B. py37 für Python 3.7)

**format** ruft black auf, so kann lokal der Code mit `tox -e format` korrekt formatiert werden.

**check-format** überprüft mittels black, dass der Code korrekt formatiert ist (wird auf Travis CI ausgeführt, siehe nächster Abschnitt).

**pylint, flake8, mypy** ruft die entsprechenden Tools auf (siehe Abschnitt 4.4.6).

**sphinx** generiert die Softwaredokumentation mittels Sphinx (siehe Kapitel 5).

[65] `https://github.com/python/mypy/issues/5957`
[66] `https://tox.readthedocs.io/`
[67] `https://virtualenv.pypa.io/`

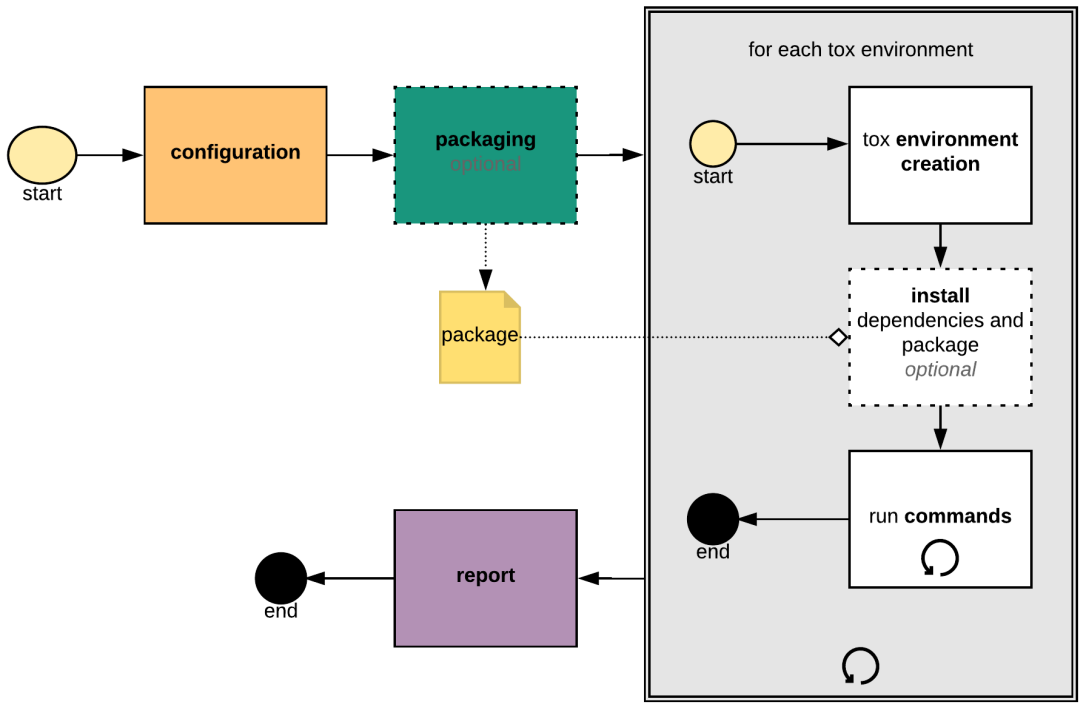



Abbildung 4.9.: Systemübersicht von tox, aus der tox-Dokumentation, Bernát Gábor. *tox documentation: System overview*. URL: `https://tox.readthedocs.io/en/latest/#system-overview` (besucht am 11. 06. 2019)

### CI-Anbieter

Es ist durchaus nützlich, die Tests und diversen Lint-Tools lokal ausführen zu können – jedoch besteht so die Gefahr, dass nicht alle Tools mit jeder Änderung vom Entwickler ausgeführt werden. Deshalb sollten diese Tools auch nach jeder Änderung von einem CI-System ausgeführt werden, welches mit GitHub integriert werden kann.

Es ist zwar möglich, selber ein System wie Jenkins, Buildbot oder GitLab CI aufzusetzen, aber es existieren auch Anbieter, welche fertige Systeme zu diesem Zweck anbieten. Einige davon sind gratis für Opensource-Projekte, weshalb wir nur diese genauer in Betracht gezogen haben. Uns sind dabei folgende Anbieter bekannt, mit den jeweiligen Limitationen für Opensource-Projekte:

- Azure Pipelines[68] (Linux/Windows/macOS, 10 parallele Jobs)
- Travis CI[69] (Linux/Windows/macOS, 5 parallele Jobs)
- Circle CI[70] (Linux/macOS, 1 paralleler Job, max. 250 Minuten pro Woche)
- AppVeyor[71] (Linux/Windows, 1 paralleler Job)

Als Teil dieser Arbeit haben wir die Tests und Tools nur auf Linux ausgeführt. Der Crashbin-Server läuft jedoch auch unter Windows, und wir wollen uns die Option offen halten, die Tests auch als Teil der CI unter Windows auszuführen.

Azure Pipelines bietet zwar ein sehr grosszügiges Gratis-Angebot, ist jedoch vergleichsweise noch sehr neu.[72] Dadurch kämpft es mit einigen Kinderkrankheiten und ist aufwändig zu konfigurieren[73].

Wir haben uns daher für Travis CI entschieden. Dank tox sind die Abhängigkeiten sowie Kommandos CI-unabhängig konfiguriert, ein Wechsel zu einem anderen CI-Anbieter wäre also einfach zu bewerkstelligen.

Die Integration von tox mit Travis ist simpel, wie in Listing 4.2 sichtbar. Daraus resultiert eine Build-Matrix bei Travis, die für jedes tox-Environment einen eigenen Buildjob benutzt, wie in Abbildung 4.10 zu sehen.

[68] `https://azure.microsoft.com/en-us/services/devops/pipelines/`

[69] `https://www.travis-ci.org/`

[70] `https://circleci.com/`

[71] `https://www.appveyor.com/`

[72] Cool Jamie. *Introducing Azure DevOps*. Microsoft Corporation. 10. Sep. 2018. URL: `https://azure.microsoft.com/en-us/blog/introducing-azure-devops/` (besucht am 11. 06. 2019).

[73] Siehe beispielsweise: Schlawack Hynek. *Python in Azure Pipelines, Step by Step*. 3. Juni 2019. URL: `https://hynek.me/articles/simple-python-azure-pipelines/` (besucht am 11. 06. 2019) – der Artikel erschien erst nach der getroffenen Entscheidung, fasst aber die Probleme gut zusammen.

```
dist: xenial
language: python
python: 3.7
os: linux

env:
  matrix:
  - TOXENV=py37
  - TOXENV=check-format
  - TOXENV=pylint
  - TOXENV=flake8
  - TOXENV=mypy
  - TOXENV=sphinx

install:
- pip install tox

script:
- tox
```

Listing 4.2: Konfiguration für Travis CI

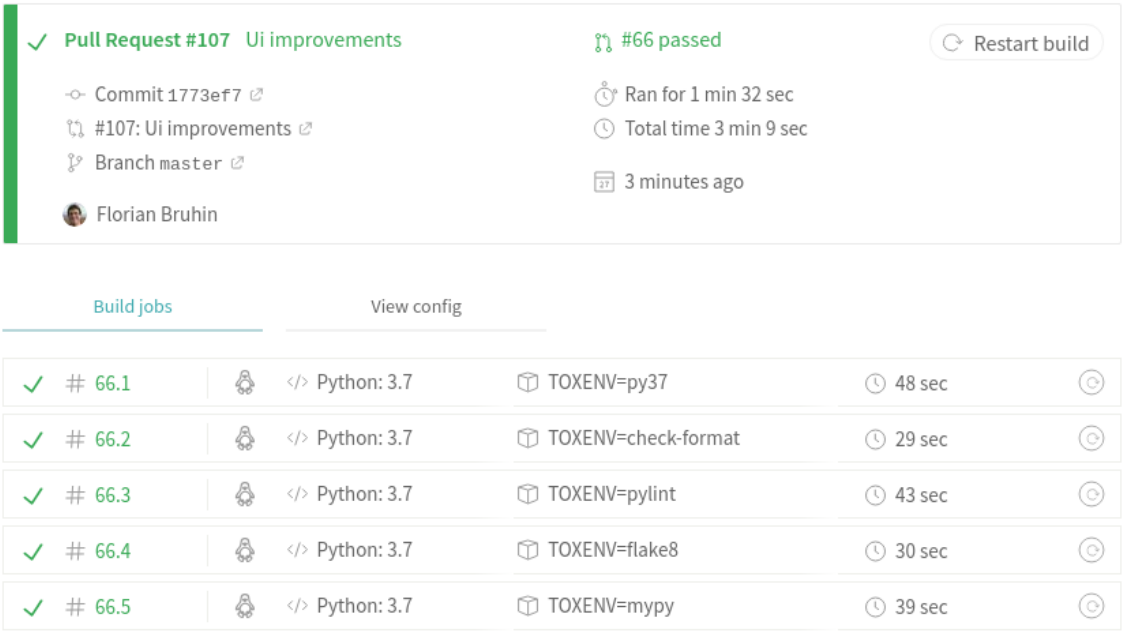



Abbildung 4.10.: Beispiel eines Travis-Builds

## 4.5. Code-Statistiken

In den Tabellen 4.5 und 4.6 werden Code-Statistiken von Crashbin gezeigt, die mit dem Tool *cloc*[74] ("count lines of code") erstellt wurden. Dabei werden nur `.py` und `.html`-Dateien gezählt – Dokumentation, Tool-Konfigurationen, leere `__init__.py`-Dateien sowie alle automatisch generierten Dateien wurden ausgelassen.

Insgesamt besteht Crashbin aus rund 2230 Code-Zeilen plus 600 leere Zeilen bzw. Kommentare.

| Datei | Blank | Kommentar | Code |
|---|---|---|---|
| tests/conftest.py | 23 | 11 | 62 |
| tests/test_api.py | 26 | 0 | 56 |
| tests/test_models.py | 30 | 0 | 96 |
| tests/test_signals.py | 9 | 0 | 22 |
| tests/test_templatetags.py | 8 | 0 | 29 |
| tests/test_utils.py | 12 | 0 | 26 |
| tests/test_views.py | 105 | 2 | 392 |

Tabelle 4.5.: Code-Statistik

[74] `https://github.com/AlDanial/cloc`

| Datei | Blank | Kommentar | Code |
|---|---|---|---|
| crashbin_app/admin.py | 1 | 0 | 8 |
| crashbin_app/api/serializers.py | 8 | 0 | 26 |
| crashbin_app/api/urls.py | 2 | 0 | 11 |
| crashbin_app/api/views.py | 10 | 0 | 22 |
| crashbin_app/apps.py | 5 | 1 | 9 |
| crashbin_app/forms.py | 9 | 0 | 17 |
| crashbin_app/migrations/0004_create_inbox.py | 6 | 0 | 7 |
| crashbin_app/migrations/0008_create_mailbox.py | 6 | 0 | 10 |
| crashbin_app/models.py | 50 | 2 | 136 |
| crashbin_app/signals.py | 8 | 7 | 19 |
| crashbin_app/templates/crashbin_app/base.html | 9 | 0 | 59 |
| . . . /bin_detail.html | 6 | 0 | 146 |
| . . . /bin_list_component.html | 1 | 0 | 58 |
| . . . /bins.html | 2 | 0 | 23 |
| . . . /form.html | 3 | 0 | 48 |
| . . . /home.html | 6 | 0 | 52 |
| . . . /labels.html | 3 | 0 | 45 |
| . . . /modal_delete_component.html | 0 | 0 | 28 |
| . . . /report_detail.html | 5 | 0 | 149 |
| . . . /report_list_component.html | 1 | 0 | 50 |
| . . . /reports.html | 4 | 0 | 18 |
| . . . /search_component.html | 0 | 0 | 9 |
| . . . /set_settings.html | 6 | 0 | 89 |
| crashbin_app/templates/registration/login.html | 2 | 0 | 17 |
| crashbin_app/templatetags/components.py | 11 | 9 | 14 |
| crashbin_app/templatetags/utils.py | 10 | 0 | 24 |
| crashbin_app/urls.py | 2 | 0 | 35 |
| crashbin_app/utils.py | 13 | 3 | 38 |
| crashbin_app/views/bin_views.py | 13 | 0 | 52 |
| crashbin_app/views/label_views.py | 8 | 0 | 32 |
| crashbin_app/views/misc_views.py | 7 | 0 | 29 |
| crashbin_app/views/report_views.py | 13 | 0 | 54 |
| crashbin_app/views/setting_views.py | 15 | 0 | 97 |
| crashbin_settings_example.py | 13 | 8 | 31 |
| crashbin_site/settings.py | 37 | 23 | 83 |
| crashbin_site/urls.py | 2 | 14 | 3 |

Tabelle 4.6.: Code-Statistik (Fortsetzung)

## 4.6. Resultate und Weiterentwicklung

### 4.6.1. Resultate

Trotz des straffen Zeitplans war es uns möglich, ein "Minimum Viable Product" (MVP) zu implementieren, das fast die komplette geplante Funktionalität enthält:

- Bins können über die Weboberfläche angelegt oder editiert werden.
- Reports können von einer Applikation über eine REST-API an Crashbin geschickt werden.
- Sowohl Bins als auch Reports können mittels Labels markiert werden.
- Reports landen in einem konfigurierbaren Inbox-Bin und können von da in andere Bins verschoben werden.
- Es ist möglich, einen Bin zu abonnieren sowie Maintainer als zuständige Personen einzutragen.
- Bins lassen sich als "related bins" untereinander verknüpfen.
- Bins lassen sich archivieren, wenn ein Problem gelöst wurde.
- Aus dem Webinterface können Reports beantwortet werden, die Antwort wird dabei als Mail verschickt.
- Eingehende Antworten auf diese Mails werden dem korrekten Report zugeordnet und in der Weboberfläche angezeigt.
- Zusätzlich lassen sich Notizen verfassen, die in der Weboberfläche angezeigt werden, jedoch nicht zum Endbenutzer geschickt werden.
- Plugins werden geladen und können auf gewisse Ereignisse (neuer Report, Report wird anderem Bin zugeordnet) reagieren.

Bei einigen Funktionen mussten jedoch gewisse Kompromisse eingegangen werden, um im Zeitplan zu bleiben:

- Es besteht keine eingebaute Möglichkeit, um automatisch Mails zu verschicken, wenn ein Report einem Bin zugewiesen wird. Jedoch wäre es möglich, dies in einem Plugin zu implementieren.
- Es ist nicht möglich, Textbausteine für ausgehende Nachrichten zu definieren.
- Events (wie etwa das Verschieben eines Reports in einen anderen Bins) werden an Plugins weitergereicht, jedoch noch nicht in der Datenbank gespeichert (wie im ursprünglichen Domainmodell vorgesehen), und dadurch auch nicht in der Timeline eines Reports angezeigt.
- Es lassen sich zwar Maintainer und Abonnenten für Bins festlegen, aber es werden momentan bei Änderungen an einem Bin (z.B. neue Reports) noch keine Benachrichtigungen verschickt. Es wäre jedoch bereits möglich, ein Plugin zu implementieren, um Crashbin um diese Funktion zu erweitern.

