Meagle - Crowdsourced software data with community-moderated software reviews

Tom Macmichael

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2017

Abstract

Finding impartial information about a given piece of software is not easy: there is not a single place users can visit to find and contribute information in a consistent manner. This is becoming an increasing concern as society is using more software than ever before.

This project created a new solution for this: Meagle.org: a free, open-source website that allows a community of users to review pieces of software, moderated in a way that allows the best, impartial reviews to be highlighted, while still allowing users to voice their opinions. The database of software that can be reviewed is crowdsourced, meaning the data is added and maintained by users, allowing any piece of software to be reviewed. The overall platform requires no paid moderators, or data imported from other sources: it is designed to moderate and maintain itself.

Other websites offering similar services, online communities, and platforms with crowdsourcing abilities, were analysed first – highlighting key features and issues. This information was used to make informed design decisions to build something that was better than existing platforms, while building on ideas that worked well elsewhere.

This report outlines the complete development cycle for building Meagle: designing, engineering, building, testing, releasing, and evaluating. This report demonstrates the paucity of existing platforms for software reviews and the need for Meagle. The report also demonstrates the best practices in modern web development, which were successfully used throughout the development and release.

Acknowledgements

I would like to thank my supervisor, Sebastian Maneth, for his help and advice while writing this report. I would also like to thank my family for supporting me throughout university, particularly my parents.

Table of Contents

Chapter 1

Introduction

The average smartphone user has 27 applications installed on their phone [1]. These applications, and devices, receive regular software releases from their developers, however not all users are necessarily aware of the updates, nor the advantages and disadvantages of installing these [2]. Currently, there is not a single place to go to find information regarding these updates. There are numerous websites that review software but these are generally found on bloated, commercial websites with clunky user interfaces. These sites are slow, cluttered with adverts, and not always optimised for mobile devices. Another issue is that information regarding software releases is often from the developers themselves, which can give misleading expectations of the software's performance and usability.

The aim of this project was to build a website that contained reviews of any piece of software. The underlying database of software was to be crowdsourced, meaning the data would be added and maintained by other users. The ultimate goal was to release scalable, modern software using agile development practices such as continuous integration. The breadth of this project covered project management, human–computer interaction, user interface design, systems design, and backend development. The end community had to have the ability to moderate itself in a term used throughout this report as "self-moderation", while the site also had to have the ability to grow itself through crowdsourcing – in a term used throughout this report as "self-maintaining". These practices, and terminology, and explained within this report.

By building an open-source platform, that was community moderated, Meagle could offer an impartial site that was free of advertising and with a focus on the content instead. This would allow users to contribute information easily regarding their experience of a piece of software, while easily finding information from other users.

This report outlines the full development process behind this, including the research around similar platforms.

1.1 Contributions

This is a summary of the final work produced from the project:

- An analysis of existing crowdsourcing platforms, software review websites, and self-moderating online communities.
- A complete site published at Meagle.org – one place where people can find, and give, information about any version of software.
- A solution for self-moderating platforms.
- A solution for crowdsourcing data.
- A responsive user interface.
- An open-source repository [3] with documentation, allowing users to contribute and extend to the project, or apply the codebase for another use case, as discussed in Section 6.2.
- A complete build and release process, allowing for continual release of new code in a rapid, safe way.

1.2 Structure

- **Chapter 2** discusses the current solutions for software reviews prior to Meagle, as well as looking at how other platforms handle reviews. It also gives an overview of what crowdsourcing is and how it used today.
- **Chapter 3** outlines the development process of Meagle, and the tools used. It also describes key framework concepts used throughout the report.
- **Chapter 4** is split into the different stages of Meagle's development, explaining what, why, and how features were introduced.
- **Chapter 5** is a final evaluation of the platform: its objective performance, usability, as well as feedback from users.
- **Chapter 6** concludes the work completed throughout the project and introduces potential ideas for further work.

Chapter 2

Background

The scope of this project covered several different fields and so it was important to look at these and see what work had been completed already and how it could be improved upon or used.

2.1 Crowdsourced platforms

As the project planned to use crowdsourcing, it was necessary to analyse the different crowdsourced platforms that exist today to see how they work. Firstly it was important to define what "crowdsourced" meant. The phrase was first coined in 2006 by Jeff Howe as a play on words from the phrase "outsourcing" - "it's not outsourcing; it's crowdsourcing" [4]. With outsourcing, third parties are paid to complete work for a company or individual; crowdsourcing, however, harnesses the power of thousands of (unpaid) users to do the work instead.

There are numerous examples of crowdsourcing being used successfully online. These can be split into two categories: implicit – whereby contributing users *are not* aware that they are contributing, and explicit – whereby users *are* aware that they are contributing [5].

At the end of 2007 Facebook began translating its site using crowdsourcing. By 2008 the community was so sophisticated that the entire site has been translated into French within 24 hours [6]. As users were actively aware of their contributions, in the form of the translation tools developed by Facebook, this is an example of an explicit crowdsourcing platform. Explicit crowdsourcing generally works by rewarding users with virtual points, often referred to as "karma" [6].

One of the best known implicit crowdsourced platforms is reCAPTCHA. ReCAPTCHA, now owned by Google [7], is used on thousands of websites as a tool of verifying whether a visitor is a human or a robot. It invites users to identify characters in a distorted image – if they correctly identify the characters, they are identified as a human. The user, however, is actually shown two images merged together: the first is an image with a known answer, and the second image is

unknown. If the user correctly identifies the characters in the first image, it is assumed they have also correctly entered the characters for the second image. As thousands of users are shown the second image, the system can learn the correct transcription of the image – this is how the reCAPTCHA implicitly crowdsources transcriptions of old books that have been scanned [5].

There is also a distinction to be made between a *community* platform and a *crowdsourced* platform. In crowdsourced platforms, many users come together to achieve a certain goal with an objective answer (such as translations), whereas in community platforms many users come together to connect or comment in a more subjective way (such as on YouTube). With Meagle, the aim was to build a *crowdsourced* database of software, which then a *community* would review and self-moderate. While reviews can sometimes be objective, discussing features or bugs introduced, they often contain a subjective opinion too.

2.2 Existing software review sites

There are several different types of software reviews sites. These can generally be split into two categories: software review sites written by journalists (such as CNET.com, wired.com, theguardian.com, techcrunch.com) and sites that contain reviews written by a community of users (such as trustradius.com, alternativeto.net). These were analysed against the following aspects:

- **User interface**. Is the site mobile-friendly? Is it cluttered? How is the review displayed? Is it easy to find important information?
- **Difficulty of finding a piece of software**. How easy is it to find a specific piece of software on the site?
- **Quality of information**. How thorough are the reviews? How impartial is the information?

2.2.1 Traditional review sites

Four traditional websites were analysed as part of the background research: CNET.com, wired.com, techcrunch.com and trustreviews.com.

Sites such as CNET and TechCrunch give extensive articles written by one person about a given software update. The format of these sites follow that of a traditional newspaper article: rather than just objectively outlining the advantages and disadvantages of an update and its features, the articles put this information in terms of the overall context of the company, by including stock prices for example.

User interface. CNET, Wired and TrustedReviews all do not have a responsive

design[1] – instead, they have a separate mobile website. This approach often results in inconsistency as the two websites have been developed independently, which can result in a different subset of features across platforms. All sites feature large adverts that clutter the site, making it harder to find the key information. This is exemplified in Figure 2.2, which shows a CNET review page headed with a very large intrusive video at both the top and bottom-right of the page, as well as including a large banner advert. The load times on all three sites took longer than 10 seconds, which is particularly slow[2].

One aspect of CNET that was pleasing was the use of a breadcrumb navigation system – shown in Figure 2.1. This concept is named after the children's story of Hansel and Gretel, who leave breadcrumbs on the floor so they can find their way back home. Similarly, with breadcrumb navigation, users are shown a trail of pages representing how they reached the page they are currently visiting. This gives users the ability to go back to pages previously visited, as well as placing their current page in context of the site's hierarchy [9].

Finding a piece of software. As these sites are not *solely* software review sites, finding a review for a given piece of software is not easy. As the sites contain many news articles, when searching for a review there is a mix of software reviews alongside news stories about the software. On all three sites, it is easier to find a review on the website using a search engine than use the site's own search engine. Most of the sites do not have reviews for more minor releases, such as iOS 10.1.

Quality of information. The articles on the sites, however, are very informative. They outline key features, bugs, and recommendations. As the author generally comes from a technology background, readers know that the information is relatively reliable. However, as the source of the information is just one author there is a question of whether a review is biased – it might be in the author's interest to not mention all the problems with a piece of software. An example of a review on CNET is shown in Figure 2.2 which shows how articles contain a succinct summary of a piece of software at the top of the page.

CNET › Software › Operating Systems › Apple iOS 10

Figure 2.1: Screenshot of breadcrumb navigation used on CNET.com

[1]Responsive design is a term given to sites that share a single codebase and stylesheet yet the website still displays optimally on all devices, regardless of screen size. This is discussed in Section 3.9.

[2]This is according to a Pingdom speed test [8] which uses real instances of the Google Chrome browser to test the speed of websites.