- Die Archivierung eines Bins hat momentan noch keinen Einfluss auf die Sortierung oder Darstellung im User-Interface.
- Die Suche ist noch sehr rudimentär und durchsucht nur die Titel der Bins bzw. Reports (siehe dazu die Architektur-Überlegungen in Abschnitt 4.3.1).
- Die Felder für Reports sind fix festgelegt (Titel/Mail/Debug-Log).
- Die Verlinkung zu externen Bugtrackern wie GitHub wurde nicht implementiert.
- Die Nutzerverwaltung ist sehr simpel aufgebaut: Ohne Nutzeraccount können nur neue Reports an Crashbin geschickt werden; mit einem Nutzeraccount sind sämtliche Aktionen zugänglich.

### 4.6.2. Möglichkeiten der Weiterentwicklung

Wie in Abschnitt 3.1.3 erwähnt, wird Florian Bruhin mit seinem qutebrowser-Projekt der erste grosse Nutzer von Crashbin sein. Deshalb ist es geplant, Crashbin auch nach Abschluss dieser Arbeit weiter zu entwickeln – natürlich primär mit den Bedürfnissen von qutebrowser im Hinterkopf, trotzdem soll Crashbin seine allgemeine Natur weiterhin behalten und auch für andere Projekte nutzbar sein.

Primär sollen die im letzten Abschnitt genannten Kompromisse aufgehoben werden und die fehlende Funktionalität implementiert werden:

- Das Beantworten von Reports soll vereinfacht werden – zum einen durch Textbausteine für häufige Antworten oder Nachfragen; zum anderen durch automatisierte Antworten, wenn ein Report einem Bin zugewiesen wird. So muss bei bekannten Problemen nur der passende Bin ausgewählt werden, und der Endbenutzer wird automatisch entsprechend benachrichtigt (beispielsweise mit der Bitte, auf die neuste Version zu aktualisieren, bei bereits gelösten Problemen).
- Das Event-System soll erweitert werden, so dass Events abgespeichert und angezeigt werden. Ausserdem sollen bei wichtigen Events (etwa neu eintreffende Reports) Benachrichtungs-Emails an die betreffenden Personen verschickt werden. Die Möglichkeiten bzw. Events, welche für Plugins zu Verfügung stehen, sollen ebenfalls erweitert werden.
- Die Zuordnung von Reports zu den passenden Bins soll einfacher werden. Es soll in einer Liste von Reports möglich sein, diesen mit wenigen Klicks in einen anderen Bin zu verschieben, ohne den Report öffnen zu müssen. Ausserdem sollen archivierte Bins (für bereits gelöste Probleme, für die aber eventuell weiterhin neue Reports eintreffen) ausgegraut und am Schluss einer Liste angezeigt werden.
- Die Suche und Filterung von Bins/Reports soll einfacher werden. Dazu sollen eine Volltext-Suche (siehe Abschnitt 4.3.1) sowie Such-Operatoren implementiert werden. Mit Operatoren wie `maintainer:me` oder `label:important` soll es möglich sein, auch mit vielen Objekten den gesuchten Bin oder Report schnell zu finden.
- Die Felder, welche zu einem Report gehören, sollen dynamisch konfigurierbar sowie durchsuchbar sein (siehe auch Abschnitt 4.3.1). So kann für qutebrowser etwa ein Feld

"qt_version" definiert werden, welches bei der Erstellung eines Reports durch qutebrowser mitgeschickt wird, und dann mit einem Operator wie `qt_version:5.12` durchsuchbar ist.

- Integration mit Bugtrackern soll (basierend auf dem Plugin-System) implementiert werden, mit GitHub als Beispielimplementation. Wenn in Crashbin ein Link zu einem GitHub-Issue angegeben wird, soll das Plugin mittels der GitHub-API einen Kommentar verfassen, die auf den jeweiligen Bin in Crashbin verlinkt.

- Die Nutzerverwaltung soll so ausgebaut werden, dass verschiedene Rollen existieren ("Role-based access control"[75]). So könnte es etwa Entwickler geben, denen lesender Zugriff auf die Reports gewährt werden soll, die aber keine Modifikationen vornehmen dürfen. Weiterhin ist es denkbar, mittels einem zufälligen Code (als Teil einer URL) einem Benutzer Zugriff auf seine eigenen Reports zu geben.

- Im Sinne der EU-Datenschutz-Grundverordnung (DSGVO/GDPR) soll es möglich sein, Reports zu editieren oder zu löschen, wenn etwa versehentlich sensible Daten im Debug-Log enthalten sind. Weiterhin soll es möglich sein, persönliche Daten wie Debug-Logs in Reports automatisch zu löschen, sobald der sie beinhaltende Bin archiviert wurde.

Ist diese Grundfunktionalität implementiert, so sind weitergehende Weiterentwicklungen denkbar:

- Automatische Korrelation von ähnlichen Reports anhand der Stacktraces, wenn das Volumen der Reports grösser wird und so eine manuelle Bearbeitung nicht mehr möglich ist. Entsprechende Algorithmen wurden für Firefox[76] sowie Firefox/Eclipse[77] bereits genauer untersucht.

- Entwicklung einer zweiten (mit Crashbin integrierter) Applikation für die Sammlung von Telemetrie-Daten. Dabei melden sich Applikationen periodisch beim Server mit ähnlichen Daten, wie sie bei Crashes mitgeschickt werden (Versionsnummern, geänderte Einstellungen, etc.). Alternativ wäre es auch möglich, diese Daten aus bestehenden Crash-Reports zu entnehmen. Anstatt die Daten einzeln zu speichern, werden diese entsprechend aggregiert und ausgewertet, um daraus Statistiken zu erstellen. So soll etwa sichtbar werden, welcher Anteil der Nutzer welches Betriebssystem benutzt, um die vorhandenen Ressourcen zur Entwicklung entsprechend einteilen zu können.

- Möglichkeit der Steuerung via Mails. Gerade bei Opensource-Projekten sind Mails für die Kommunikation und Organisation eines Projektes sehr beliebt – so ist etwa der Debian-Bugtracker[78] komplett via Mail kontrollierbar[79], die Zusammenarbeit für den Linux-Kernel passiert fast ausschliesslich via Mail[80], und auch grosse Dienste wie GitHub oder GitLab unterstützen Mail-Antworten auf Benachrichtigungen. Ähnlich dazu sollte es möglich sein,

[75] Fernandez-Buglioni, *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*, S. 249.

[76] T. Dhaliwal, F. Khomh und Y. Zou. „Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox". In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. Sep. 2011, S. 333–342. DOI: 10.1109/ICSM.2011.6080800.

[77] Shaohua Wang, Foutse Khomh und Ying Zou. „Improving Bug Localization Using Correlations in Crash Reports". In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR '13. San Francisco, CA, USA: IEEE Press, 2013, S. 247–256. ISBN: 978-1-4673-2936-1. URL: http://dl.acm.org/citation.cfm?id=2487085.2487135.

[78] https://bugs.debian.org/

[79] https://www.debian.org/Bugs/server-control

[80] https://lkml.org/

via Mail auf eine Report-Benachrichtigung direkt zu antworten, oder Crashbin darüber Befehle wie etwa `crashbin: assign-bin qt-crashes` zu geben.

## 4.7. Projektplan

Der Projektplan wurde während der Arbeit als separates Dokument geführt und bei Änderungen angepasst. Erst gegen das Ende wurde er in die Projektdokumentation integriert.

### 4.7.1. Organisation

Die Projektstruktur für diese Arbeit ist Scrum+, ein Mix aus reinem Scrum und Unified Process. Somit wird in diesem Projekt iterativ vorgegangen, wobei folgende Phasen durchlaufen werden: Inception, Elaboration, Construction und Transition. Für genauere Angaben kann das Skript von Daniel Keller aus der Vorlesung "Software Engineering 1" konsultiert werden.

#### Involvierte Personen und Rollen

Grundsätzlich sind alle Teammitglieder gleichgestellt und besitzen die gleiche Entscheidungsgewalt. Sollte es jedoch zu Unstimmigkeiten kommen, nimmt der Projektleiter seine Funktion wahr und bestimmt den weiteren Verlauf des Streitpunktes. Grundlegende Design- und Anforderungsbestimmungen werden im Team definiert. Ansonsten gelten folgende Rollen der involvierten Personen:

- Florian Bruhin:
  Nimmt die Rolle des Projektleiters ein und versucht über das Projekt einen Überblick zu bewahren und bei Bedarf zu lenken. Ist ein langjähriger Python Programmierer, weshalb er für die Code-Qualität verantwortlich ist und "Bad Practices", sowie konzeptionelle Unschönheiten erkennen und für die Überarbeitung markieren soll. Prüft zusätzlich die Test-Coverage, die Test-Qualität und das Linting. Ist ein Code-Writer und unterstützt seinen Teamkameraden bei allfälligen Programmierschwierigkeiten betreffend Python.

- Luca Tavernini:
  Ist in erster Linie Code-Writer und lernt parallel zu dieser Arbeit neu die Programmiersprache Python. Sein Hauptgebiet ist der Frontend-Bereich und das UI-Design. Ist verantwortlich für die Erstellung der Usability-Tests, sowie deren Organisation und Durchführung an Probanden.

- Stefan Richter:
  Professor und Studiengangleiter der Fachrichtung Informatik an der Hochschule für Technik in Rapperswil. Ist zusätzlich Institutspartner des "Institute for Networked Solutions" der HSR. Begleitet und betreut diese Studien- und Bachelorarbeit.

### 4.7.2. Projektmanagement

#### Zeit-Budget

Das Frühlingssemester besteht aus 15 Wochen, wobei vereinbart wurde, dass sowohl für die SA als auch für die BA die Abgabe zwei Wochen nach Semesterende erfolgt. Dies ergibt somit einen Zeitrahmen von insgesamt 17 Wochen.

Basierend auf den gutgeschriebenen ECTS-Punkten (ausgehend von einem Workload von 25–30 Stunden pro ECTS[81]) entsteht dadurch folgende Zeitaufteilung:

| | SA (Tavernini) | BA (Bruhin) | Total |
|---|---|---|---|
| ECTS | 8 | 12 | |
| Stunden | 200–240 | 300–360 | 500–600 |
| Stunden pro Woche | 12–14 | 18–21 | 30–35 |

Tabelle 4.7.: Zeit-Budget basierend auf ECTS-Punkten

Aufgrund eines Themenwechsels in der dritten Semesterwoche wurden von beiden Teammitgliedern für ein vorheriges Thema (Hex-Browser) bereits ca. 45h investiert. Dieser Aufwand gehört zum Zeit-Budget dazu und wird in der Zeitabrechnung separat aufgelistet.

#### Meilensteine

Die angestrebten Meilensteine sind in Tabelle 4.8 aufgeführt. Zielvorgabe für das Erreichen der Meilensteine ist die Sitzung am Dienstagnachmittag der jeweiligen Woche.

Bis auf zwei Meilensteine konnten alle Terminvorgaben eingehalten werden. Die Gründe für die Verschiebung dieser Meilensteine sehen wie folgt aus:

- End of Elaboration: Aufgrund des Themenwechsels blieb uns weniger Zeit bis zur Zwischenpräsentation. Aus diesem Grund wurde von Beginn an eine verkürzte Elaborationsphase geplant, damit auch genug Zeit für die Erstellung eines Prototypen bleibt. Gegen Ende der Elaborationphase mussten wir jedoch merken, dass wir für eine saubere Konzeption mehr Zeit benötigten. Dies erschien uns für die Zwischenpräsentation wichtiger als ein umfänglich funktionierender Prototyp. Aus diesem Grund wurde die Elaborationsphase um eine Woche nach hinten verschoben.

- Feature Freeze: Dieser Meilenstein wurde eine Woche nach hinten verschoben, damit die Kernfunktionen von Crashbin noch fertig implementiert werden konnten. Hierfür wurde der Buffer aus KW 15 nach vorne verschoben und aufgebraucht.

[81] Hochschulrat. *Richtlinien des Hochschulrates für die koordinierte Erneuerung der Lehre an den universitären Hochschulen der Schweiz im Rahmen des Bologna-Prozesses*. 2015. URL: `https://www.admin.ch/opc/de/classified-compilation/20150869/index.html` (besucht am 25. 02. 2019).

| KW | SW | Meilenstein | Beschreibung |
|---|---|---|---|
| 10 | 3 | M0: Kickoff | Start des Projektes |
| 11 | 4 | M1: Besprechung Projektplan | Erster Entwurf vom Projektplan erstellt und mit dem Betreuer besprochen. |
| 13 | 6 | M2: End of Elaboration | Use Cases und nicht funktionale Anforderungen erfasst. Erster Entwurf für folgende Punkte: Konzept, Wireframes, Architektur und API für Plugins. Doku: Kapitel *Vision*, *Anforderungsspezifikation*; *Analyse* und *Design* erster Entwurf. |
| 15 | 8 | M3: Zwischenpräsentation | Nach etwa der Hälfte der Zeit findet eine Zwischenpräsentation statt, an der die Studierenden den Fortschritt der Arbeit dem Betreuer und dem externen Experten sowie eventuell dem Gegenleser präsentieren. Doku: Kapitel *Einleitung* und *Stand der Technik*. |
| 19 | 12 | M4: Feature Freeze | Alle erwünschten Features sollten abgeschlossen sein. Der Fokus liegt auf Bugfixes, Code-Qualität, etc. Doku: Kapitel *Design* und *Implementation und Tests*. |
| 22 | 15 | M5: Code Freeze | Programmierung ist abgeschlossen. Markiert das Ende der Construction Phase. |
| 24 | 17 | M6: Projektende | Abgabe gemäss Regelung HSR/Betreuer. |

Tabelle 4.8.: Übersicht Meilensteine

### Phasen / Iterationen

Um eine möglichst hohe Flexibilität zu erreichen, werden kurze Iterationen von einer Woche angewendet. Die geplanten Phasen und Iterationen sind in Abbildung 4.11 ersichtlich.