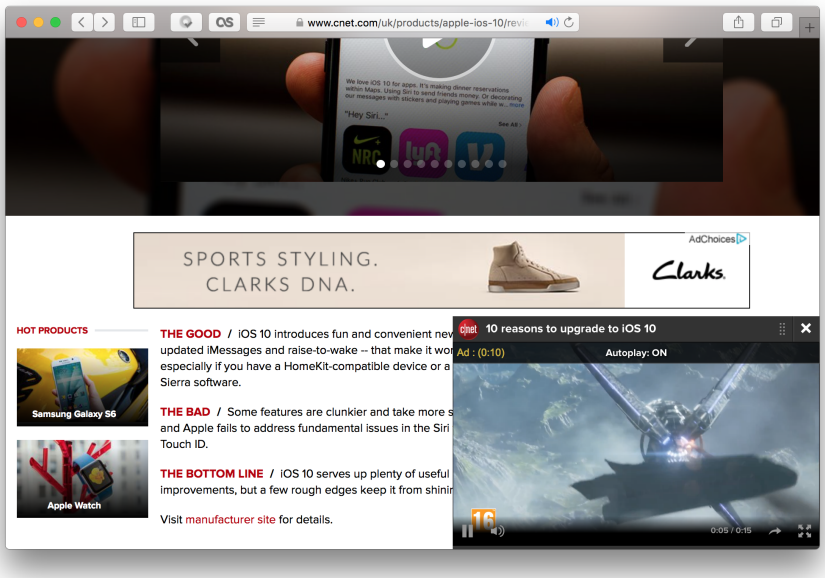

Figure 2.2: Screenshot from a CNET.com review of iOS 10

2.3 Community platforms

Two websites that have software information provided by a community of users are TrustRadius.com and AlternativeTo.net. Neither site offers software reviews for a specific version of consumer software, however as they are a moderated by a community of users writing information about software, they were useful to look at.

TrustRadius

TrustRadius focuses on business-to-business software and so it does not include most consumer software such as Android, iOS and Windows.

User interface. TrustRadius is mobile-optimised with a responsive design. The interface feels quite cluttered with lots of unnecessary information on the page. However, they have a good use of tooltips: these are small popups that appear when you hover over an element on the site. An example for a review on TrustRadius, with a tooltip popup being displayed, can be seen in Figure 2.3. TrustRadius, like CNET, uses a breadcrumb navigation system.

Finding a piece of software. TrustRadius features a large search box on its homepage making it very easy to find a piece of software. The search box has the ability to search specifically for reviews, or for articles, too – this means the two do not get combined in one result page, like on CNET.

Quality of information. TrustRadius allows any user to leave a review on a piece of software: it asks users a small set of questions such as the use cases of

the software, the "pros and cons" of the software, and finally whether they would recommend it. This is accompanied by a star rating out of ten. Users can break down reviews by the review metadata itself (such as the review date) or by an author's background (such as the size of the company they are from). A notable feature used on TrustRadius is its "tr score". This is a way of summarising whether users generally recommend a piece of software. This takes into account the authenticity of the author, how long the review is, and when the review was written.

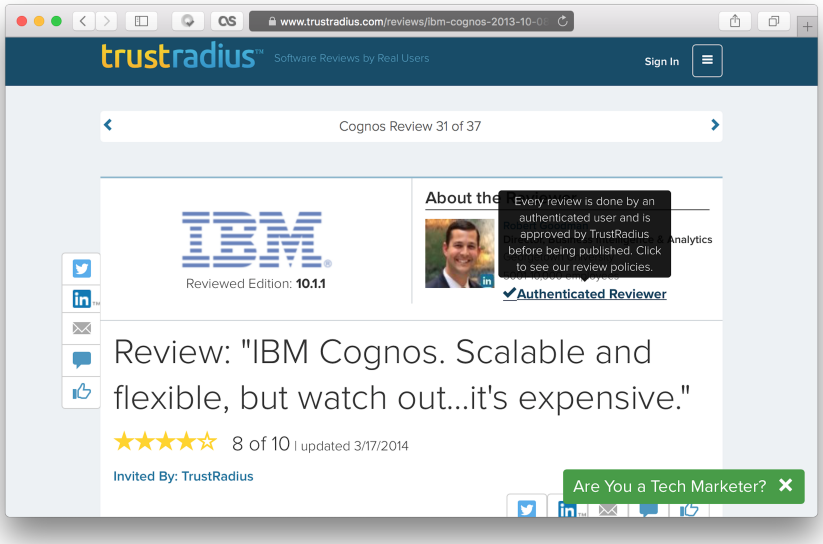

Figure 2.3: Screenshot from a TrustRadius.com review

2.3.1 AlternativeTo

AlternativeTo is not specifically a software update review platform, however it contains information about software provided by a community that moderates itself, as well as crowdsourcing some data. Users have collectively built a database of pieces of software and how they are related. The community can then vote on alternatives for a given piece of software. Ultimately, the site's main use case is the ability to compare different pieces of software that do a certain task, or for finding an open-source alternative to a commercial piece of software.

The site does also offer software reviews written by users, however these reviews are for the software in general, rather than a particular version of software: for example, there are reviews for "Google Chrome", but it is not possible to see reviews for "Google Chrome version 35" – there is no distinction.

Key features

- Ability to add new pieces of software.

- Ability to add features to a piece of software, in the form of tags, as shown in Figure 2.9.
- Ability to suggest alternatives to pieces of software.
- Ability to comment on pieces of software.
- Ability to write reviews about a piece of software. Figure 2.8 shows an example of a user's review.
- Ability to feedback whether others' comments, reviews, contributions are good or not.

Site moderation

Anyone can add a piece of software to AlternativeTo. The process involves signing up, providing the requested meta information regarding the software – such as the developer, then submitting it to the website. The software is then manually reviewed by one of the site's team before going live on the site. This is a different approach from what Meagle aimed to achieve: *complete* crowdsourcing and self-moderation by the community – no need for manual moderators.

AlternativeTo also has several, subtly different, feedback mechanisms:

- **Liking**. Software can be *liked* by users. This is a way of approving a piece of software and is not a comment towards the contribution itself: it is simply a question of whether or not the user likes that piece of software. Each piece of software is labelled with the number of likes it has received.
- **Helpful/not helpful** buttons are used on comments and reviews, as a way of filtering out bad or unhelpful content. Users can then see how many users voted a review as "helpful" or "not helpful".
- **Yes/no** buttons are used for the question "Good alternative?". This is the mechanism for verifying whether a user's suggestion that a piece of software is alternative to another one, is correct or not. The results for this question are not shown and are presumably used by a hidden algorithm.
- **Agree/disagree** buttons are used to moderate the comments on other's suggestions. Again, the results for this question are not displayed directly to users.

Although using multiple different systems might sound quite confusing, the design implemented by AlternativeTo makes it seem intuitive and clear what is being asked and why it is being asked. These different types of voting systems are shown below.

Figure 2.4: Screenshot of the helpful/unhelpful voting system on AlternativeTo.net

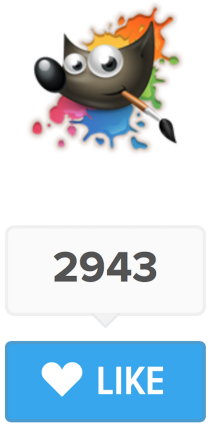

Figure 2.5: Screenshot of the "like" voting system on AlternativeTo.net

Figure 2.6: Screenshot of the yes/no voting system on AlternativeTo.net

Figure 2.7: Screenshot of the agree/disagree voting system on AlternativeTo.net

Summary

User interface. AlternativeTo is mobile-optimised with a responsive design. Some aspects of the design are very clear – such as the alternatives for a piece of software. This part of the site uses clear tags and like buttons to show information. The header for the site contains large amounts of text and colours which makes the page feel like there is lots going on.

AlternativeTo utilises tags to display information on its site about the features of a piece of software. These tags are a pleasing summary of a piece of software, which are also useful for discovering other software that has a particular feature.

Finding a piece of software. There is a very clear search box on every page of the site, making it very easy to find a piece of software. The navigation of the site is relatively simple as the site's hierarchy is just one level; there are no subpages.

Quality of information. The site contains lots of useful information. They have crowdsourced large amounts of metadata on pieces of software, such as the operating system they were developed for, its features, the developer, and a selection of alternatives. Reviews for pieces of software were mixed – generally short and subjective.

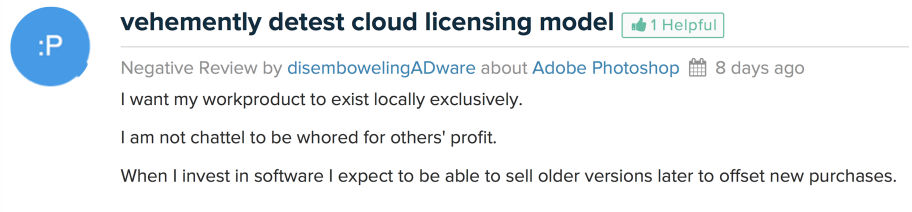

Figure 2.8: Screenshot of a user's review on AlternativeTo.net

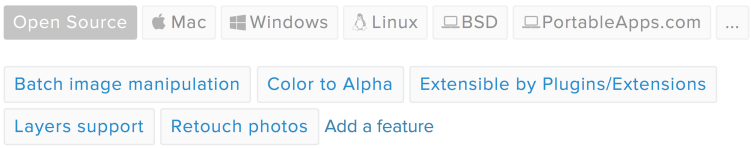

Figure 2.9: Screenshot of tags being used on AlternativeTo.net

2.4 Other platforms with review systems

After analysing what products existed specifically for reviewing software, I wanted to look at how platforms in general do reviews.