### Abgabe

Die von der HSR definierten Abgabetermine sind in Tabelle 4.9 aufgelistet.

| Datum | KW | SW | Abgabe |
|---|---|---|---|
| 07.06.2019 | 23 | 16 | Die Studierenden geben den Abstract für die Diplomarbeitsbroschüre zur Kontrolle an ihren Betreuer/Examinator frei. Die Studierenden erhalten vorgängig vom Studiengangsekretariat die Aufforderung mit den Zugangsdaten zur Online-Erfassung des Abstracts im DAB-Tool. Die Studierenden senden per Email das A0-Poster zur Prüfung an ihren Examinator/Betreuer. Vorlagen sowie eine ausführliche Anleitung betreffend Dokumentation stehen auf dem Skripteserver zur Verfügung. |
| 12.06.2019 | 24 | 17 | Der Betreuer/Examinator gibt das Dokument mit dem korrekten und vollständigen Abstract der Broschüre zur Weiterverarbeitung an das Studiengangsekretariat frei. Für die Ausstellung der Bachelorarbeiten das A0 Posters per Email bis 10.00 Uhr an das Studiengangsekretariat senden. |
| 14.06.2019 | 24 | 17 | Hochladen aller verlangten Dokumente auf archiv-i.hsr.ch sowie Abgabe des Berichts an den Betreuer bis 12.00 Uhr |

Tabelle 4.9.: Abgabetermine

### Zeiterfassung

Für die zeitliche Erfassung der getätigten Arbeiten wird die Webapplikation "Clockify"[82] verwendet, da Florian Bruhin bei seiner SA bereits gute Erfahrungen damit machte. Die Zeit wird dabei nicht pro Arbeitspaket abgerechnet, sondern in grobe Kategorien aufgeteilt. Diese Aufteilung erfolgt folgendermassen:

- Vorbereitung (Repository aufsetzen, Python-Tutorials, etc.)
- Recherche (Auswahl von Frameworks, Literaturstudium, etc.)
- Dokumentation (Projektplan, Zwischenpräsentation, Projektdokumentation, Softwaredokumentation, etc.)
- Sitzung Team bzw. Betreuer
- Implementation (inklusive Tests und Bugfixes)

[82] `https://www.clockify.me/`

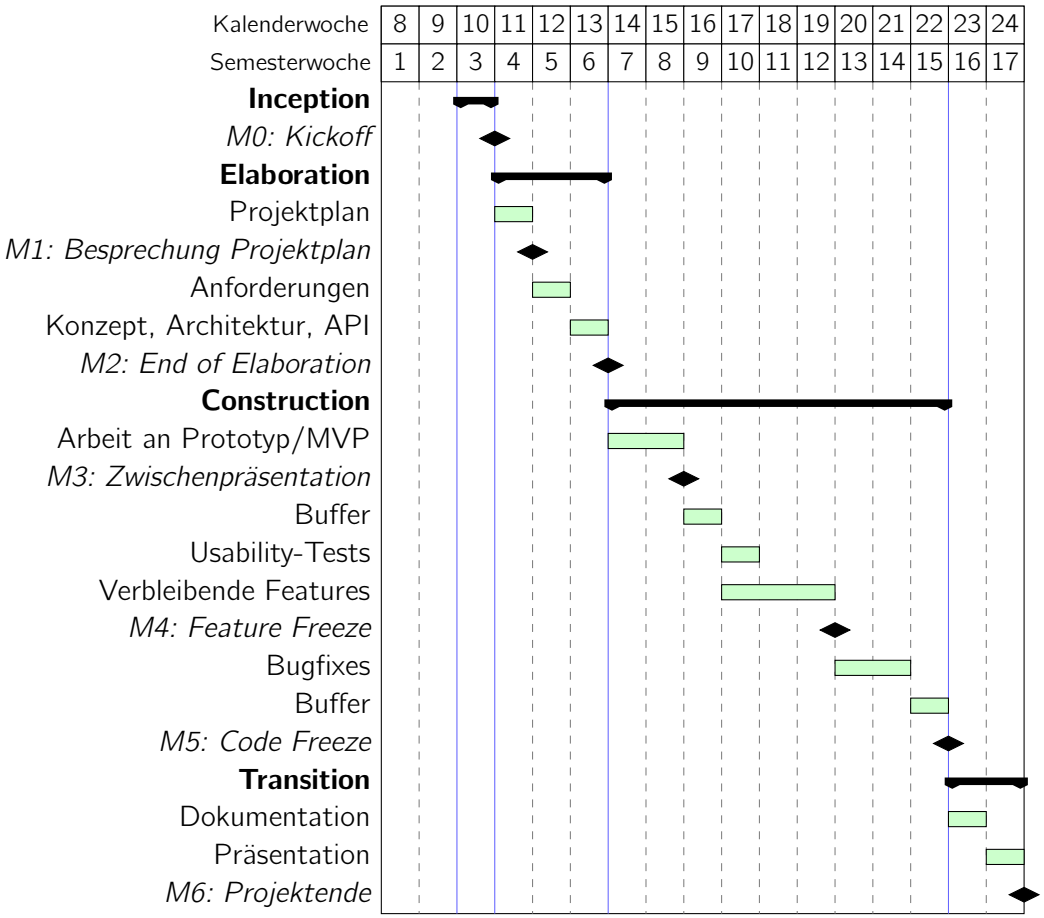



Abbildung 4.11.: Projektplan

Um den Ressourcenverbrauch des Projektes genug präzise zu dokumentieren, jedoch nicht den Verwaltungsaufwand unnötig in die Höhe zu treiben, wird die effektiv benötigte Zeit auf 15 Minuten genau erfasst.

In Abbildung 4.12 ist die effektiv aufgewendete Zeit pro Thema und Woche sichtbar. Die im Projektplan definierten Phasen sind anhand der Themen klar ersichtlich.

Abbildungen 4.13 und 4.14 sowie die Tabelle 4.10 zeigen die insgesamt investierte Zeit. Die aufgrund der ECTS-Punkte vorgegebene durchschnittliche Soll-Zeit ist auf der Y-Achse grün eingezeichnet. Luca Tavernini hat leicht mehr Zeit investiert als eigentlich für eine SA vorgesehen – dies ist dadurch zu erklären, dass Luca trotz des Themenwechsels in Woche 3 schon viel Zeit in die Python-Tutorials investieren konnte, beim neuen Thema aber trotzdem noch viel zu erledigen war.

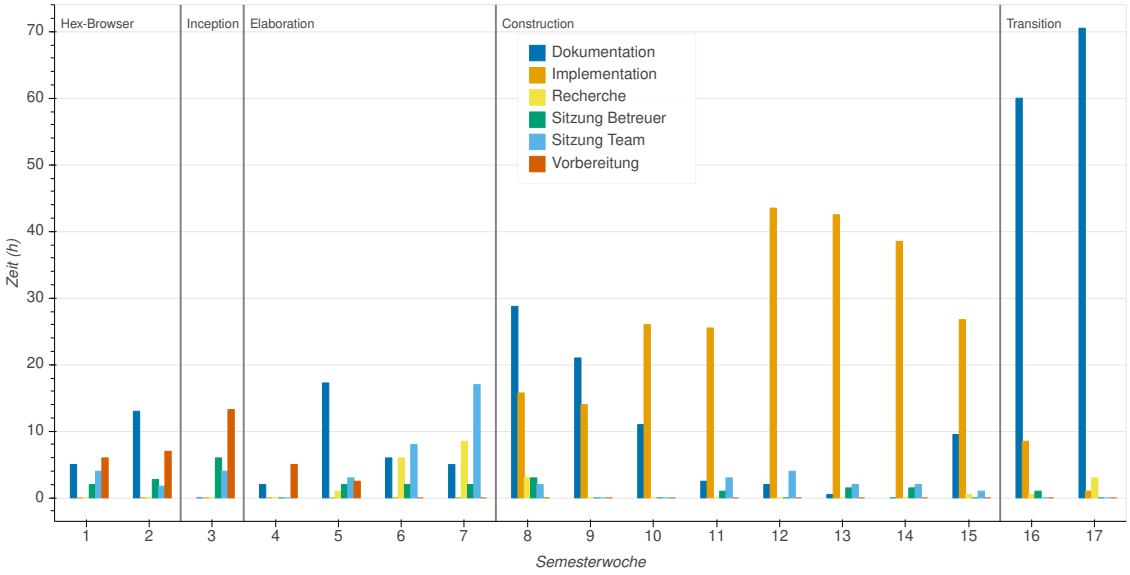



Abbildung 4.12.: Aufgewendete Stunden pro Thema

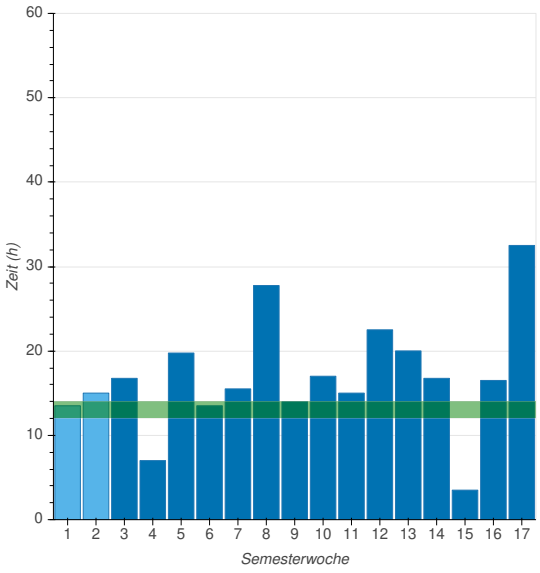



Abbildung 4.13.: Zeiterfassung L. Tavernini

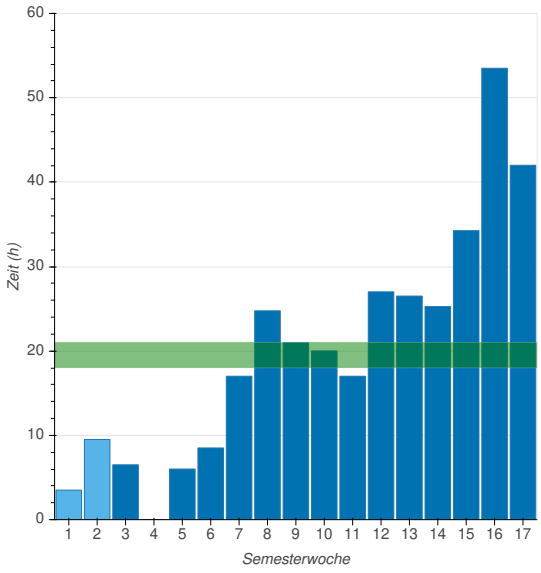



Abbildung 4.14.: Zeiterfassung F. Bruhin

| Woche | Florian | Luca | Total |
|---|---|---|---|
| 1 | 4 | 14 | 17 |
| 2 | 10 | 15 | 24 |
| 3 | 6 | 17 | 23 |
| 4 | 0 | 7 | 7 |
| 5 | 6 | 20 | 26 |
| 6 | 8 | 14 | 22 |
| 7 | 17 | 16 | 32 |
| 8 | 25 | 28 | 52 |
| 9 | 21 | 14 | 35 |
| 10 | 20 | 17 | 37 |
| 11 | 17 | 15 | 32 |
| 12 | 27 | 22 | 50 |
| 13 | 26 | 20 | 46 |
| 14 | 25 | 17 | 42 |
| 15 | 34 | 4 | 38 |
| 16 | 54 | 16 | 70 |
| 17 | 42 | 32 | 74 |
| Total | 342 | 286 | 629 |

Tabelle 4.10.: Aufgewendete Stunden (Werte auf volle Stunden gerundet)

#### Besprechungen

Alle Projektmitglieder treffen sich mindestens einmal pro Woche. Diese Meetings finden jeweils nach Absprache an einem Montag Nachmittag nach 15:00 Uhr oder an einem Dienstag statt. Sie dienen zum Austausch, um administrative Fragen zu klären und die nächsten Schritte zu planen. Wöchentlich am Dienstag um 14:00 Uhr bestand die Möglichkeit, eine Sitzung mit dem Betreuer durchzuführen. Diese fand nur auf Wunsch der Projektmitglieder statt.

### 4.7.3. Risikomanagement

#### Risiken

Zu Beginn des Projekts wurde folgende Risiken festgehalten:

| Nr | Bereich | Beschreibung | Schaden total [h] | Eintritts-WSK | Gewichteter Schaden [h] |
|---|---|---|---|---|---|
| R1 | Python | Programmierschwierigkeiten wegen fehlendem Python-Wissen bei Luca Tavernini. | 30 | 60% | 18 |
| R2 | Architektur | Fehlerhafte Architekturentscheide zu Beginn des Projektes. | 40 | 20% | 8 |
| R3 | Framework | Schwierigkeiten beim Gebrauch des Frameworks. Ungeeignetes Framework wurde ausgewählt. | 40 | 20% | 8 |
| R4 | Schnittstellen | Komplikation bei der Anbindung von weiteren Diensten wie z.B. Github. | 30 | 30% | 9 |
| R5 | Usability | Schwierigkeiten bei der Bedienung der Applikation. UI ist nicht selbsterklärend. | 30 | 30% | 9 |
| R6 | Integration | Probleme beim Gebrauch der ausgewählten Tools wegen unterschiedlichen Versionen oder Einstellungen, eventuell auch wegen erstmaligen Gebrauch. | 15 | 40% | 6 |
| | | | | Total: | 58 |

Tabelle 4.11.: Relevanteste Risiken für das Projekt

Basierend auf dem totalen gewichteten Schaden von 58 Stunden wurden zwei Wochen Pufferzeit im Zeitplan (Seite 69) berücksichtigt: Eine Woche vor der Zwischenpräsentation für initiale Probleme mit dem Prototyp, sowie eine Woche vor dem Code Freeze.

**Umgang mit Risiken**

| Nr | Vorbeugung | Verhalten beim Eintreten |
|---|---|---|
| R1 | Elaboration Phase nutzen, um Tutorials durchzuarbeiten. Einführung in die Frameworks durch Florian Bruhin bevor Construction Phase beginnt. Regelmässige Code Reviews für stetige Qualitätssteigerung des Codes. Vor Featureimplementierung kurz das Vorgehen im Team besprechen. | Pair Programming nach Bedarf durchführen. Frequenz der Code Reviews erhöhen. Zusätzliche Schulungen durch Florian Bruhin durchführen. |
| R2 | Kein Overengineering der Architektur in der Elaboration Phase anstreben. Möglichst leichte Architekturstrukturen definieren, mit dem Fokus auf die Erweiterbarkeit. Fortlaufendes Refactoring anstreben. | Architektur anpassen. |
| R3 | Einen Vergleich zwischen den verschiedenen Frameworks anstellen und gemäss den Requirements beurteilen und auswählen. Betroffenes Framework frühzeitig auswählen, da es ev. Einfluss auf die Architektur haben kann. Frühzeitiges Einarbeiten in das Framework. | Sich mehr in das Framework einarbeiten. Worst Case Szenario: Framework wechseln. |
| R4 | Schnittstellen so früh wie möglich in einem Prototyp testen. Dokumentation frühzeitig konsultieren. | Dokumentation der API konsultieren. Externe Hilfe suchen. |
| R5 | Wireframe möglichst in Elaboration Phase erstellen. Frühzeitiges Usability-Testing, möglichst schon in der Mitte der Construction Phase. | Zusätzliche Usability-Tests durchführen. UI anpassen gemäss Feedback. |
| R6 | Bis End of Elaboraton sicherstellen, dass alle Projektmitglieder die gleichen Tools in der gleichen Versionierung und mit den gleichen Einstellungen installiert haben. | Das betroffene Tool mit einer Alternative auswechseln. Je nach Wichtigkeit komplett auf das Tool verzichten. |

Tabelle 4.12.: Vorbeugung und Massnahmen für die erkannten Risiken

Weiterhin werden folgende Massnahmen zur Risikovorbeugung und Schadensverminderung getroffen:

- Obligatorische Codereviews, Tests, und Achten auf gute Codequalität.
- Pflegen einer offenen Kommunikation und ehrlichen Fehlerkultur, um allfällige Risiken möglichst früh zu erkennen.
- “Keep it simple”-Ansatz, mit Modularität und simpler Architektur im Fokus.

#### Auswertung

Während des ganzen Projektes sind keine der oben genannten Risiken eingetreten. Im Verlauf der Arbeit konnten einige davon sogar ganz gestrichen werden, so z.B. R1 und R4. Nichtsdestotrotz haben sich einige Ereignisse ergeben, welche die durch die Risikoanalyse geplante Buffer-Zeit aufbrauchten:

- Krankheit eines Teammitgliedes
- Arbeitsaufwand anderer Module während des Semesters
- Unterschätzung der verkürzten Projektzeit, welche durch den Themenwechsel entstanden ist
- Knappes Zeitfenster um die Dokumentation abzuschliessen
- Implementationsaufwand für ein minimum viable product von Crashbin

### 4.7.4. Qualitätsmanagement

#### Dokumentation

Die Dokumentation des "Crashbin"-Projektes wird in einem eigens dafür eingerichteten Git-Repository abgelegt, das sich auf GitHub befindet[83]. Die Qualität der Dokumente wird in erster Linie dadurch sichergestellt, dass sämtliche Änderungen nachverfolgt und dann im jeweils nächsten Teammeeting im Plenum besprochen und nachvollzogen werden. Auf diese Weise repräsentiert die Dokumentation immer den Konsens des gesamten Teams und nicht die Meinung von Einzelpersonen.

#### Projektmanagement

Aufgrund der kleinen Teamgrösse wird das ganze Projektmanagement über GitHub organisiert. Hierfür werden die Arbeitspakete als Issues erfasst. Für die Sprint-Planung werden diese Issues für jede Projektphase in einem separaten Kanban-Board auf GitHub geplant.

#### Entwicklung

Der Quellcode wird mittels dem Git-Versionskontrollsystem verwaltet und befindet sich auf Github[84]. Zur Ausführung der Tests und Codequalität-Tools bei jedem Commit wird Travis CI eingesetzt.

[83] `https://github.com/The-Compiler/crashbin-docs`
[84] `https://github.com/The-Compiler/crashbin`

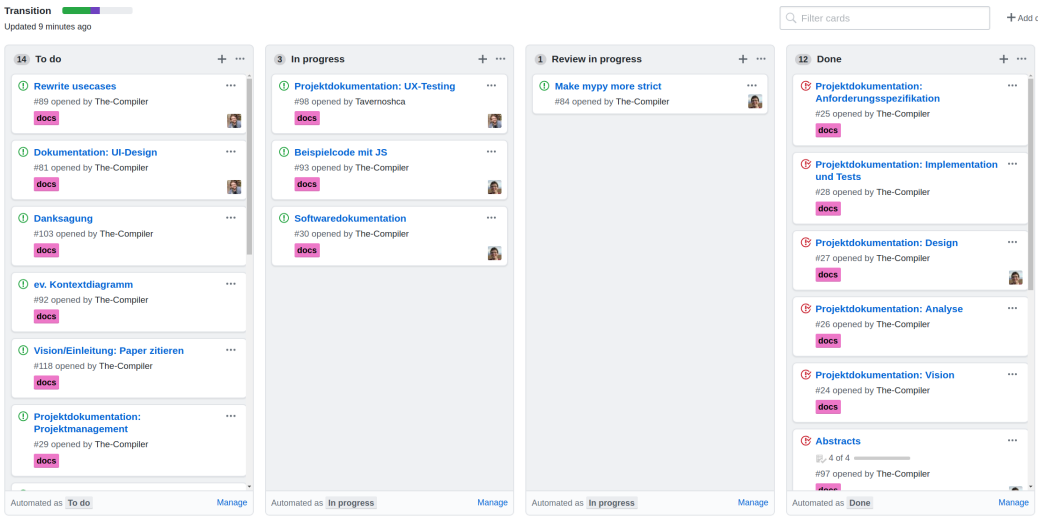



Abbildung 4.15.: Kanban-Board auf GitHub

### Code Reviews

GitHub Pull Requests werden benutzt, um Code Reviews durchzuführen. Der master-Branch wird als "protected branch" eingerichtet, sodass jeglicher Code nur nach dem Vier-Augen-Prinzip in das Repository eingepflegt wird.

Code Reviews erfolgen regelmässig nach Fertigstellung eines Features von ausreichender Komplexität. Hierbei wird das Review eigenständig vom jeweiligen Teamkollegen durchgeführt. Bei Bedarf kann das Review auch zu zweit mit sofortiger Rückmeldung durchgeführt werden. Sollten im Verlauf des Development an einer Stelle Schwierigkeiten auftreten, so wird bevorzugt auf die Technik des Pair-Programming zurückgegriffen.

## 4.7.5. Tools und Infrastruktur

Die benötigte Infrastruktur ist bereits vorhanden – wir arbeiten primär auf unseren eigenen Laptops (Luca Tavernini unter Windows, Florian Bruhin unter Linux).

Die Wahl der IDE und weiteren Tools ist bewusst nicht vorgegeben – Luca Tavernini benutzt primär PyCharm, während Florian Bruhin emacs bevorzugt. Es werden deshalb aber keine Probleme bzw. Inkompatibilitäten erwartet.

Die diversen zusätzlichen Tools (Style Checker, mypy, static analyser, etc.) sind bereits teilweise in den entsprechenden Umgebungen eingebunden, laufen aber auch unabhängig davon auf Travis CI.

Die Repositories für das Projekt befinden sich auf GitHub:

- `https://github.com/The-Compiler/crashbin`
- `https://github.com/The-Compiler/crashbin-docs`

Ausser Clockify und GitHub wird keine weitere Projektmanagement-Software (wie Redmine oder JIRA) verwendet. GitHub bietet inzwischen selbst diverse Funktionen zum Projekt-Management, welche wir für diese Grösse von Projekt sowie nur zwei Personen als ausreichend erachten.

Dokumentationen werden in LaTeX verfasst, kürzere Notizen oder Sitzungsprotokolle im Markdown-Format als GitHub-Wikiseiten.

Als Umgebung für den Webservice wird ausschliesslich Linux mit Python 3.6 oder 3.7 unterstützt. Dadurch ist Support für "ältere" Linux-Distributionen wie Ubuntu 18.04 LTS (Support seitens Ubuntu bis April 2023) ebenfalls gegeben.

# 5. Softwaredokumentation

Die auf den folgenden Seiten enthaltene Software-Dokumentation wurde mittels Sphinx[1] generiert. Bei Sphinx handelt es sich um ein Dokumentations-Tool, welches in Python-Projekten sehr weit verbreitet ist – so ist auch die Dokumentation für Python selbst unter `https://docs.python.org/` mittels Sphinx verfasst.

Typischerweise wird die HTML-Version der Dokumentation in einem Browser angesehen, aber Sphinx bietet auch die Möglichkeit, aus der Dokumentation ein PDF zu generieren. Diese haben wir hier genutzt. Die Dokumentation ist in Englisch verfasst, da Crashbin auch für andere Projekte nutzbar sein soll, und so am meisten Leute erreicht werden.

[1] `https://www.sphinx-doc.org/`

CHAPTER

ONE

# GETTING STARTED

## 1.1 Installing Crashbin

Crashbin is a normal Python package, and as such, can be installed using the `pip` package manager. See the Python Packaging User Guide for details on setting up pip.

Currently, no stable release has been pushed to PyPI (the Python Package Index) yet. However, pip can be used to install directly from GitHub:

```
pip install git+https://github.com/The-Compiler/crashbin.git
```

## 1.2 Starting Crashbin

To set up an initial database, add a new user, and run Django's integrated development server, use:

```
python manage.py migrate
python manage.py createsuperuser
python manage.py runserver
```

Then, navigate to http://localhost:8000/ to access Crashbin.

On a production system, it's recommended to set up a proper webserver to run Crashbin with WSGI, see the Django deployment documentation for examples. Popular choices for servers include Apache (with `mod_wsgi`), Gunicorn or uWSGI.

## 1.3 Basic configuration

See *Configuring Crashbin* for detailed instructions on all available settings. When setting up Crashbin on a production system, it's recommended to change the following settings:

- If Crashbin should use an existing SQL server (like MariaDB or PostgreSQL), adjust the `DATABASE` setting accordingly.
- Adjust the `EMAIL` settings to be able to send and receive mails in Crashbin. Also, adjust the subjects used to match your project.
- Replace `SECRET_KEY` by a new, unique, secret key. You can use:

```
python -c 'from django.core.management import utils; print(utils.get_random_
↪secret_key())'
```

to generate such a key.

- Turn off `DEBUG`.
- Add the host used to serve crashbin to `ALLOWED_HOSTS`.

## 1.4 Basic Concepts

It's useful to understand the following basic concepts how Crashbin organizes data:

- An application crashes and sends a new report to Crashbin. See *Integrating Crashbin* for details on how this happens.
- Crashbin collects the report and puts it into the configured "Inbox bin".
- Bins are "containers" used to organize reports which are similar in some way. As an example, you could create a bin to collect all reports caused by a particular bug, or to group all reports related to a certain subsystem.
- Based on the information in the report, it should be moved from the inbox bin into the correct bin, or a new bin should be created for it.
- Both bins and reports can be tagged by using labels, for example to mark reports from an important customer.
- When viewing a report, it's possible to reply to the user who sent it using the Crashbin webinterface. Similarly, internal notes can be taken, which aren't sent to the user.

CHAPTER

TWO

# CONFIGURING CRASHBIN

Crashbin includes a `crashbin_settings_example.py` file in its repository, which contains all available settings with their default values. To configure crashbin, copy that file to `crashbin_settings.py` and adjust the settings to your liking.

The following settings are available:

- `HOMEDIR` (string or a `pathlib.Path` object): The directory used by Crashbin to store its data. Default: `~/.crashbin`.
- `DATABASE`: (dict) The Django database settings. Refer to DATABASES in the Django documentation for details. Default: A sqlite database in `HOMEDIR/db.sqlite3`.
- `LANGUAGE_CODE`: (str) The language code to use. Refer to LANGUAGE_CODE in the Django documentation for details. Note that Crashbin currently is only available in English. Default: `en-us`.
- `TIME_ZONE`: (str) The timezone to use. Refer to TIME_ZONE in the Django documentation for details. Default: `Europe/Zurich`.
- `MEDIA_ROOT`: (str or a `pathlib.Path` object) The path used to store media files. Refer to MEDIA_ROOT in the Django documentation for details. Default: `HOMEDIR/media`.
- `EMAIL`: (dict) A dictionary of email settings, with the following keys:
  - `backend`: (str) The email backend to use for outgoing mails (`smtp`, `console`, `filebased`, `locmem`, `dummy`), see the section about email backends in the Django documentation for details. Default: `console`.
  - `smtp_host` (str), `smtp_port` (int): The server/port to connect to when using SMTP.
  - `smtp_user` (str), `smtp_password` (str): The SMTP credentials to use.
  - `smtp_tls` (bool): Whether to use TLS encryption for the SMTP connection.
  - `incoming_url`: (str) The URL to use to fetch incoming mails, see the django-mailbox documentation for details. If set to `None` (the default), mails can be supplied manually via `manage.py processincomingmessage`.
  - `outgoing_subject`: (str) The subject to use for outgoing messages. A `{}` placeholder is replaced by the report ID. Default: `qutebrowser report #{}`
  - `incoming_subject`: (str) A regex for incoming message subjects. The capture group needs to capture the report ID. If your `outgoing_subject` is `example report #{}`, a sensible value would be `.*example report #(.*)`. The initial `.*` accounts for added text such as `Re:` in incoming mails. Default: `.*qutebrowser report #(.*)`
  - `outgoing_address`: (str) The mail address to use for outgoing mails. Default: `crashes@qutebrowser.org`

- `SECRET_KEY`: (str) A randomly generated secret key to use for cryptographic signing. Refer to SECRET_KEY in the Django documentation for details. **Should be kept secret!**
- `DEBUG`: (bool) Whether to enable debug features. Refer to DEBUG in the Django documentation for details. **Don't run with debug turned on in production!** Default: true
- `ALLOWED_HOSTS`: (list of str) The hostnames allowed to serve this Django site. Refer to ALLOWED_HOSTS in the Django documentation for details.
- `CUSTOM_CSS`: (str) Custom CSS to add to every Crashbin page. Can be used to restyle its interface.
- `INBOX_BIN`: (str) The name of the Bin to use for new reports. Default: `Inbox`.
- `LABEL_COLORS`: (list of str) Colors to suggest in the color picker when adding a new label.

CHAPTER

THREE

# INTEGRATING CRASHBIN

Crashbin exposes a HTTP REST API to collect reports from applications. Since the API is very simple to use, there is no client-side `crashbin` library needed to report crashes to Crashbin.

Each report contains a mandatory title (such as a short text describing the exception which occurred), as well as an optional debug log and the reporter's email address.

Reports should be sent as a POST request to https://crashbin.example.com/api/report/new/ (including the trailing slash!), with the following fields:

- `title` (The title to use, mandatory)
- `log` (Debug log, optional)
- `email` (The reporter's email address, optional)

As an example, here's how unhandled exception reports could be sent to Crashbin from Python, using the requests HTTP library:

```
1  import sys
2  import requests
3  import traceback
4
5  CRASHBIN_URL = 'http://crashbin.example.org/api/report/new/'
6
7  def handle_exception(exc_type, exc_value, exc_traceback):
8      title = traceback.format_exception_only(exc_type, exc_value)[0]
9      text = ''.join(traceback.format_exception(exc_type, exc_value, exc_traceback))
10     requests.post(CRASHBIN_URL, {'title': title, 'log': text})
11     sys.__excepthook__(exc_type, exc_value, exc_traceback)
12
13 sys.excepthook = handle_exception
14
15 def main():
16     raise Exception("Unhandled exception")
17
18 main()
```

On line 7, an exception handler is defined, which uses the Python `traceback` module to get a suitable title and stacktrace text. It then submits the crash log to Crashbin and calls the original Python excepthook. Similarly, the exception hook could prompt the user for their email address and further information.

> **Warning:** Code in exception hooks must be carefully written in order to not trigger additional exceptions. Otherwise, the exception hook will not finish, and thus no report will be received by Crashbin.

Submitting reports is possible in any language which supports handling crashes, such as JavaScript with Node.JS:

```
const request = require('request');
const CRASHBIN_URL = 'https://crashbin.example.org/api/report/new/';

process.on('uncaughtException', (err, origin) => {
    console.error(err);
    request.post(CRASHBIN_URL, {form: {title: err, log: origin}}, function() {
        process.exit(1);
    });
});

function main() {
    throw "Unhandled exception";
}

main();
```

CHAPTER

FOUR

# EXTENDING CRASHBIN

To automate repetitive tasks, Crashbin can be extended with plugins. Plugins are simple Python files in `~/.crashbin/plugins` (or in `HOMEDIR/plugins` if the homedir location was changed).