2.4.1 StackOverflow

StackOverflow is an online community of programmers who ask programming-related questions which are then answered by other users. Users can then vote on others' answers, and finally the author of the question then can pick a "correct" answer. This system is how StackOverflow curates itself, bringing the best answers to the top and the worst answers to the bottom: an excellent example of community curation and moderation.

As programming questions generally have a fixed, true answer, StackOverflow can be described as a *community* of users *crowdsourcing* answers to programming questions. A screenshot of a StackOverflow question page can be seen in Figure 2.11.

Key features

- The ability to provide feedback on someone's answer is done through a vote: this can be either positive or negative ("upvote" and "downvote" respectively).
- The ability to write formatted answers with Markdown[3].

[3]Markdown is a lightweight language used as an alternative to HTML on websites, allowing users to format their posts [10].

- Answers with the most votes are placed at the top – answers are organised by the number of votes.
- Questions have tags, which can be used to find other related questions, as shown in Figure 2.11.
- The author can pick a correct answer, which is then highlighted with a green tick.
- Below each answer is information about the author – such as their "reputation" and "badges" on the site (shown in Figure 2.10).

StackOverflow's overall design feels very clean, even though it includes banner adverts. A lot of the concepts were applicable to Meagle and were an example of how a community can moderate others' answers. It is interesting to note that the "correct" answer is not the answer with the most votes: but the answer which is chosen as correct by author, and can be chosen at any time. After a correct answer is chosen, users can continue to give answers and the author can change the correct answer: there is no time restriction.

Site moderation

"Reputation" on StackOverflow is a measure of how much the community trusts a user [11]. Users can gain reputation by various means such as voting on others' answers, asking questions and contributing answers. The more reputation they have, the more privileges they unlock – such as the ability to edit other users' answers. Similarly, users can lose reputation for having a negative impact on the site, or when other users downvote their contributions [11].

Users also gain badges for achieving various different goals – these work in a similar way to a video game, where specific tasks repeated a certain amount of time warrant a badge: for example, if a user visits StackOverflow consecutively for 100 days they will receive the "Fantatic" gold badge. There are many different badges available, all of which are split into three categories representing their difficulty: gold, silver, and bronze. The ultimate aim of badges is to encourage users to have a positive impact on the site.

These mechanisms are how StackOverflow self-moderates its community while rewarding the top users.

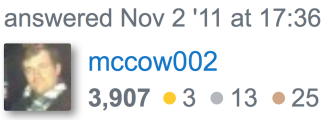

Figure 2.10: Screenshot of the metadata shown next to an answer on StackOverflow.com. 3,907 is the "reputation" of the user, 3 is the number of gold badges, 13 is the number of silver badges, and 25 is the number of bronze badges

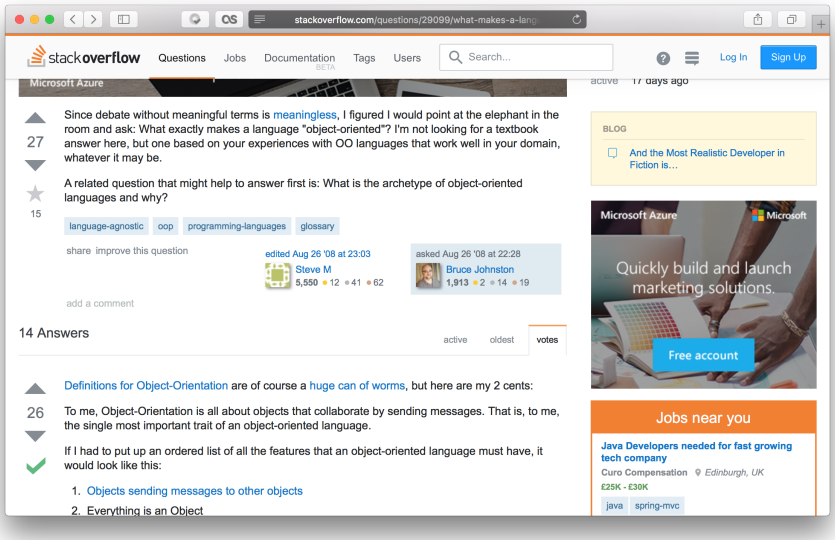

Figure 2.11: Screenshot from a StackOverflow.com question page

2.4.2 Glassdoor

Glassdoor is an online community of users which anonymously review and rate the companies they have worked for – at the time of writing it has over 600,000 companies reviewed on its platform. Glassdoor does have aspects of a crowd-sourcing site, as it offers the ability to add new companies onto the site, as well as provide information about salaries at a company.

Key features

- "Pros and cons" for a company are summarised at the top of a company page (Figure 2.12).
- Each review has a title, a body, a star rating, a positive/neutral/negative indicator for three specific topics, and a pros and cons box (Figure 2.14).
- At the top of a company page there is a star rating, alongside other metrics (Figure 2.13).
- Companies can respond to reviews.
- If a company does not already exist on the platform, users can add it.
- Users can add salaries, interview experiences, and company reviews.

Glassdoor has a lot of other information contributed by its users, such as salaries paid by a company. A large amount of this content is not visible until you have reviewed a company – this content-lock is used to incentivise users to contribute information to the platform. Glassdoor is monetised by helping employers find new recruits, as well as offering other employer-specific tools. The advantage of

this is that the site does not have any adverts and so the overall user interface of the site is clean and responsive.

Figure 2.12: Screenshot showing pros and cons for a company on Glassdoor.co.uk

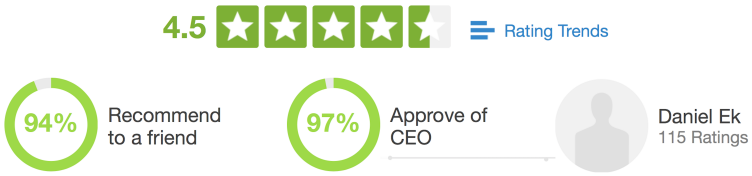

Figure 2.13: Screenshot showing a company's summary on Glassdoor.co.uk

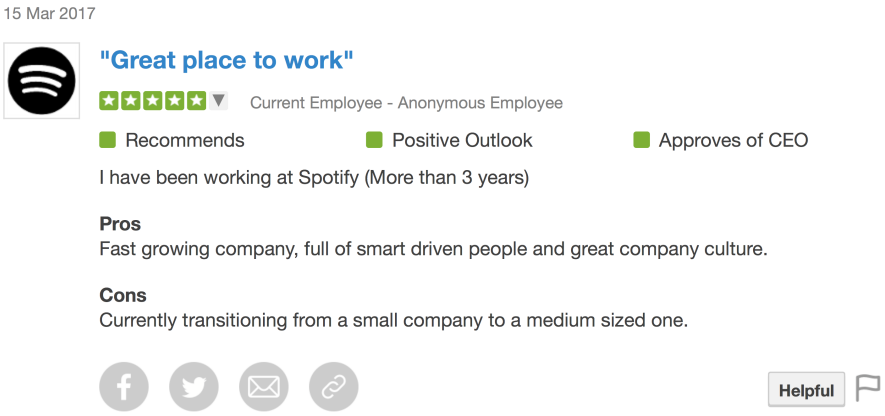

Figure 2.14: Screenshot showing an example review for acompany on Glassdoor.co.uk

2.4.3 TrustPilot

TrustPilot is an online platform that allows customers to review their experiences with websites. A screenshot of a company review page can be seen in Figure 2.15.

Key features

- Summary of ratings for a company is shown in a pie graph at the top of the page.

- A 5 star rating system is used and is displayed alongside a written review. An example review is shown in Figure 2.16.
- Companies can embed reviews on their website and can respond to reviews (for a fee).
- If a website does not exist on the site already a user can add it.

TrustPilot has a very clean, responsive site free of adverts. At the top of a company's page it is very clear to see how the reviews are broken down. The reviews themselves are quite simple: with a title, body, and a score out of 10. When users start writing a review, there is an explanation as to what "4 stars" means (as shown in Figure 2.17), which is a clever way of keeping star ratings consistent across users.

It could be argued that those who have had a bad experience are happy to share it with others – the incentive being revenge on the company. However, it is not clear what a user stands to gain by writing a positive review on TrustPilot, other than helping the company out. Unlike StackOverflow and Glassdoor, TrustPilot does not reward users for writing reviews. This leads to a question of authenticity, especially as the issue of fake reviews on TrustPilot has become the interest of the media and is something the site continues to tackle [12].

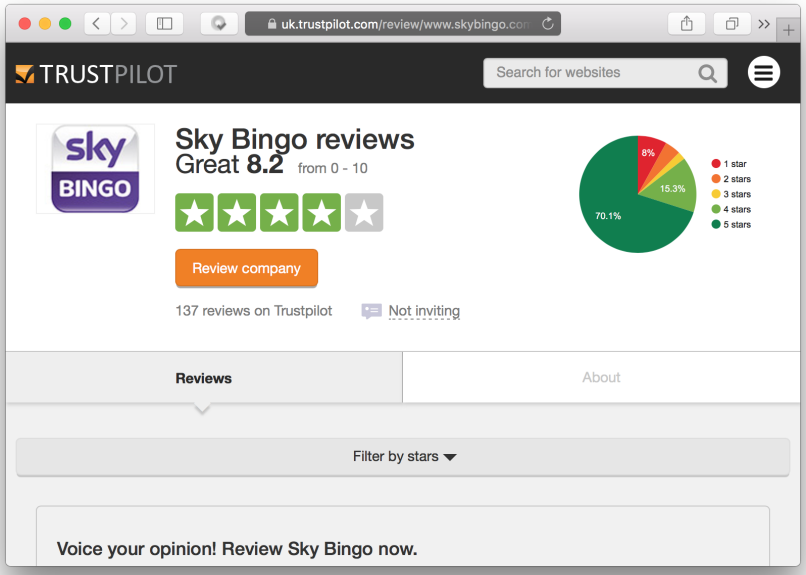

Figure 2.15: Screenshot showing a summary of Sky Bingo on TrustPilot.com

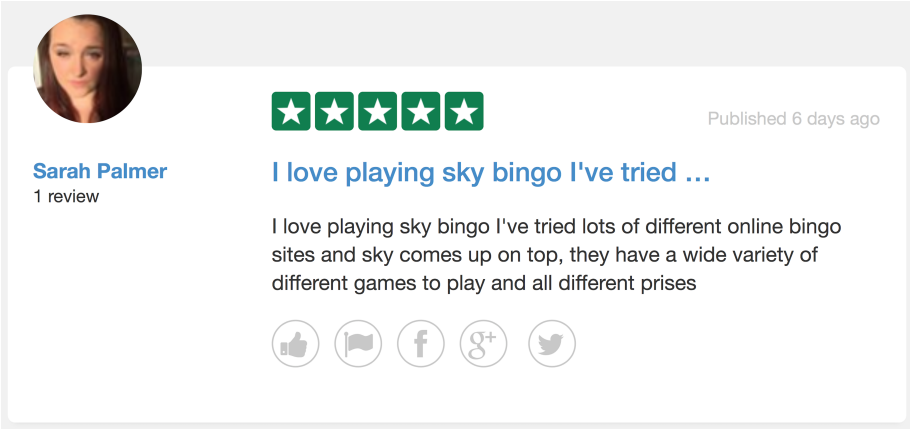

Figure 2.16: Screenshot showing an example review for Sky Bingo on TrustPilot.com

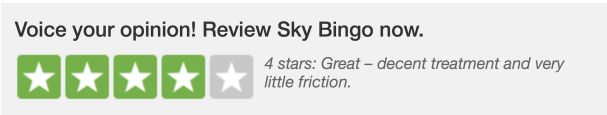

Figure 2.17: Screenshot showing a description of what 4 stars represents on TrustPilot.com

2.4.4 Others

- **Amazon** is an online retailer. It invites users to leave reviews of products to help other users make informed decisions about their purchases. It also uses a 5 star rating system alongside a written review. Sellers can respond to reviews if they wish. Although anyone can leave a review on any product (even if they did not purchase it through Amazon), reviews written by users who have purchased the product directly via Amazon are highlighted. This is a mechanism for discouraging fake reviews - an issue Amazon has been tackling as recent as 2016 [13].

- **Apple App Store** is the online marketplace for applications for Apple devices, such as iPhones. Users can only review applications they have downloaded. The review system is a 5 star rating system alongside a written review. Previously, developers could not respond to reviews. However, in January 2017 Apple announced they would soon be allowing this [14].

- **Google Play Store** is an online marketplace for applications for Android devices. It operates in the same way as Apple's marketplace. It has always allowed developers to respond to reviews submitted by users.

2.5 Summary

After analysing the various different platforms the following key points were established:

- There are very few *consumer* software review specific websites moderated by a community of users. AlternativeTo is the closest to this, however the reviews on its site are more of a side-product, rather than its sole focus.
- Most websites give a star rating for a piece of software.
- Current sites are cluttered with information and adverts.
- TrustRadius's "tr score" is a way of summarising information in multiple reviews, without losing key information such as the author's authenticity.
- AlternativeTo is a good example of how to crowdsource data about software.
- All sites have a form of rating in the form of stars, likes, or TrustRadius's own scoring system.
- Most review platforms allow the the subject being reviewed, whether it is a company, app, or website, to respond to reviews.
- All websites had a clear summary at the top of the page.
- Rewarding users with extra access to the site (in the case of StackOverflow and Glassdoor) is a good way of incentivising users to contribute good content.
- StackOverflow was an excellent example of how a community can moderate itself effectively.

Chapter 3

Development and Initial Design

This chapter looks at the technologies and practices chosen for the development of Meagle, and the design decisions behind these.

3.1 Stakeholders

"Stakeholders" is a term given to any person, or group, that may affect or may be affected by the piece of software being developed [15]. They should be considered at each stage of the software development cycle. For Meagle, the stakeholders were:

- users writing reviews on the site,
- companies that build the software being reviewed,
- those developing Meagle, and
- those maintaining Meagle.

The first three stakeholders are fairly straightforward. The fourth, in the case of this project, is the open-source community. As the platform was being developed as an extensible open-source project, those who might develop it in the future needed to be considered: by means of clear documentation and code.

3.2 Non-functional requirements

Non-functional requirements describe how a system should perform, rather than how it should behave. For Meagle, the following requirements were decided on:

- **Compatibility**. The site should work the same across all browsers and devices.

- **Security**. The security practices used by the platform should be industry standard.
- **Usability**. The platform needs to have an easy-to-use user interface.
- **Extensibility**. The platform should be easily extended without having to re-write large parts of the code.
- **Testable**. The platform should be written in a way that is easily testable.
- **Open-source**. The platform should been open-source and have clear documentation for future developers.
- **Performance**. The site should have a fast response time.

These requirements were considered throughout the development of Meagle.

3.3 Development methodology

It was decided that the development of Meagle would follow agile methodology practices. Agile is an umbrella term for a list of processes that result in a rapid cycle of requirements planning, engineering, testing, releasing, and user feedback. This opposes more traditional approaches, such as the waterfall model, which would usually have a much longer development cycle and fewer releases [16]. Traditional software projects tend to be over-budget, late, and defective, whereas agile projects are the opposite [17]. Agile methodology is used by many large websites such as Facebook [18].

Agile development is not appropriate for all projects. In some environments, such as safety-critical environments like aeroplane software, it is not appropriate to rapidly release software that has not been fully tested. Nor is it necessary to have continuous updates. However, as Meagle is not a safety critical platform it was decided that a rapid release process would be fine to use: any bugs that are released accidentally, through the faster release process, are unlikely to have any major real-world issues.

3.3.1 User stories

A practice used within agile development is the concept of "user stories". An example of a user story might be *"I want to be able to login and see the reviews I've added"*: this is essentially a task, for an engineer, to implement a user profile. The user story should be small enough to be designed, coded and tested within a development cycle. It is also written from the perspective of the person who would benefit from the feature [15].

3.3.2 Sprint planning

Another agile idea is the concept of a "sprint" - this is a set period of development which includes all parts of the development process: planning, building, testing, releasing, and receiving user feedback. It is generally quite short – 2 for 4 weeks, for example [15].

The development of Meagle was split into 4 sprints. At the beginning of each sprint user stories were written, which were then split into tasks and added to Github issues[1]. These were linked to the public GitHub repository[2] and so they were all open-source and publicly visible. This gave the project transparency as anyone could see the project's current status and what future features were planned. It also allowed users to report bugs easily.

3.3.3 Continuous integration

A key principle of agile methodology is the concept of continuous integration. Continuous integration (also known as continuous deployment) is the process of continually "integrating" new code into a platform. That is, continually building and releasing new code [17]. This practice is used by many large organisations such as Facebook [18].

3.4 Making design decisions

When building a large project that needs to utilise multiple existing tools and frameworks it is important to make informed decisions about which ones to use. For example, in 2016 Facebook discontinued a very popular mobile-backend-as-a-service platform, Parse, which caused much upset to the 500,000 applications that relied on it [20] [21]. Developers that had chosen to build their business around the platform had to find or build an alternative.

Throughout this project the following questions were asked when making a decision regarding which tools or frameworks to use:

- **What is the scope the framework?** How useful is the framework? What are the overheads of using it?
- **Who uses the framework?** How many people are using this framework? Are any large companies using it?
- **How active is the development of the framework?** If it has not been developed recently it could be reaching the end of its lifecycle, which could prove problematic in the future.

[1] GitHub issues is a tool to keep track of tasks, enhancements and bugs within a project [19].

[2] GitHub.com is a website where developers can share and store their code. A repository is simply a collection of code for a project.

- **How abstract/coupled is it?** Can this framework easily be replaced with another framework at a later stage?
- **What is the licence?** If it is not open-source is there any guarantee the developer will not discontinue it?