Plugins can use the Django signals mechanism to get notified on certain events. Currently, only two such events are available:

`crashbin_app.signals.`**`new_report = <django.dispatch.dispatcher.Signal object>`**
: Emitted when a new report is received by Crashbin. The Report object is passed as the `report` argument.

`crashbin_app.signals.`**`bin_assigned = <django.dispatch.dispatcher.Signal object>`**
: Emitted when a report is assigned to a new bin. Receivers are called with `report`, `old_bin`, `new_bin` and `user` arguments.

A minimal example plugin which reacts on those events could look like this:

```
from crashbin_app import signals

@signals.receiver(signals.new_report)
def on_new_report(report, **kwargs):
    print(f"new report: {report.title}")

@signals.receiver(signals.bin_assigned)
def on_bin_assigned(report, old_bin, new_bin, **kwargs):
    print(f"bin assigned: {old_bin} -> {new_bin}")
```

Crashbin's model classes can be used by Plugins as Django models:

**class** `crashbin_app.models.`**Report**(*id*, *email*, *created_at*, *bin*, *log*, *title*)

> **assign_to_bin**(*new_bin: crashbin_app.models.Bin*, *, *user: django.contrib.auth.models.User = None*)
> Assign this report to a bin.

**class** `crashbin_app.models.`**Bin**(*id*, *name*, *description*, *created_at*, *is_archived*)

> **static get_inbox**()
> Get the inbox bin to be used for new reports.

**class** `crashbin_app.models.`**Label**(*id*, *name*, *color*, *description*, *created_at*)

For example, the following plugin moves any new report with a title such as "Exception: Forced crash" into the bin named "Forced crashes":

```
from crashbin_app import signals, models

@signals.receiver(signals.new_report)
def on_new_report(report, **kwargs):
    if report.title.endswith('Exception: Forced crash'):
        new_bin = models.Bin.objects.get(name='Forced crashes')
        report.assign_to_bin(new_bin)
```

# 6. Danksagungen

Besten Dank an Stefan Richter, der diese Arbeit betreut hat. Wir schätzen seine guten Ratschläge, seinen Pragmatismus und die uns entgegengebrachte Flexibilität sehr.

Wir danken Marco Zollinger, AnneMarie O'Neill und Méline Sieber für die Korrekturlesung von Abstract und Management Summary.

Florian Bruhin dankt Méline Sieber für die unbezahlbare Unterstützung, Ermutigung, Aufmunterung und Beruhigung. 我爱你!

Luca Tavernini bedankt sich bei seiner Schwester Lisa Tavernini, ohne deren Socken diese Arbeit nicht geworden wäre, was sie nun ist. Zusätzlich gebührt sein Dank Jacqueline Koller, für dass sie ihm stets mit viel Geduld, Interesse und unermüdlicher Hilfsbereitschaft während der ganzen Arbeit zur Seite stand. "And if you ever get lost, let me be your guiding light..."

Vielen Dank an kezabelle, doismellburning (Kristian Glass) und moldy (René Fleschenberg), die unsere Fragen im Django IRC-Channel beantworteten.

Der Abgabetermin dieser Bachelorarbeit (der 14. Juni 2019) markiert gleichzeitig den zweiten nationalen Frauenstreik. Danke an alle Frauen, die – ob sichtbar oder unsichtbar; direkt oder indirekt – etwas zum Gelingen dieser Arbeit beigetragen haben.

# 7. Fazit

Der Anfang dieser Arbeit war zugegebenermassen holprig – wir starteten ursprünglich mit einem anderen Thema und wechselten erst in der dritten Woche zu der Idee von Crashbin. Trotz des straffen Zeitplans sind wir mit dem Endergebnis zufrieden, auch wenn wir an gewissen Orten Kompromisse eingehen mussten, um die fehlende Zeit aufzuholen.

Wir freuen uns darüber, dass wir an einem Projekt arbeiten konnten, das einen grossen Praxisbezug mit sich bringt. Wir konnten so zwei Fliegen mit einer Klappe erschlagen: Die Arbeit dient nicht nur dem Studium, sondern es entsprang daraus ein Tool, das wir zukünftig selbst nutzen können und wollen. Diese Arbeit ist nun zwar abgeschlossen – wir freuen uns trotzdem darauf, weiter an Crashbin zu arbeiten, um das neue Tool bald im Produktiveinsatz verwenden zu können.

# A. Quellen


Bettenburg, Nicolas u. a. „Quality of bug reports in Eclipse“. In: *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*. ACM. 2007, S. 21–25.

Bettenburg, Nicolas u. a. „What makes a good bug report?“ In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM. 2008, S. 308–318.

Breu, Silvia u. a. „Information Needs in Bug Reports: Improving Cooperation Between Developers and Users“. In: *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*. CSCW '10. Savannah, Georgia, USA: ACM, 2010, S. 301–310. ISBN: 978-1-60558-795-0. DOI: 10.1145/1718918.1718973. URL: http://doi.acm.org/10.1145/1718918.1718973.

Cohn, M. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Signature Series (Cohn). Pearson Education, 2009. ISBN: 9780321660565.

Davies, Steven und Marc Roper. „What's in a Bug Report?“ In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '14. Torino, Italy: ACM, 2014, 26:1–26:10. ISBN: 978-1-4503-2774-9. DOI: 10.1145/2652524.2652541. URL: http://doi.acm.org/10.1145/2652524.2652541.

Dhaliwal, T., F. Khomh und Y. Zou. „Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox“. In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. Sep. 2011, S. 333–342. DOI: 10.1109/ICSM.2011.6080800.

*Django overview*. URL: https://www.djangoproject.com/start/overview/ (besucht am 04. 06. 2019).

Fernandez-Buglioni, E. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. Wiley Software Patterns Series. Wiley, 2013. ISBN: 9781119970484.

*Foreword - Flask 1.0.2 Documentation*. URL: http://flask.pocoo.org/docs/1.0/foreword/ (besucht am 04. 06. 2019).

Gábor, Bernát. *tox documentation: System overview*. URL: https://tox.readthedocs.io/en/latest/#system-overview (besucht am 11. 06. 2019).

Garousi, Vahid, Ebru Göçmen Ergezer und Kadir Herkilolu. „Usage, usefulness and quality of defect reports: an industrial case study". In: *Proceedings of the 20th international conference on evaluation and assessment in software engineering*. ACM. 2016, S. 39.

Hochschulrat. *Richtlinien des Hochschulrates für die koordinierte Erneuerung der Lehre an den universitären Hochschulen der Schweiz im Rahmen des Bologna-Prozesses*. 2015. URL: https://www.admin.ch/opc/de/classified-compilation/20150869/index.html (besucht am 25. 02. 2019).

Hynek, Schlawack. *Python in Azure Pipelines, Step by Step*. 3. Juni 2019. URL: https://hynek.me/articles/simple-python-azure-pipelines/ (besucht am 11. 06. 2019).

Jamie, Cool. *Introducing Azure DevOps*. Microsoft Corporation. 10. Sep. 2018. URL: https://azure.microsoft.com/en-us/blog/introducing-azure-devops/ (besucht am 11. 06. 2019).

Just, Sascha, Rahul Premraj und Thomas Zimmermann. „Towards the next generation of bug tracking systems". In: *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE. 2008, S. 82–85.

Lazaris, Louis. *The Results of The Ultimate CSS Survey*. 16. März 2016. URL: https://www.sitepoint.com/results-ultimate-css-survey/ (besucht am 12. 06. 2019).

Nielsen, Jakob und Thomas K. Landauer. „A Mathematical Model of the Finding of Usability Problems". In: *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*. CHI '93. Amsterdam, The Netherlands: ACM, 1993, S. 206–213. ISBN: 0-89791-575-5. DOI: 10.1145/169059.169166. URL: http://doi.acm.org/10.1145/169059.169166.

Nolan, Ashley. *The Front-End Tooling Survey 2018 - Results*. 25. Juli 2018. URL: https://ashleynolan.co.uk/blog/frontend-tooling-survey-2018-results#css-frameworks (besucht am 12. 06. 2019).

Okken, B. *Python Testing with pytest: Simple, Rapid, Effective, and Scalable*. Pragmatic Bookshelf, 2017. ISBN: 9781680504408.

Python Software Foundation und JetBrains. *Python Developers Survey 2018 Results*. Feb. 2019. URL: https://www.jetbrains.com/research/python-developers-survey-2018/ (besucht am 04. 06. 2019).

René, Fleschenberg. Juni 2017. URL: https://fleschenberg.net/django-architecture-diagram/ (besucht am 07. 06. 2019).

Schroter, Adrian u. a. „Do stack traces help developers fix bugs?" In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE. 2010, S. 118–121.

*Sentry Pricing*. URL: https://sentry.io/pricing/ (besucht am 22. 04. 2019).

Stack Overflow. *Stack Overflow Developer Survey Results 2019*. Apr. 2019. URL: https://insights.stackoverflow.com/survey/2019 (besucht am 04. 06. 2019).

Stocker, Mirko u. a. „Interface Quality Patterns – Communicating and Improving the Quality of Microservices APIs“. In: *23rd European Conference on Pattern Languages of Programs 2018*. Juli 2018. URL: http://eprints.cs.univie.ac.at/5661/.

Thomson, Laura. *Socorro: Mozilla's Crash Reporting System*. 19. Mai 2010. URL: https://blog.mozilla.org/webdev/2010/05/19/socorro-mozilla-crash-reports/ (besucht am 13. 06. 2019).

Wang, Shaohua, Foutse Khomh und Ying Zou. „Improving Bug Localization Using Correlations in Crash Reports“. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR '13. San Francisco, CA, USA: IEEE Press, 2013, S. 247–256. ISBN: 978-1-4673-2936-1. URL: http://dl.acm.org/citation.cfm?id=2487085.2487135.

# B. Glossar

**IRC-Channel** IRC ist ein Internetprotokoll für Online-Chats, das zwar über 30 Jahre alt ist, aber bei Opensource-Projekten immer noch sehr weite Verbreitung findet.

**Pastebin** Eine Webapplikation, bei der Texte hochgeladen und publiziert werden können. Pastebins werden oft genutzt, um Quellcode in Foren oder Chats zu teilen, um den Gesprächsverlauf nicht mit langen Texten zu stören.

**PSF** Die Python Software Foundation.

**Python decorator** Eine "Annotation" wie `@login_required` oberhalb einer Python-Funktion. Ein Decorator ist selber eine Funktion, welche die dekorierte Funktion beliebig verändern bzw. ergänzen kann.

**qutebrowser** Ein auf die Tastaturbedienung ausgelegter Web-Browser, welcher von Florian Bruhin entwickelt wird.

**REST APIs** "Representational State Transfer", "Application Programming Interface". Ein verbreitetes Paradigma für die Architektur einer Programmierschnittstelle, die es externen Applikationen erlaubt, auf eine Webapplikation zuzugreifen.

**Value-Klassen** Klassen für Objekte, die verschiedene Werte gruppieren, und oft wenig bis keine Logik enthalten.

# C. Package-Statistiken

Die nachfolgend aufgeführten Statistiken halfen uns bei gewissen Entscheidungen, um die Popularität von verschiedenen Libraries abzuschätzen. Abgedruckt ist der Stand vom 5. Juni 2019.

Die Download-Statistiken (DL) stammen von der PyPI Stats API[1] unter der Zuhilfenahme von `pypistats`[2].

Diese Statistiken sind nur bedingt aussagekräftig, da insbesondere die Anzahl Downloads nicht zwingend mit der Anzahl Nutzer korreliert. Trotzdem waren es nützliche Anhaltspunkte, um herauszufinden, in welcher Grössenordnung sich die Community um eine Library bewegt.

[1] `https://pypistats.org/api/`
[2] `https://github.com/hugovk/pypistats`

Tabelle C.1.: Statistiken für Python-Module

| Package | DL: Tag | DL: Woche | DL: Monat | GitHub: Used by |
|---|---|---|---|---|
| Flask | 424'503 | 2'063'959 | 9'454'532 | 277'349 |
| Django | 156'890 | 838'385 | 3'242'149 | 258'845 |
| django-dynamic-models | 1 | 2 | 29 | 1 |
| django-dcolumns | 2 | 19 | 89 | 0 |
| attrs | 563'834 | 3'535'660 | 14'483'028 | 45'313 |
| django-colorful | 710 | 3'996 | 17'416 | 243 |
| django-colorfield | 970 | 4'268 | 20'522 | 366 |
| django-crispy-forms | 9'880 | 242'553 | 53'324 | 19'403 |
| django-mailbox | 200 | 795 | 3'705 | 51 |
| Django REST framework | 65'551 | 356'680 | 1'585'499 | 66'960 |
| TinyDB | 2'709 | 57'201 | 14'278 | 1'504 |
| tinymongo | 14 | 74 | 362 | 60 |
| PyMongo | 136'120 | 797'859 | 136'120 | 37'561 |
| Haystack | 60 | 380 | 1'754 | nicht auf GitHub |
| Whoosh | 2'992 | 16'728 | 74'292 | 4'257 |
| pytest | 462'238 | 2'604'867 | 11'318'084 | 102'860 |
| nose | 133'806 | 833'637 | 4'233'947 | 62'214 |
| nose2 | 3'850 | 20'683 | 91'337 | 1'688 |

# D. Test-Log von pytest

```
============================= test session starts ==============================
platform linux -- Python 3.7.3, pytest-4.5.0, py-1.8.0, pluggy-0.12.0
Django settings: crashbin_site.settings (from ini file)
rootdir: /home/florian/proj/crashbin/git, inifile: pytest.ini
plugins: django-3.4.8, cov-2.7.1, mock-1.10.4
collected 126 items

tests/test_api.py .......                                                [  5%]
tests/test_models.py ......................                              [ 23%]
tests/test_signals.py ...                                                [ 26%]
tests/test_templatetags.py ............                                  [ 35%]
tests/test_utils.py .....                                                [ 39%]
tests/test_views.py ................................................     [ 80%]
........................                                                 [100%]

----------- coverage: platform linux, python 3.7.3-final-0 -----------
Coverage HTML written to dir htmlcov
Coverage XML written to file coverage.xml


========================== 126 passed in 10.30 seconds ==========================
```

# E. GUI

## E.1. Wireframes

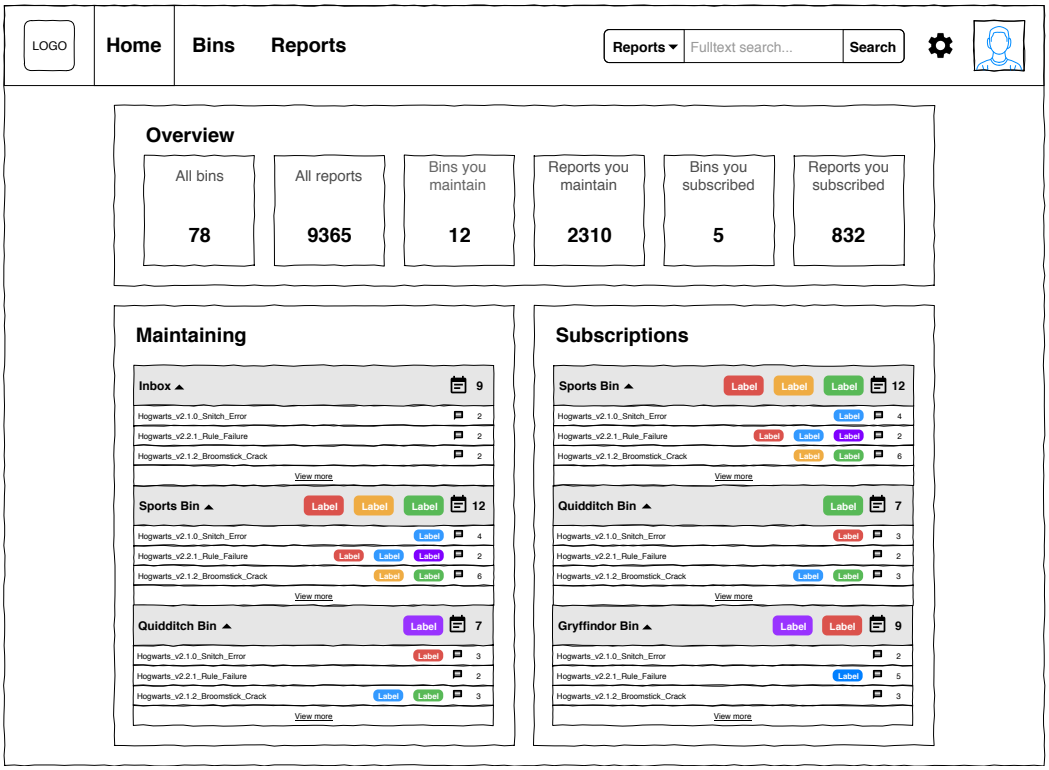



Abbildung E.1.: Wireframe der Home-Ansicht

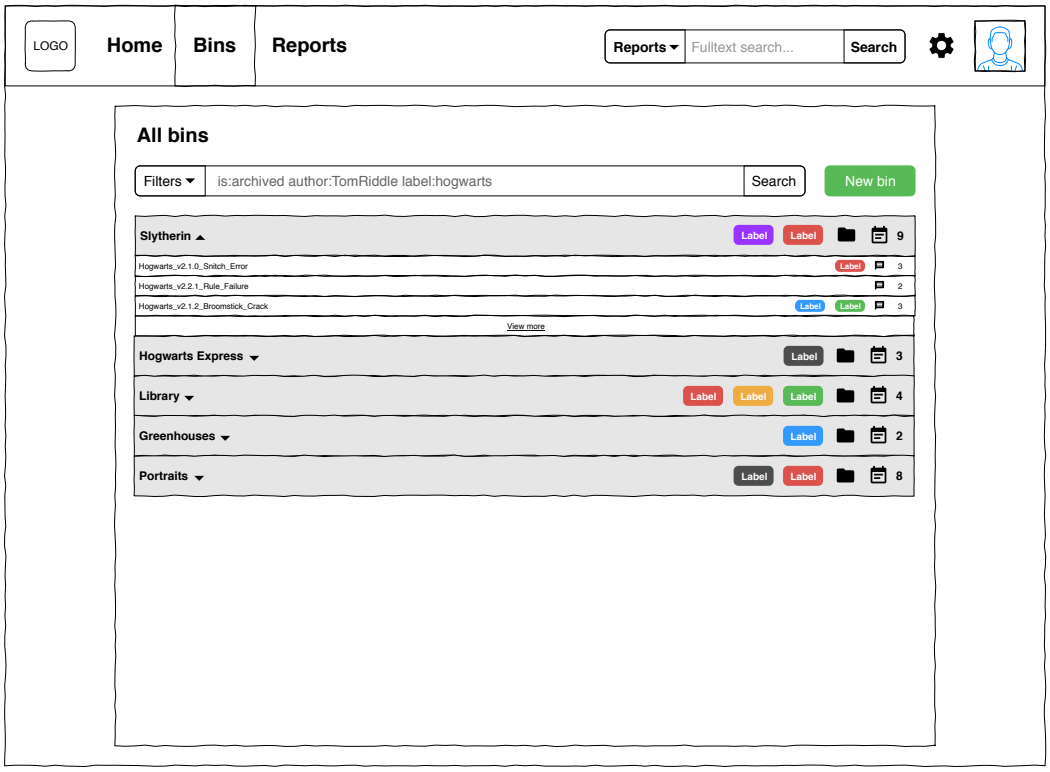



Abbildung E.2.: Wireframe der Bin-Übersicht

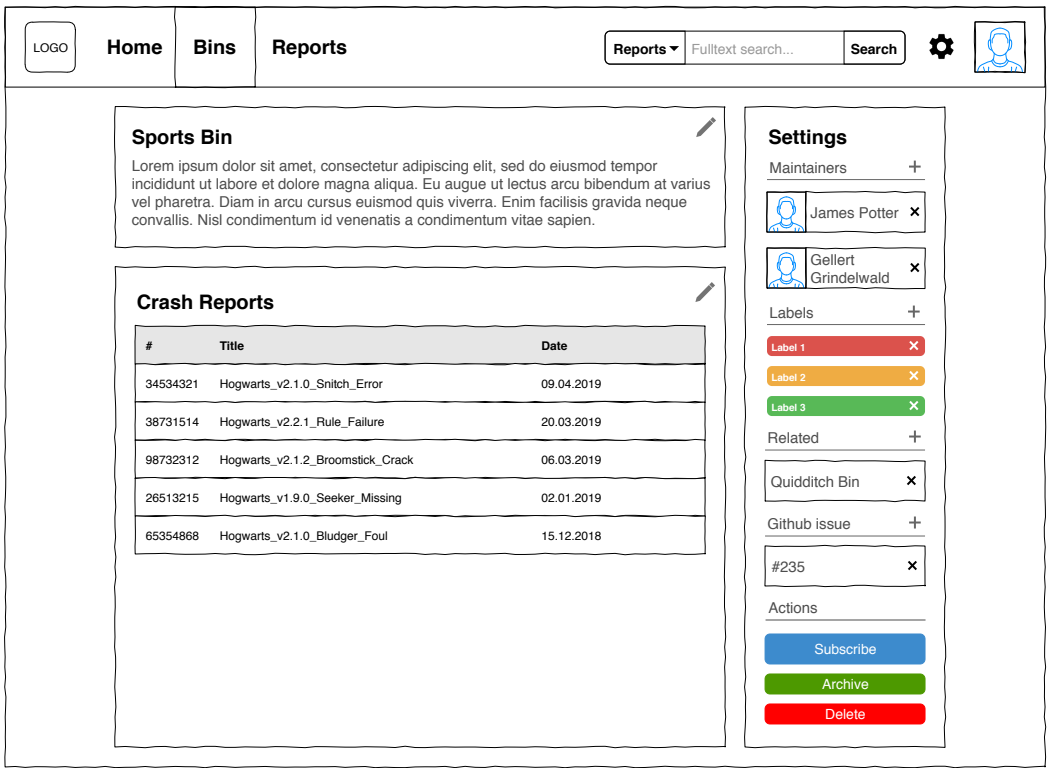



Abbildung E.3.: Wireframe der Detailansicht eines Bins

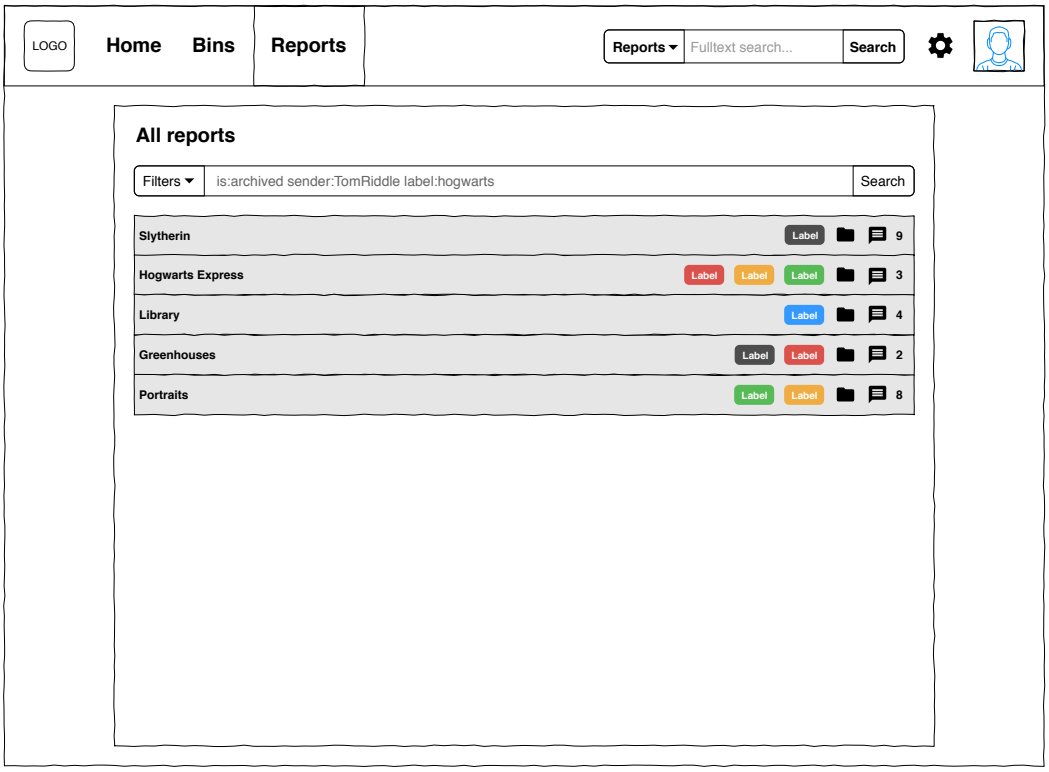



Abbildung E.4.: Wireframe der Report-Übersicht

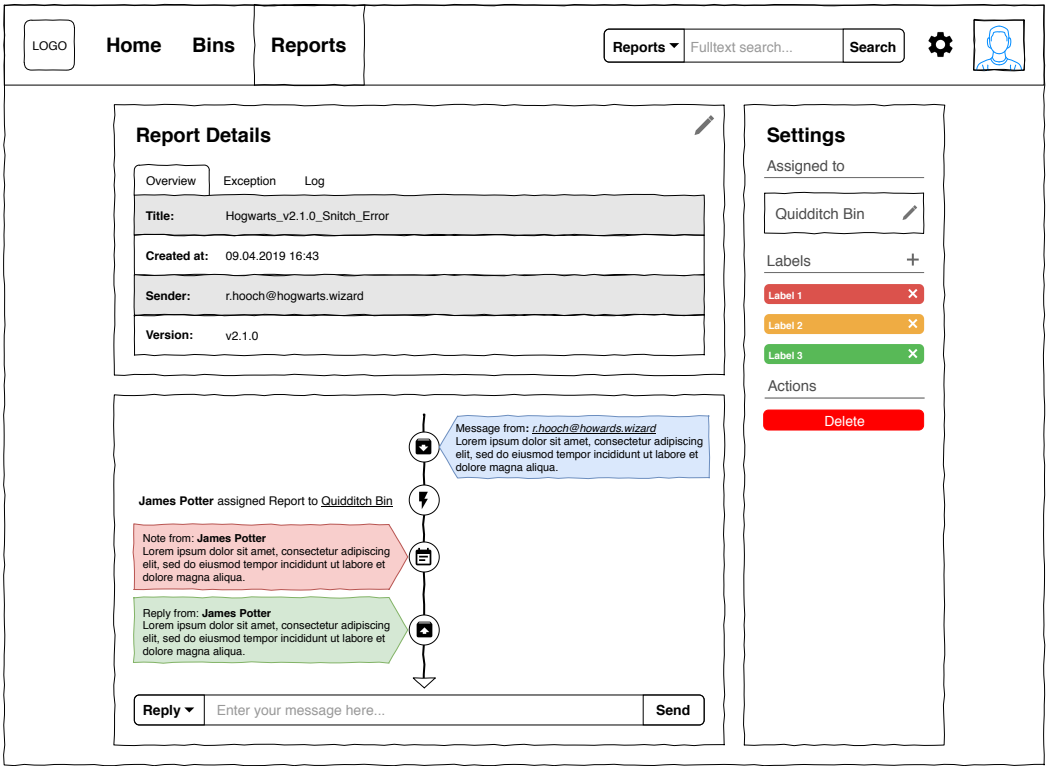



Abbildung E.5.: Wireframe der Detailansicht eines Reports

## E.2. Screenshots

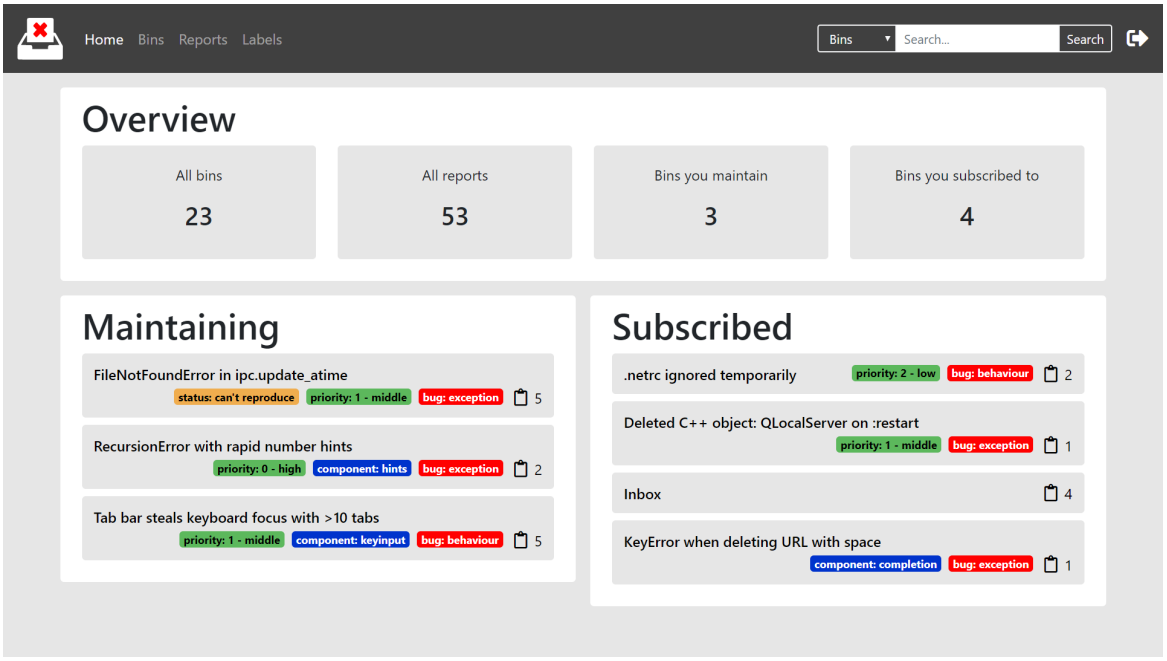



Abbildung E.6.: Home-Ansicht

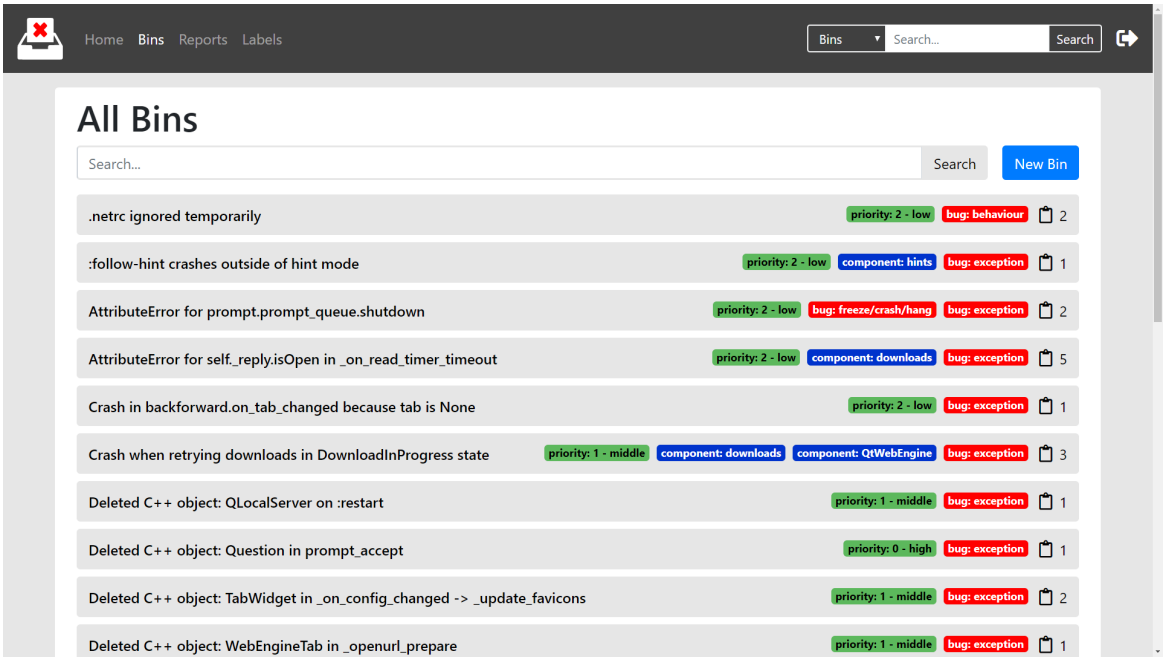



Abbildung E.7.: Bin-Übersicht

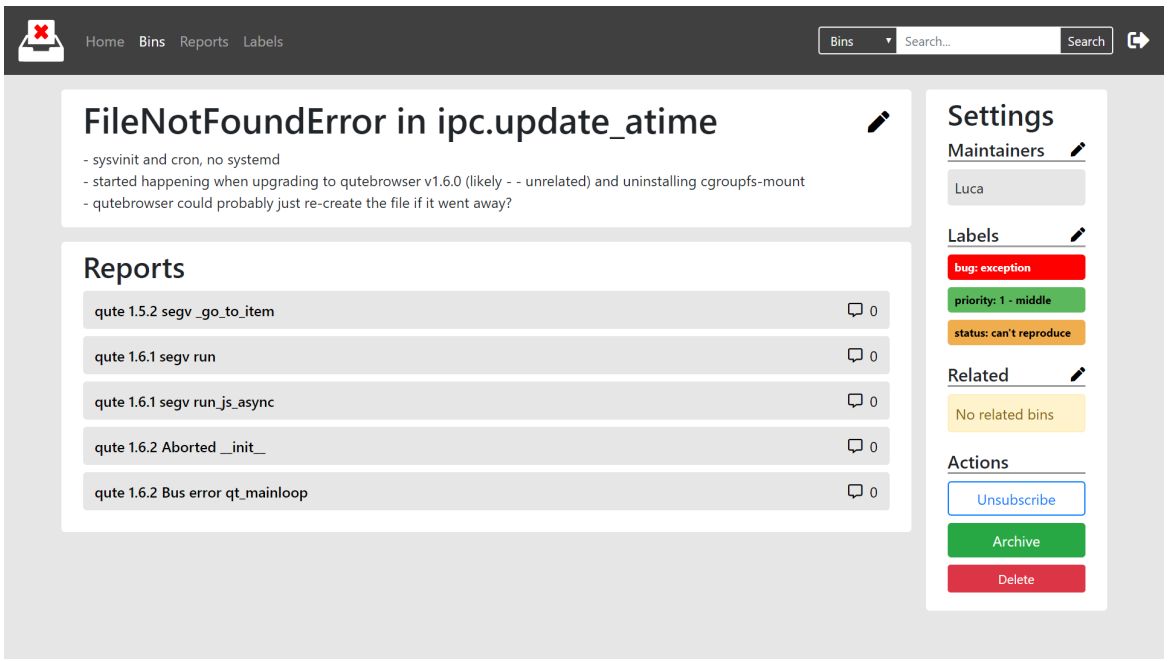



Abbildung E.8.: Detailansicht eines Bins

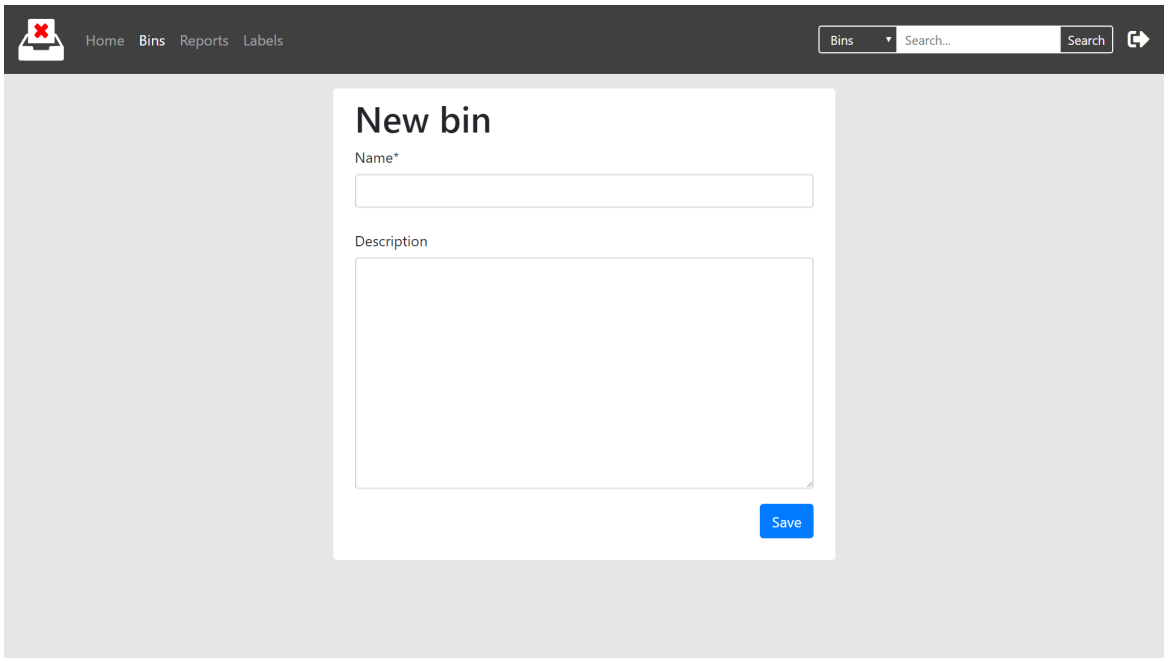



Abbildung E.9.: Formular, um einen Bin zu erstellen oder zu editieren

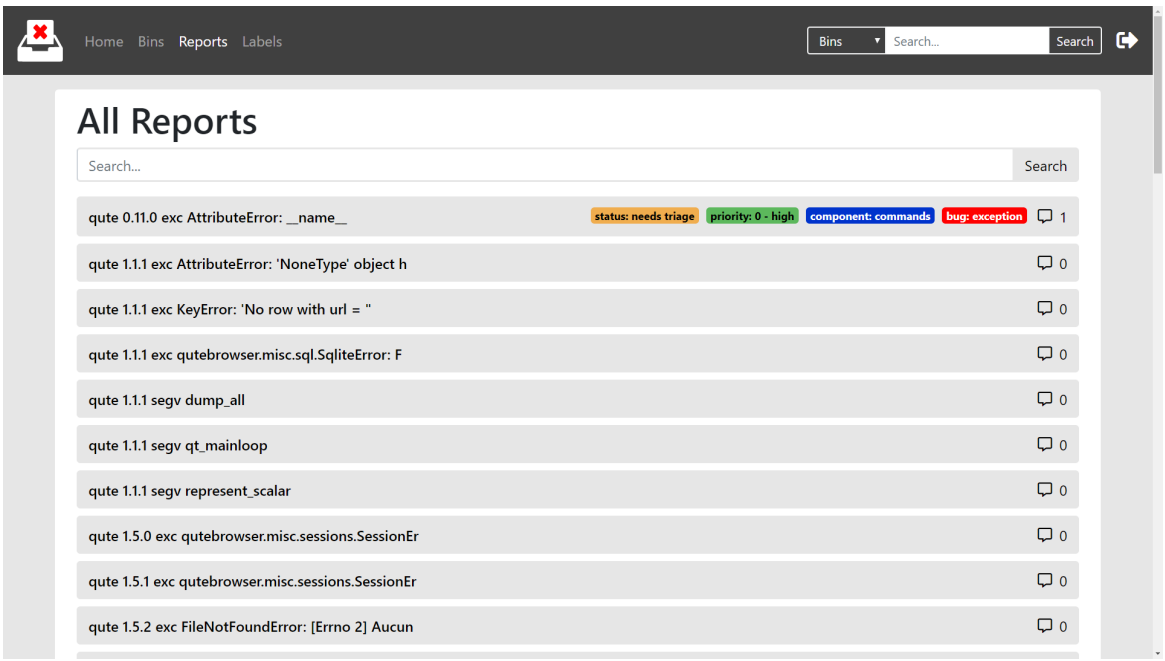



Abbildung E.10.: Report-Übersicht

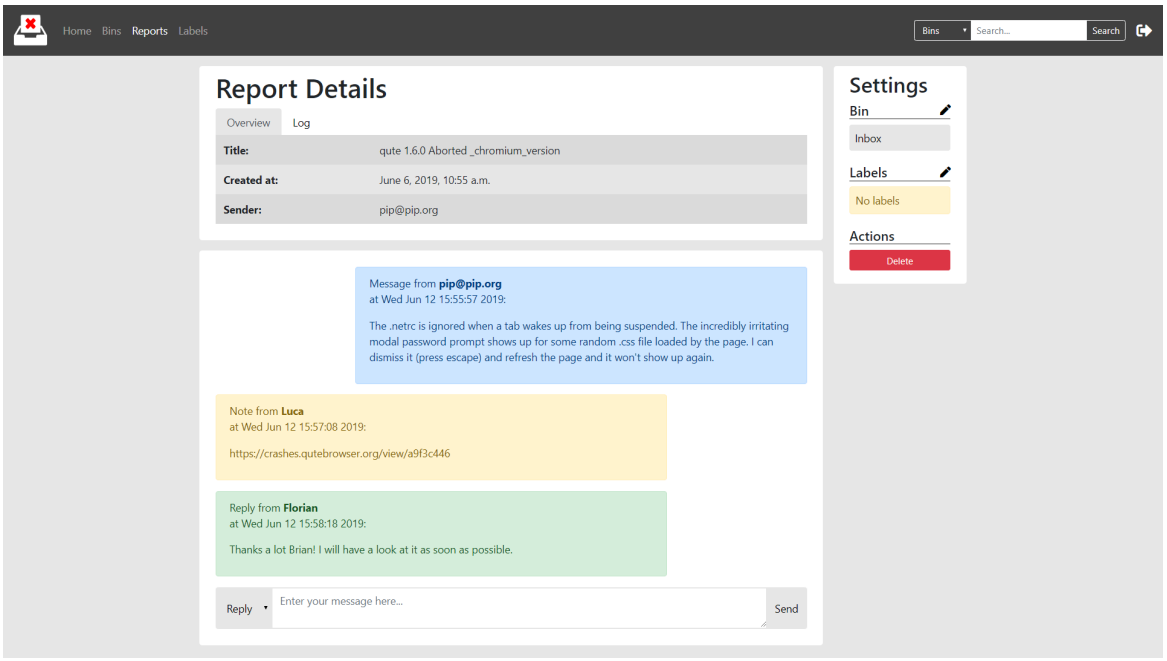



Abbildung E.11.: Detailansicht eines Reports

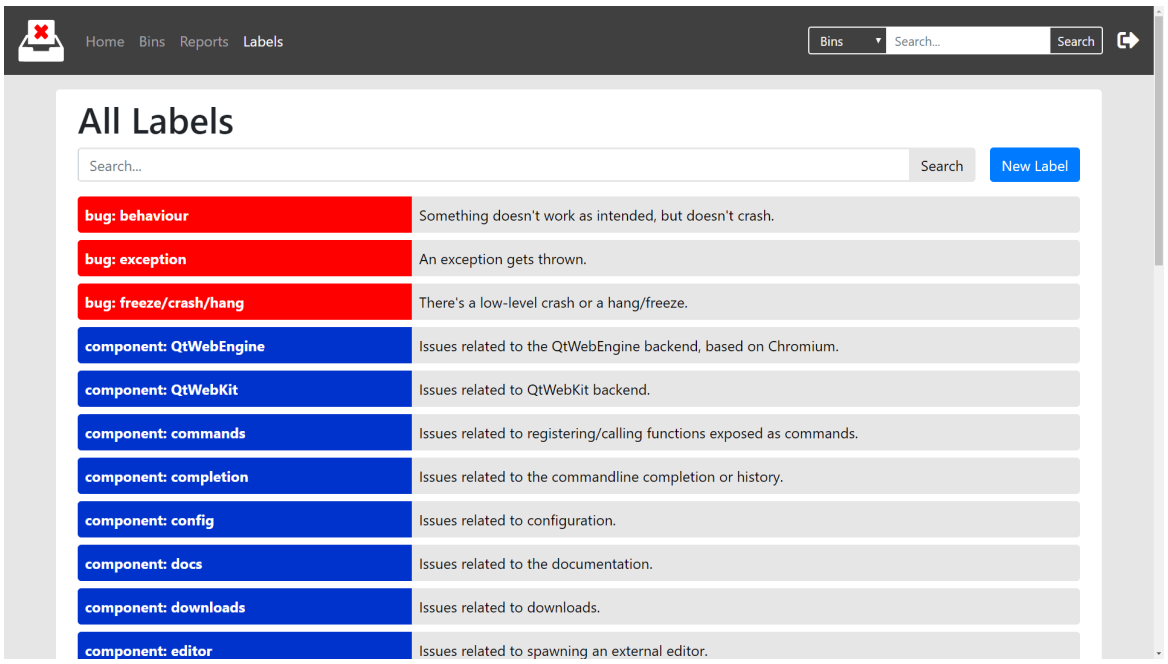



Abbildung E.12.: Label-Übersicht

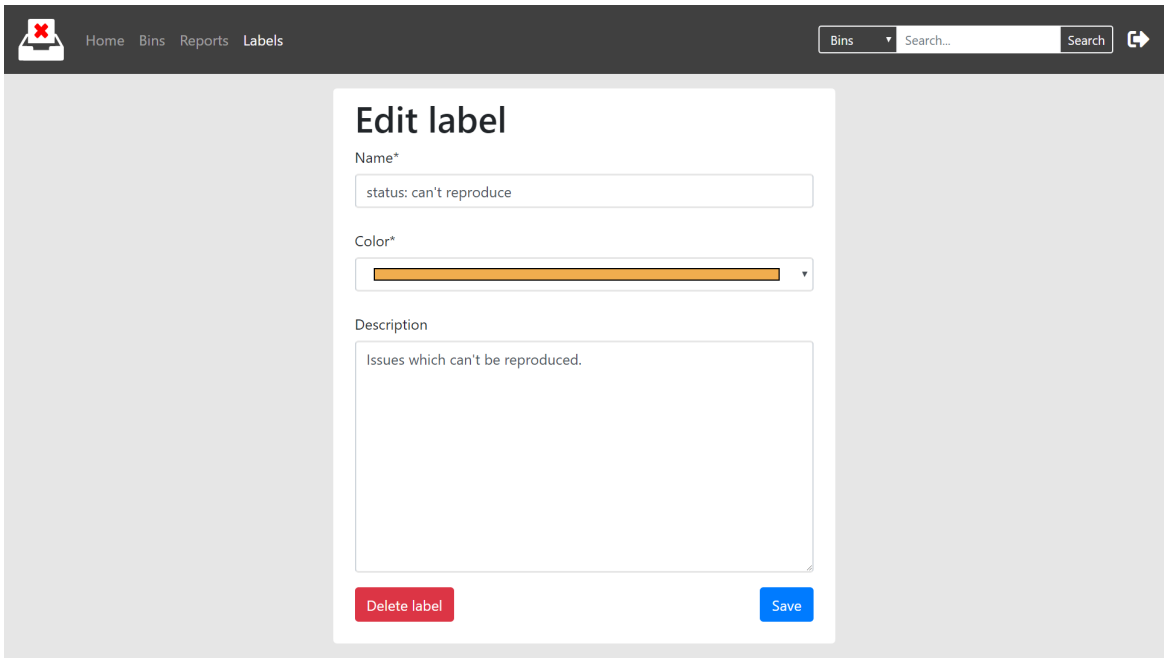



Abbildung E.13.: Detailansicht eines Labels

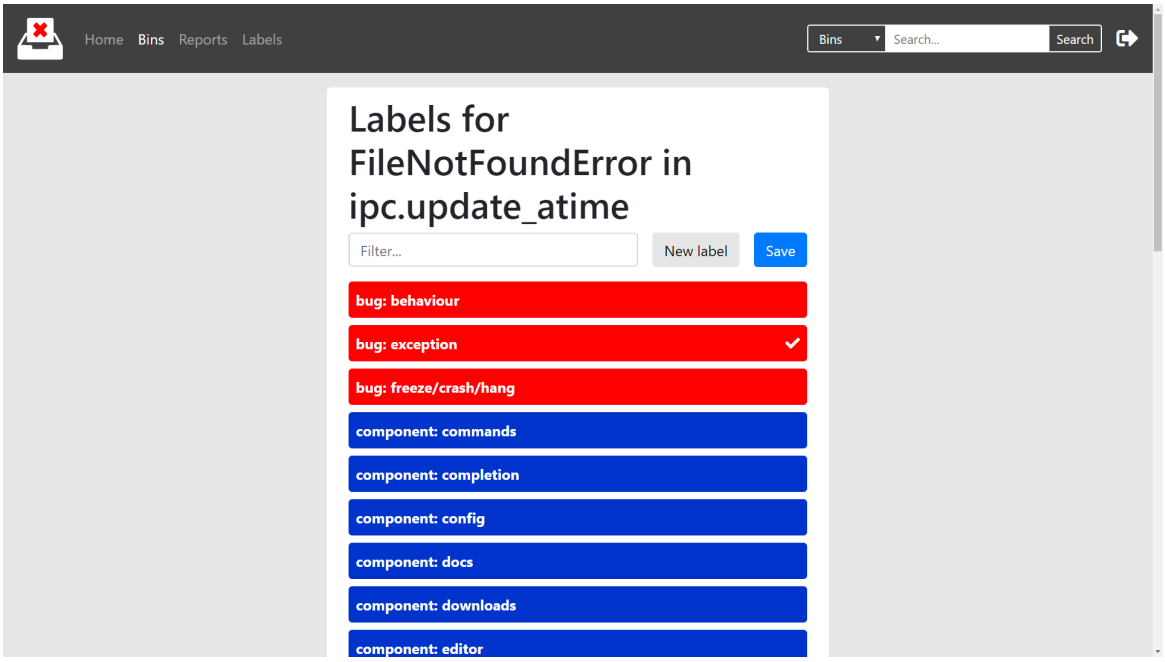



Abbildung E.14.: Settings-Ansicht

# F. Mails

Um herauszufinden, welche Lösungen für die Kommunikation mit Endbenutzern in bestehenden Crash-Reportern vorhanden sind, wurde die folgende Anfrage an verschiedene solche Anbieter versendet:

| | |
|---|---|
| Date: | Thu 25 Apr 2019 19:10:26 CEST |
| To: | support@... |
| From: | Florian Bruhin <me@the-compiler.org> |
| **Subject:** | **Communication with crash reporters** |

```
Hi!

I'm currently looking into various crash monitoring services for my opensource
project.

However, my application only creates a couple of crash reports per week. In
order to build up a community and interact with the people who experienced
issues, I often use the contact information they supplied to follow up.

Does [...] offer anything to facilitate such communication, other than a
custom field with the email address where I'd need to send messages by hand?
What I have in mind is basically some mixture between a crash monitoring
system and a ticket tracking system.

Thanks!

Florian

--
https://www.qutebrowser.org | me@the-compiler.org (Mail/XMPP)
   GPG: 916E B0C8 FD55 A072 | https://the-compiler.org/pubkey.asc
         I love long mails! | https://email.is-not-s.ms/
```

Die Antworten der entsprechenden Anbieter sind auf den folgenden Seiten ersichtlich.

Date: Thu 25 Apr 2019 19:26:28 CEST
To: me@the-compiler.org
From: Stuart from Airbrake <stuart.moore@airbrake.intercom-mail.com>
**Subject: Re: Communication with crash reporters**

Hi Florian,
 Thanks for writing in. Airbrake is an online tool that provides robust exception tracking in your application. In doing so, it allows you to easily review errors, tie an error to an individual piece of code, and trace the cause back to recent changes. Airbrake enables for easy categorization, searching, and prioritization of exceptions so that when errors occur, your team can quickly determine the root cause.
 Airbrake isn't a crash reporter and doesn't provide a way for your customers to communicate with you. You may, however, be able to use Airbrake to track down a particular customer's issue and see what happened.
 Please let me know if you have any other questions.

> On Thu, Apr 25, 2019 at 10:03 AM, "Florian Bruhin"
<me@the-compiler.org> wrote:
>
> Hi!
>
>
>
> I'm currently looking into various crash monitoring services for my
opensource
>
> project.
>
>
>
> However, my application only creates a couple of crash reports per week. In
>
> order to build up a community and interact with the people who experienced