3.5 Evaluation techniques

Throughout the development process a group of users from a mixed background of technical ability were asked for their feedback regarding the site's design, usability and usefulness. Their feedback was then used to generate the user stories for the next sprint. After the final sprint a survey was sent to an extended group of people asking them to perform certain tasks on the site and feedback on the same criteria.

The sites performance was analysed technically using online tools that look at page size, page speed, and mobile optimisation.

3.6 Site framework

The decision was made to use PHP – an open-source server-side scripting language that can be deployed on almost any operating system and platform, free of charge. It is used by some of the largest websites in the world including Facebook [22] and Wikipedia [23]. It has one of the largest communities of developers in the world using it [24] and it is also actively being developed itself [25] with a yearly release cycle [26].

At the time of writing, the most popular PHP framework on GitHub is Laravel [27]. It is an open-source framework with support for tasks that feature in most applications such as authentication, routing, database interaction, while having an expressive, easy-to-read syntax. Laravel is also regularly developed: it has a minor release every six months, with a patch release cycle as often as every week. It also has support for PHP 7 – the latest version of PHP – allowing utilisation of the latest features and security fixes [28]. When developing Meagle, PHP 7 was used.

Some of the key Laravel features and concepts that are used throughout the development are outlined here. Other concepts, such as events and jobs (Section 4.3.8) are explained when they are first used in the development timeline.

Eloquent models

Laravel has a built-in ORM – which stands for object-relation mapping. This is a programming technique for converting data between incompatible data types [29].

This allows interaction with PHP objects with the persistent store (such as a database) abstracted away from the developer, allowing the persistent store implementation to be changed easily without having to re-write the entire application. Each database table has a corresponding "Eloquent model" [30]. These are essentially classes that extend a general "Eloquent" class that interacts with the database – allowing models to be inserted, modified and deleted from the database. An example model is shown in Figure 3.1.

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

Figure 3.1: Example of an Eloquent model class as given in the Laravel documentation

Migrations

Laravel utilises the concept of "migrations". These are files that describe any database schema changes, and how to reverse them [30]. The changes expressed in these files can then be executed using the command `php artisan migrate`: any migration files that have not been executed before are performed, updating the database schema to match that of the developer. Essentially this avoids direct interaction with the database. Rather, a new migration file is made that describes the changes a developer wants to make and then the `php artisan migrate` command is used to update the database. This also means no raw SQL has to be written – abstracting the persistent data store away from the developer. This means that if in the future the developer wishes to use another type of persistent data store, no code has to be re-written.

Accessor attributes

These are dynamic properties for an Eloquent model that are not necessarily stored in the database. They are written as additional methods within an Eloquent model, with the method name `get<attribute-name>Attribute()`, which is then called using `$model-><attribute-name>`. Figure 3.2 shows an example

of this used in a `User` model. When `$user->firstName` is called, the first name stored in the database is passed into this method, manipulated using the function `ucfirst`, and then returned.

This functionality is useful for adding new, dynamic, properties to Eloquent models and is used throughout the project.

```
public function getFirstNameAttribute($value)
{
    return ucfirst($value);
}
```

Figure 3.2: Example code for a Laravel accessor attribute

Routing

Routing is the process of taking a user's HTTP request and "routing" it to the part of the application that contains the logic for the requested page.

Laravel has a built-in routing system – an example of a route definition is shown in Figure 3.3. This would take a URL in the form of `meagle.org/user/5` and display `User 5` to the end user – it takes the `id` parameter from the URL request and passes it into the application as a usable variable.

```
Route::get('user/{id}', function ($id) {
    return 'User '.$id;
});
```

Figure 3.3: Example code for a Laravel route definition

3.7 Service providers

To develop and host a platform like Meagle required the use of third parties. Deciding which companies to use required the considerations outlined in Section 3.4.

3.7.1 DNS

Cloudflare is a provider of various internet security services and is used by thousands of companies including Uber [31]. It operates at a DNS[3] level, routing traffic via Cloudflare's servers before hitting the destination servers. This allows

[3]DNS stands for Domain Name System and is used as a way of translating website addresses into IP addresses of the servers that host them [32].

it to provide protection against distributed-denial-of-service attacks[4], by routing malicious traffic elsewhere. A diagram of this process is shown in Figure 3.4.

Cloudflare also acts as a content delivery network, taking the load off of the final server by storing static files, such as images and stylesheets, on Cloudflare's servers instead.

The domain, meagle.org, was connected to the website host using Cloudflare's DNS hosting.

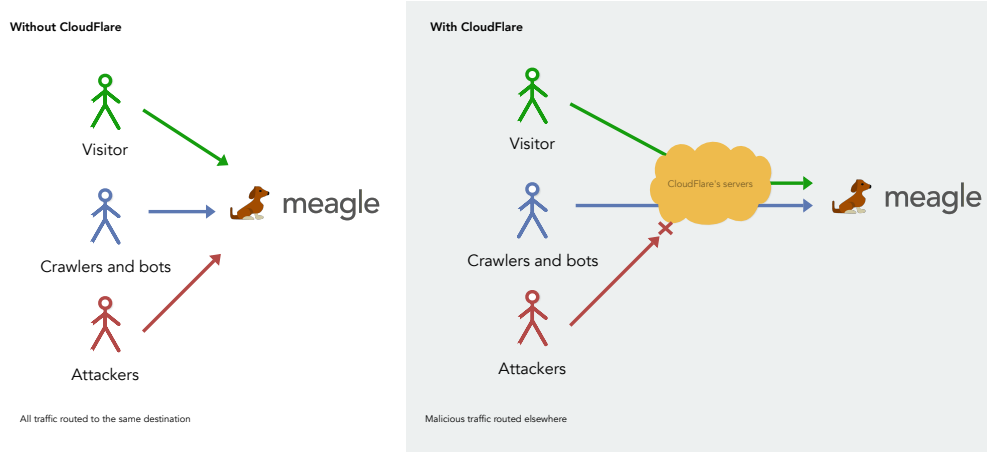

Figure 3.4: Diagram showing how Cloudflare works [33]

3.7.2 Code repository

One of the non-functional requirements was to open-source the entire codebase, allowing anyone to contribute to the platform. As the software was being built using git[5] a decision had to be made where to host the git repository[6].

There are two major git repository hosts currently:

- **GitHub**. This is the largest host of open-source projects in the world [34].
- **Bitbucket** is a commercial host of git repositories developed by Atlassian.

As GitHub is free, extremely popular, and has many integrations with other tools, it was chosen for Meagle.

3.7.3 Site

There are many ways to host a website. The following options were considered:

[4]A distributed-denial-of-service attack happens when a server is bombarded with traffic, overloading it.

[5]Git is a type of version control system for tracking software changes, allowing the ability for multiple developers to work simultaneously and combine their changes, as well as the ability to revert any change.

[6]A git repository contains the entire codebase as well as the history of the files.

- **Amazon AWS EC2** is a web service that provides secure, resizable compute capacity in the cloud. A developer can create a server with almost any capacity and resize it seamlessly depending on the demand. This allows almost infinite scalability and is used by numerous large companies such as Netflix [35]. It is also the largest hosting provider in the world [36].

- **Google Compute Engine (GCE)** is a similar service to Amazon's EC2. It is used by many large companies also, including Spotify [37].

- **Heroku** is a platform-as-a-service. Developers can push their code straight to Heroku, after choosing a "container". A container is simply an application stack (such as operating system, database and programing language) that the code runs on [38]. Heroku's underlying technology is hosted on Amazon's AWS [39].

Amazon EC2 and GCE take a low-level approach whereby the developer needs to manually login into the server and install their application stack, set firewall permissions, and maintain the entire server. While this is the cheapest, most flexible, option, it would have increased the complexity of the platform – requiring frequent server maintenance to ensure the latest security patches are installed and that the firewall conditions are correct – for a minor saving in cost. With Heroku [38] the required technology stack could still be used, with the ability of logging into the server if necessary, but Heroku would handle the more low-level configurations such as the firewall, as well as keeping the server up-to-date. Heroku also integrates with many other tools allowing for easy continuous integration.

3.7.4 Database

- **Amazon RDS**. A MySQL or Postgres database can be created and scaled using the Amazon AWS interface. After adding it the database needs to be configured to work with remote hosts.

- **Google Cloud SQL**. Another option that works in the same way as RDS.

- **Heroku PostgreSQL**. A Postgres database can be added on within the Heroku user area. A free database allows up to 10,000 rows. The database itself is hosted on an Amazon EC2 instance. There is no support for MySQL.

As Heroku's offering was more expensive than Amazon's, Amazon RDS was chosen. Unlike setting up a server, the database setup on Amazon required very little effort or maintenance and so there was little advantage of choosing one over than the other, other than cost.

3.7.5 Continuous integration

- **CircleCI** is a commercial continuous integration tool used by Spotify, Google and many other companies [40]. It is completely free for open-source projects.
- **TravisCI** is another commercial continuous integration tool that works in a similar way to CircleCI.
- **Jenkins** is an open-source tool used by several big companies including Facebook [41].