>
> issues, I often use the contact information they supplied to follow up.
>
>
>
> Does Airbrake offer anything to facilitate such communication, other than a
>
> custom field with the email address where I'd need to send messages by hand?
>
> What I have in mind is basically some mixture between a crash monitoring
>
> system and a ticket tracking system.
>
>
>
> Thanks!
>
>
>
> Florian
>

Date: Fri 26 Apr 2019 14:40:28 CEST
To: Florian Bruhin <me@the-compiler.org>
From: Matt Young (Bugsnag) <support@bugsnag.zendesk.com>
**Subject: [Bugsnag] Re: Communication with crash reporters**

```
##- Please type your reply above this line -##

Your request (18251) has been updated. To add additional comments, reply to
this email.
-------------------------------------------

Matt Young, Apr 26, 05:40 PDT

Hi Florian

Thanks for reaching out.

No, we don't offer any tools to aid communication with users directly. Only
ways of drilling down to identify which error are affecting which users.

We do support a number on integrations so you may be able to configure Bugsnag
to integrate with another system which supports ticket tracking:
https://docs.bugsnag.com/product/integrations/

Best regards
Matt

-------------------------------------------

Florian Bruhin, Apr 25, 10:12 PDT

Hi!

I'm currently looking into various crash monitoring services for my opensource
project.

However, my application only creates a couple of crash reports per week. In
order to build up a community and interact with the people who experienced
issues, I often use the contact information they supplied to follow up.

Does Bugsnag offer anything to facilitate such communication, other than a
custom field with the email address where I'd need to send messages by hand?
What I have in mind is basically some mixture between a crash monitoring
system and a ticket tracking system.

Thanks!

Florian

--
https://www.qutebrowser.org | me@the-compiler.org (Mail/XMPP)
   GPG: 916E B0C8 FD55 A072 | https://the-compiler.org/pubkey.asc
         I love long mails! | https://email.is-not-s.ms/

Attachment(s):
```

Date: Mon 29 Apr 2019 00:10:20 CEST
To: raygun.com@the-compiler.org
From: Georgina from Raygun
<georgina.wollerman-effaaefbbfa8@raygun.intercom-mail.com>
**Subject: Re: Contact type: Sales - raygun.com@the-compiler.org**

Hi Florian
Thanks for getting in contact.
Basically the way we handle this in Raygun is:
- Set up Crash Reporting and set up user tracking to monitor the users visiting your app
- Once in Crash Reporting you can click into a particular error group (crash report) you are interested in. Within that error group you will find (in the right-hand side bar) a link to view a list of all users affected by the error
- When viewing this list, you have the option to export the list as a CSV file, containing all user data
- Using the CSV file, you can bulk email affected users
The benefit Raygun provides here is that you don't have to rely on your users to send in contact information for follow up, as we collect this data for you. This ensures you can reach out to any and all users affected by errors in your application.
Additionally, not 100% sure of your software development process but thought it would be worth mentioning that we have integrations with Github and Jira. This will allow you to create a ticket for a particular error from within an error report in Raygun.
I hope this helps, but let me know if you have any further questions.
Kind regards
Georgiina

> On Sat, Apr 27, 2019 at 03:50 AM, "Joyce Padua" <joyce.padua@raygun.intercom-mail.com> wrote:
>
> Hi Florian,
>
> Thanks for getting in touch. I have assigned your request to my colleague in Accounts, who will be able to help.
>
> As a note, our Accounts team is located in New Zealand and it is currently the weekend there so there will be a delay in response. They will respond to you once their working week resumes and they've had the chance to review your request.
>
> Thank you, Joyce
>
>> On Fri, Apr 26, 2019 at 07:16 PM, "Florian Bruhin" <raygun.com@the-compiler.org> wrote:
>>
>> ====Sent from raygun.com/about/contact form==== Contact type: Sales Hi! I tried sending a mail to sales@raygun.com but I'm not sure if it came through, so I'm trying via the contact form again. I'm currently looking into various crash monitoring services for my opensource project. However, my application only creates a couple of crash reports per week. In order to build up a community and interact with the people who experienced issues, I often use the contact information they supplied to follow up. Does Raygun offer anything to facilitate such communication, other than a custom field with the email address

Date: Fri 26 Apr 2019 01:10:13 CEST
To: me@the-compiler.org
From: Jessica from Rollbar <support@rollbar.com>
**Subject: Re: Communication with crash reporters**

Hello Florian,
Unfortunately, we don’t offer anything in terms of a ticket tracking system--just the crash monitoring!
Let me know if you’re interested in finding out more about Rollbar.
Thank you, Jessica

> On Thu, Apr 25, 2019 at 10:10 AM, "Florian Bruhin" <me@the-compiler.org> wrote:
>
> Hi!
>
>
>
> I’m currently looking into various crash monitoring services for my opensource
>
> project.
>
>
>
> However, my application only creates a couple of crash reports per week. In
>
> order to build up a community and interact with the people who experienced
>
> issues, I often use the contact information they supplied to follow up.
>
>
>
> Does Rollbar offer anything to facilitate such communication, other than a
>
> custom field with the email address where I’d need to send messages by hand?
>
> What I have in mind is basically some mixture between a crash monitoring
>
> system and a ticket tracking system.
>
>
>
> Thanks!
>
>
>
> Florian
>
>
>
> --
>
> https://www.qutebrowser.org | me@the-compiler.org (Mail/XMPP)
>

Date: Thu 25 Apr 2019 19:42:22 CEST
To: Florian Bruhin <me@the-compiler.org>
From: Katie Byers (Sentry) <support@sentry.zendesk.com>
**Subject: [Sentry] Re: Communicating with crash reporters**

```
##- Please type your reply above this line -##

Your request (21171) has been updated. To add additional comments, reply to
this email.
-------------------------------------------

Katie Byers, Apr 25, 10:42 PDT

Good morning!

Though we have the ability to collect [user
feedback](https://docs.sentry.io/enriching-error-data/user-feedback/?platform=b
rowser), we don't currently have a mechanism for reaching back out to
customers. Happy to put in the feature request, though!



Thanks,
Katie
Sentry Support Engineer

-------------------------------------------

Florian Bruhin, Apr 25, 10:06 PDT

Hi!

I'm currently looking into various crash monitoring services for my opensource
project.

However, my application only creates a couple of crash reports per week. In
order to build up a community and interact with the people who experienced
issues, I often use the contact information they supplied to follow up.

Does Sentry offer anything to facilitate such communication, other than a
custom field with the email address where I'd need to send messages by hand?
What I have in mind is basically some mixture between a crash monitoring
system and a ticket tracking system.

Thanks!

Florian

--
https://www.qutebrowser.org | me@the-compiler.org (Mail/XMPP)
   GPG: 916E B0C8 FD55 A072 | https://the-compiler.org/pubkey.asc
         I love long mails! | https://email.is-not-s.ms/

--
https://www.qutebrowser.org | me@the-compiler.org (Mail/XMPP)
```