Although Jenkins is open-source, it requires self-hosting, which requires extra maintenance and costs. CircleCI, although commercial, is free for open-source projects and is used by many large companies, indicating (but not guaranteeing) a long lifespan. It was decided that its ease-of-use, clean documentation, and integration with GitHub outweighed the disadvantages of using it.

3.8 Security

One of the non-functional requirements was to use industry-standard security techniques.

3.8.1 Forced SSL

A man-in-the-middle attack is a type of attack that can take place between a user and a server: the attacker can intercept data being sent between the two parties without either knowing. This of particular concern when sending confidential information such as credit card information or passwords. This can be resolved with the use of the SSL protocol, which stands for Secure Sockets Layer. When an SSL certificate is installed on a server and enabled, data sent between the user and the server is encrypted, making it useless for anyone who manages to intercept the communication [42].

Although Meagle is not necessarily processing anything confidential, SSL on as a default is becoming increasingly common – this is the practice where websites force all pages to load through SSL, meaning there is never any unencrypted communication. Google recently announced that sites not following this practice would be negatively weighted in their search algorithm [43].

Cloudflare, the chosen DNS provider of Meagle, offers a one-click-SSL tool which simplifies the process of setting up an SSL. An additional feature offered is the ability to "force" the SSL so that *only* encrypted connections to the site are allowed [44] – something that was enabled from the first release of Meagle.

3.9 User interface

A large part of the project involved user interface (UI) design. This included designing how the navigation of the site should work, how users would find and contribute content, how content was displayed, and finally the overall look of the site.

UI design also considers how a site operates on different devices. As compatibility was one of the non-functional requirements, it was important that the site worked consistently across platforms without duplicated code or loss of functionality; mobile-optimised sites are also boosted in Google's search results over ones that are not [45].

Responsive design is a term given to sites that share a single codebase and stylesheet yet the website still displays optimally on all devices, regardless of screen size. The trend started into 2011 when first coined by Ethan Marcotte [46]. To do this, elements such as buttons, columns, and text fields move and resize for the device they are being viewed on. This is contrast to a traditional development practice which would involve creating several versions of the site: generally one for mobile and one for desktop. The disadvantage of this approach is that more code needs to be written and maintained, inevitably leading to inconsistencies when new features or updates are released.

3.9.1 Frontend framework

Bootstrap is an open-source responsive CSS framework developed by Twitter. Bootstrap is highly customisable, already responsive, and contains hundreds of elements such as progress bars, alerts and buttons. It is also the most popular responsive frontend framework on the web [47]. At the time of writing it is the most tagged CSS framework on StackOverflow - indicating the size of the community. Another advantage of Bootstrap is that it fully supports all modern browsers, including very old browser versions such as Internet Explorer 8 which was released in 2009 [48]. These reasons were why Bootstrap was chosen – it allowed focus on the design of the site, without having to worry about more tedious issues such as cross-browser compatibility or device support.

3.9.2 CSS precompiler

LESS is an open-source CSS[7] pre-processor, meaning that it extends the CSS language adding additional features which streamline the development of stylesheets.

Some notable examples of these features include the ability to define variables which can be re-used throughout the stylesheet, particularly useful for colour schemes. LESS also offers "mixins" – functions which can take inputs and produce

[7] CSS stylesheets are used to style websites and define how a website should look.

an output. An example use case of this is on the styling of a button. A mixin could automatically generate a darker colour, given the button colour, for the button border. If the button colour changes, the border colour will also change without any input from the developer [49].

Bootstrap's framework was written in LESS and the raw LESS files are available to download. This allows developers to customise all the variables used throughout Bootstrap, allowing for complete customisation. This was how the framework was customised.

3.9.3 Automation

Gulp is an automation tool to improve the workflow of frontend development. It can perform tasks such as Javascript minification – the process of shrinking Javascript files into smaller ones, as well as compiling LESS files into CSS [50].

Laravel Elixir is a tool which provides an API for defining basic Gulp tasks. Gulp tasks are written in a "gulpfile" – an example gulpfile is shown in Figure 3.5.

```
const elixir = require('laravel-elixir');

require('laravel-elixir-vue-2');

elixir(function(mix) {
    mix.less('app.less')
        .scripts('app.js')
        .version(['css/app.css', 'js/app.js'])
});

```

Figure 3.5: Example Gulp automation configuration file using Laravel Elixir's API

In this example, every time the `gulp default` command is executed in the source directory, three tasks are executed:

- `mix.less('app.less')` compiles the **app.less** file into CSS. This involves combining many less files together into one CSS file.
- `.scripts('app.js')` can combine multiple javascript files together – in this case just one – into one minified[8] javascript file.
- `.version(['css/app.css', 'js/app.js'])` checks to see if any changes have been made to the stylesheets or javascript files – if so it versions the file. This involves renaming the file slightly so that any browsers that have cached the previous file will be forced to request the new file, ensuring updates are seen by the user.

[8] A minified file is a compressed version of the original file, reducing the file size. This is done by shortening variable names, removing whitespace, and various other techniques.

This automation allows a developer to focus on the design and implementation of a site rather than menial tasks. It also ensures a consistent process if multiple developers are working on the same codebase. This process was used when developing the frontend of Meagle.

3.10 Dependency management

As part of the project many existing packages were used, such as Laravel. These are known as dependencies. A dependency manager is a tool that keeps track of the various libraries and packages used within a project, and the versions used. Rather than make a copy of the library into the git repository, a single dependency file keeps a list of the libraries required and the versions used. A developer can then install these libraries by executing a simple command. This separates the files developed externally from the files developed internally. If a dependency needs to be removed, it only requires a single line removed from the dependency file – rather than deleting potentially hundreds of files, which could lead to the wrong files being deleted. Also, any bug fixes released by the original developer can easily be installed using the dependency manager.

A dependency may have dependencies itself; for example, Laravel requires over 20 other dependencies [51]. These dependencies are not listed within Meagle's dependency file, however Composer will go through every dependency in Meagle's dependency file and recursively look for their dependencies and install everything that is required. Dependency conflicts can occur during this process; for example, if two dependencies require a different version of another dependency. However, it is easier to resolve using a dependency manager than if done manually.

There were three dependency managers used while developing Meagle:

- **Node Package Manager (NPM)**. This managed Node dependencies, such as the tool Gulp (Section 3.9.3).
- **Composer**. This managed PHP dependencies, such as Laravel.
- **Bower**. This managed Javascript and CSS dependencies, such as Bootstrap (Section 3.9.1).

3.11 Releasing

The remote git repository on GitHub was connected to the continuous integration tool CircleCI. This means that any time code is pushed to GitHub it triggers an alert at CircleCI to start a "build": as part of this build process, dependencies defined in `composer`, `bower` and `npm` files are installed and tests are run.

Once all dependencies are installed and tests pass, CircleCI starts a release on Heroku. The build files are then copied to Heroku's server and database migra-

tions run, updating the sites database schema to match the schema defined in the migration files. As the files are built by CircleCI, the release process at Heroku only takes several seconds, during which the site is taken temporarily offline.

3.11.1 Release process

This is an ordered list of the commands and tasks that would happen when a release is triggered:

1. **`git push origin`**. This is where the developer pushes the code from their local branch to GitHub.
2. **Build is triggered on CircleCI**. GitHub sends an alert to CircleCI, notifying it of the new code. This tells CircleCI to start building.
3. **`git pull`**. CircleCI pulls the new code from GitHub.
4. **`npm install`**. Node dependencies are installed as defined in `package.json` file. These include Gulp and Bower.
5. **`composer install`**. PHP dependencies are installed as defined in `composer.json` file. This includes the Laravel framework itself.
6. **`bower install`**. Frontend dependencies are installed as defined in `bower.json` file,. This includes Bootstrap.
7. **`gulp --production`**. Automation tool Gulp compiles the LESS files into a minified CSS file. The `--production` tag tells Gulp to perform any tasks that are relevant only to the public release.
8. **`phpunit`**. Unit tests are executed from the `tests/` directory. These are run on a test database. If any test fails, the build process stops here.
9. **Release build to Heroku**. The build is then sent to Heroku.
10. **`php artisan optimize`**. This command removes any Laravel cache from previous releases.
11. **`php artisan down`**. The site is briefly taken into maintenance mode while the database is being altered.
12. **`php artisan migrate`**. Database migrations are run. Any issues here will result in an error and the site will remain in maintenance mode.
13. **`php artisan up`**. Following a successful migration, the site is brought back up.
14. **Release is live at Meagle.org**. The build is released to the public.

This allows Meagle to be rapidly updated as just one command is required to update the site, mirroring the local development environment, with any new dependencies or database changes being reflected. As deployments cannot happen

without tests passing, this is a fairly safe process. This is how Meagle was frequently updated throughout the timeline of this report.

3.11.2 Model-View-Controller pattern

Laravel is structured to follow the Model-View-Controller, MVC, pattern. This is a design pattern that splits the logic of a website. A model is an object that carries data – in Laravel this would be an Eloquent model. A view controls how data is shown to the user on a page. Finally, controllers controls the flow of data flow from a model to a view, keeping the view and model separate [52].

3.12 Code structure

Items written in *italics* are not on the Git repository.

- **resources/views/assets** This is where the source for Javascript and CSS files are stored (for example LESS files).
- **public/** Files in the assets folder are compiled and copied here, as well as dependencies from the *bower_components* folder, using Gulp (Section 3.9.3). This folder also includes static images.
- **app/** The class files for the models are found here (Section 3.6).
- **database/** Database migration files are stored here (Section 3.6).
- **tests/** Test files are stored here (Section 3.13).
- **routes/** Route files determine, given a request from a user, where in the code the request should be directed; how it should handle the request (Section 3.6).
- **resources/views** This is where the Blade views are stored (Section 4.1.4).
- *bower_components/* Where bower dependencies are installed.
- *node_components/* Where node dependencies are installed.
- *composer/* Where composer dependencies are installed.
- **bower.json** This is a JSON file that defines the Bower dependencies (Section 3.10).
- **composer.json** This is a JSON file that defines the Composer dependencies (section 3.10).
- **package.json** This is a JSON file that defines the Node dependencies (Section 3.10).
- **gulpfile.json** This is a JSON file that defines the automation steps that are used by Gulp (Section 3.9.3).

3.13 Testing

The site was tested in two ways: through automated testing and through user testing. User testing involved users simply using the site looking for bugs. Automated testing is the process of writing code that automatically tests the main code written. Laravel supports PHP's automated testing framework, PHPunit [53]. There are three types of tests that had to be considered: unit tests, integrations test and functional tests.

3.13.1 Unit tests

A unit test is a test of the smallest element of code that can be sensibly tested. It compares an actual value (returned by the piece of code) with an expected value. When writing unit tests it is important to isolate the function that is being tested to test *just* that function. In practice, this can be very difficult and involves decoupling code so that functions are short. One way to do so involves mocked objects – fake versions of the classes being tested [54]. PHP has an open-source mocking framework called Mockery [55] which was used when writing unit tests. If the function being tested depends on the result of another function, the dependency can be "mocked" out with a fixed return value. As the behaviour of the dependency is then effectively static, the developer can then isolate what is and what is not being tested.

3.13.2 Integration tests

Integration tests look at how the different functions interact with each other: testing the "integration" between functions [56]. Whereas a unit test would mock out database interactions, an integration test would not. Its purpose is to test how multiple functions operate together.

Factories

Laravel has a tool called Laravel factories [57]. This allows an automated way of creating fake models with seemingly real data. These can then be interacted with as if they were real users. These factories can be used to generate as many models as needed, with either randomly generated values or specified values, as well as specified relations.

For example, a factory can be used to generate a user with 10 points and 4 reviews. A test could be written to see what happens when the user tries to vote on their own review. Integration tests operate on a test database and before each test is run, the database is reset to ensure that the previous tests do not interfere with the next tests. This adds some overhead to the tests and so they run slower

than unit tests. However, it is beneficial to do as it highlights issues that would not appear in unit tests such as a database schema not matching a model.

3.13.3 Functional tests

Functional tests check to see that the entire application performs as it should. This tests the entire flow a user would experience: for example, typing out in a text field and trying to post a review. It is essentially encompassing unit and integration tests with a form of browser testing. These are the slowest tests to run as they run using a browser engine and so it is best practice to minimise the number of these tests as they delay the build time. They are useful for testing common user interactions [56]. Laravel has a built-in framework for this kind of testing [58].

Chapter 4

Design and Releases

The project was split up into 4 separate sprints, each one month long. At the start of each sprint user stories were created – these were based on both the initial goals of the project, as well as user feedback from the previous sprint. These user stories also considered the initial stakeholders. During each sprint features were created and released continuously as part of the continuous integration practice. A user group was consulted at the end of each sprint to evaluate the success of the releases.

4.1 Sprint 0

The first sprint involved setting up and releasing a clean Laravel installation – essentially just building the code from Laravel's repository and releasing it, without making any changes. The start date of this sprint was November 30th 2016.

4.1.1 Design brief

As this sprint focussed on an initial setup, rather than end-user features, there was only one user story of *"I want to be able to go to Meagle.org and see the Laravel installation welcome page"*. After Laravel is installed, a welcome page is shown indicating everything has been setup correctly, and so the aim of this sprint was to reach the point where this was shown at Meagle.org.

4.1.2 Setting up Laravel

Initially Laravel was downloaded and installed with all of its necessary dependencies. These were added to a git repository and pushed to the remote git repository on GitHub.

4.1.3 Data store

Laravel offers support for 4 different data stores, all with subtly different features: MySQL, Postgres, SQL Lite and SQL Server. MySQL is currently the most widely used open-source relational database management system in the world [59]. As of March 2017 it currently has over 6 times as many questions on StackOverflow to the nearest competitor, Postgres. Although this is not the most important factor to consider, it did mean that should there be any issues or help required, there is a currently a larger community available for MySQL. These two factors lead to the decision to use MySQL.

4.1.4 Setting up the frontend

Setting up Bootstrap

The frontend dependency manager, Bower, was used to install Bootstrap. Using the command `bower install bootstrap --save`. This updated the "bower.json" file to reflect the new package.

Workflow automation and LESS

Bootstrap was installed with all of its original LESS files that make up the framework. When compiled together, they create a single CSS file. By splitting up the CSS into multiple component files that can be combined as and when needed gives the ability to produce multiple CSS files that can contain all or just parts of the original components. This results in reusable code and smaller CSS files.

Every time LESS files are edited, they need to be compiled into CSS. Gulp, the automation tool described in Section 3.9.3, can "watch" for changes in LESS files and run a certain task when they have been edited, and so Gulp was setup to compile the LESS files together - Bootstrap's and Meagle's own LESS files - into a CSS file, which was then minified and stored in our `public/css` directory.

Displaying content using Blade

Adhering to the Model-View-Controller principle, the display of data is the responsibility of *views*. In Laravel, these are written in its templating engine called Blade [30]. In a similar way to the LESS being broken down into subcomponents, the pages shown to the user can be broken down into reusable components. They should be stored in `resources/views` and can be written in Laravel's Blade syntax, as well as any regular HTML or PHP.

Figure 4.1 shows a snippet from the main navigation view - the search box view was included within this view. This allowed the search box code to be used

within other views and other parts of the application without having to rewrite or duplicate code.

```
<!-- Left Side Of Navbar -->
<ul class="nav navbar-nav">
    <li><a href="{{ route('home') }}">Home</a></li>
    <li><a href="{{ route('browse') }}">Browse software</a></li>
</ul>
<div class="nav navbar-nav hidden-xs">
    @include('segment.search_box')
```

Figure 4.1: Snippet from Blade view **navigation.blade.php** showing another view, `search_box`, being included

4.1.5 Releasing the site

As the GitHub repository was already connected to Heroku, as outlined in Section 3.11.1, to deploy the site with the new code and dependencies, all that was required was executing the `git push origin` command. The domain, Meagle.org, had been connected to Heroku using Cloudflare's DNS hosting (discussed in 3.7.1). This released a blank Laravel installation at Meagle.org – this is just a page that says "Welcome to Laravel".

4.2 Sprint 1

This sprint started on December 20th 2016.

4.2.1 Design brief

This sprint aimed to build a "minimal viable product" – an agile term to describe a product that encapsulates the minimum number of features required to meet its initial goals; so the fewest number of features required for Meagle to be a "crowdsourced software review platform" [15]. This was summarised with this sentence: "A responsive website with the ability to login, register, vote on other users reviews, and with some ability to contribute new data in the form of crowdsourcing". This lead to the following user stories:

- "I want to be able to create an account and login"
- "I want to be able to review a piece of software"
- "I want to be able to give feedback on reviews left by other users"
- "I want to be able to use the site on my desktop computer and mobile"
- "I want to be able to add new pieces of software that are not on the site"

4.2.2 Architecture

From these user stories, the models required, and the relationships between them, shown in Figure 4.2, was established.

The models that were planned were:

- `Review`. Reviews have one author and can have many votes attached to them.
- `Vote`. Individual votes are associated to one review and one author.
- `Software`. Pieces of software are linked to reviews through versions. Versions are a particular piece of software which users can review. Versions have authors, as they are created by users, and have votes too: these votes decide whether a version added is real or not.
- `SuggestedReleaseDate`. These are suggested release dates by users and are individually associated to versions of software.
- `SuggestedReleaseDateVote`. Individual votes are associated to one suggested release date and one author.
- `Version`. Associated with one piece of software, one author, can have many votes, and many suggested release dates.
- `VersionVote`. Individual votes are associated to one version of software and one author.
- `User`. A user can have many reviews, many votes, many versions, and many suggested release dates associated with them.

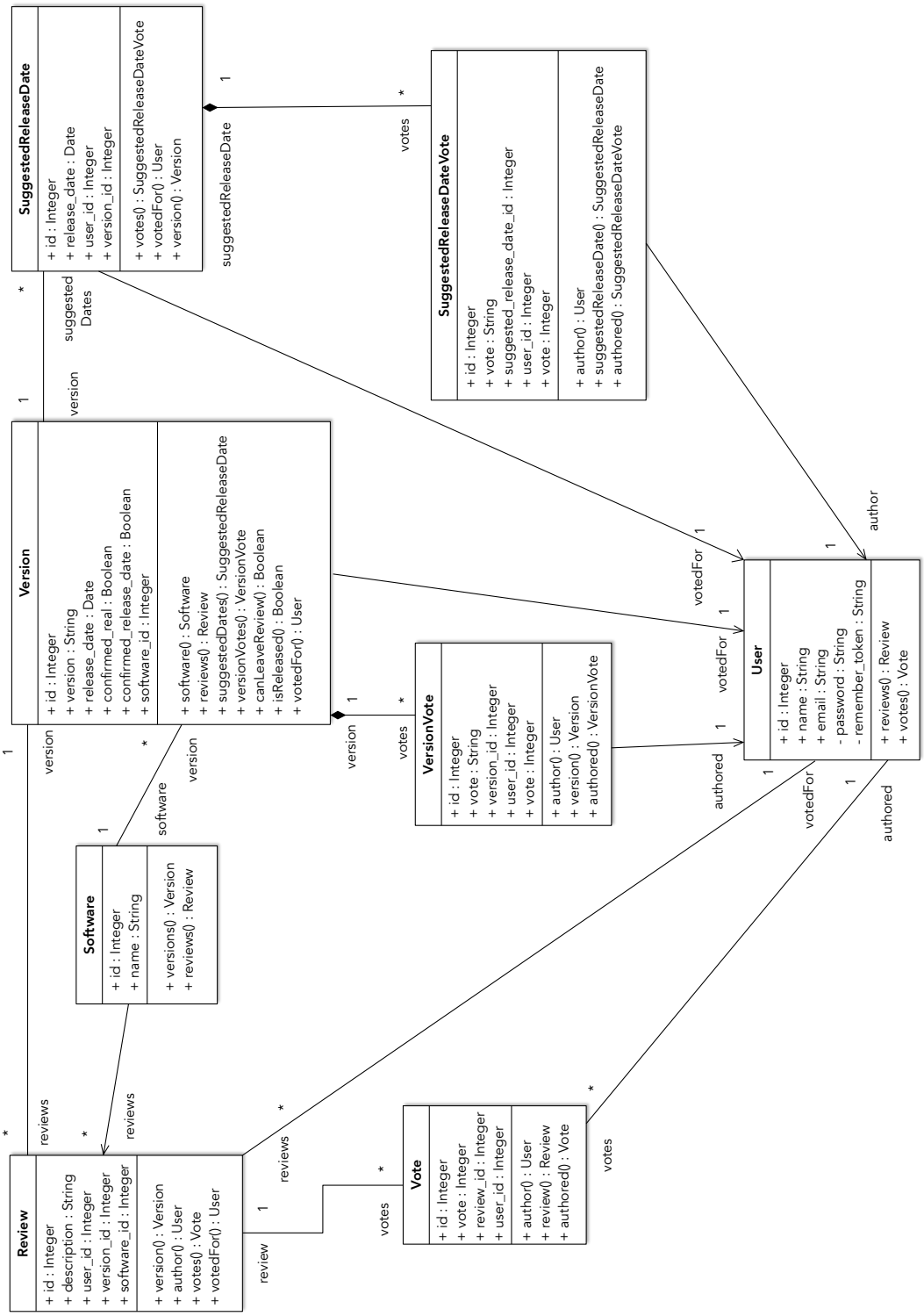

Figure 4.2: UML diagram of Eloquent classes in Sprint 1

Creating a model

After determining the models required, the models needed to be created. In Laravel, this is done using the following command: `php artisan make:model Model -m`. This creates a new class called `Model.php`, that extends the Eloquent parent class. It also creates a database migration file to create the necessary table in the database shown in Figure 4.4. This is discussed in the next section.

Figure 4.3 shows the class that was created after running the command. Laravel Eloquent classes have a variable called `$fillable` which specifies which attributes should be saved into the database.

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Model extends Model
{
    protected $fillable = [];
}
```

Figure 4.3: An example of an Eloquent model file after created using `php artisan make:model Model -m`

Migrations

Figure 4.4 shows an example migration file. The `up` method outlines the table to be created, and the attributes following Laravel's migration syntax [30]. The `down` method outlines what should happen if this migration file is reversed – what needs to happen to revert the database schema back to what is was before the update.

The database fields that are to be created should match the ones defined in the Eloquent class; the types of the fields also have to be specified. Default values, and foreign key relations, can also be specified.

The migration is then run using the `php artisan migrate` command, which generates and executes the necessary SQL query to execute the above and generate the table in the database. This process was done for all the models specified in the architecture.

Defining relations

Relations between classes can be expressed in the model classes [30]. These relations can be one-to-many, many-to-one, many-to-many. The types of relations

```php
<?php
class CreateQuestionsTable extends Migration
{
  /**
   * Run the migrations.
   *
   * @return void
   */
  public function up()
  {
      Schema::create('questions', function (Blueprint $table) {
          $table->increments('id');
          $table->string('question');
          $table->text('answer');
          $table->integer('user_id')->unsigned();
          $table->foreign('user_id')->references('id')->on('users');
          $table->boolean('confirmed_real')->default(0);
          $table->softDeletes();
          $table->timestamps();
      });
  }
  /**
   * Reverse the migrations.
   *
   * @return void
   */
  public function down()
  {
      Schema::dropIfExists('questions');
  }
}
```

Figure 4.4: An example a Laravel migration file

were planned in the architecture diagram in Figure 4.2. For example, the `Review` model has a `user_id` field which stores the id of the user which wrote the review; it is related to the `User` model: every question has one user (who owns it - the author). This relation is written in the `Review` class as a function shown in Figure 4.5.

Defining this relation gives the ability to quickly access a related object, given an object. For example, if variable `$review` is an instance of `Review`, the author object can be accessed using `$question->author` – this would be an instance of the `User` class. This makes the code readable and maintainable. It also makes it easier to switch between different implementations of an "author" if ever necessary as the only part that would change is the one `author()` method shown in Figure 4.5.

```
public function author()
{
    return $this->belongsTo('App\User', 'user_id');
}
```

Figure 4.5: Snippet from the `User` class showing the `author` relation

4.2.3 User registration / login

Laravel has a built-in authentication system known as Laravel auth [30]. Setting this up requires executing the command `php artisan make:auth`. This creates the necessary routes, models, controllers, and views. Laravel's authentication system includes advanced password encryption, CSRF-protected login forms, and token-based session management. Therefore it met all the security requirements for the application.

Encrypted passwords

The registration process requires users to sign up with an email address and password which is then stored in the database. As this password data is confidential, and if accidentally leaked, like in the TalkTalk breach, could lead to legal issues [60], it was important to encrypt it properly so that it would be unreadable if hacked.

Laravel uses a password hashing function called bcrypt to encrypt passwords [30]. The algorithm is based off of the Blowfish cipher algorithm [61]. Hashing is the process of applying a one-way function to a string to produce a constant sized output. As it is a one-way function, the process cannot be reversed: given a hashed password, no function can be applied to obtain the original string. An issue with hashing algorithms, such as MD5, is that two identical passwords will always be given the same hash. Although an MD5 hash cannot be reversed using a particular algorithm, it is possible to generate the corresponding MD5 hash of every combination of a string, up to a certain length – this is known as a brute-force approach, and can be used to compute a rainbow table: a precomputed table for reversing encrypted passwords [61].

Salting a password involves adding an additional random string to the beginning of the password before hashing it known as a "salt". This salt is then stored alongside the hash in the database. Two identical passwords will have different salts and therefore different computed hashes. This makes a database with salted hashes substantially harder to crack as a rainbow table for every salt would need to be computed to compromise the entire database [61]. Salted hashes also have the advantage of obscuring common passwords, as two identical passwords would appear differently in a database.

With bcrypt, this is taken a step further by repeatedly hashing the password using the same salt, making it extremely slow to compute a rainbow table. The

number of times it is repeated is specified by the work factor, and it is prepended to the salt in the final hashed password. The number of hashing iterations is $2^{\text{work factor}}$, meaning that an increase of just 1 in the work factor will result in double the computing time. The ability to easily increase this future proofs the algorithm: as computing power increases, the algorithm can be adjusted so that far more computing power is required to perform a brute-force attack [62].

The final hashed password is `$2y$<work-factor>$<salt><hash>`, where `$2y` is a way of identifying the type of algorithm that was used to generate this hash – to identify the hash as a bcrypt hash, to prevent any future ambiguity [61].

4.2.4 Responsive design

The user interface was built in Bootstrap, as discussed in Section 3.9.1. Before starting on writing the necessary (LESS) CSS and HTML for the site several mockups of the site were designed. This gave a clear idea of how the site should look at the end of the sprint. Figure 4.7 shows an example mockup that was created, designed in the application Sketch – a mockup application that requires no coding [63].

While looking at the mockup Bootstrap's framework was customised. This involved overriding some of Bootstrap's LESS variables – shown in Figure 4.6 – as well as creating several new Laravel Blade views. The final design can be seen in Figure 4.9. While slightly different, it closely resembles the original mockup.

```
/* Colors */
@background-color: white;
@brand-primary: #2A2A2A;
@headings-color: #2A2A2A;
@brand-danger: #BA6E6E;
@brand-success: #A1D490;
```

Figure 4.6: Snippet from LESS file **variables.less** showing Bootstrap variables, such as `@brand-primary` being overridden

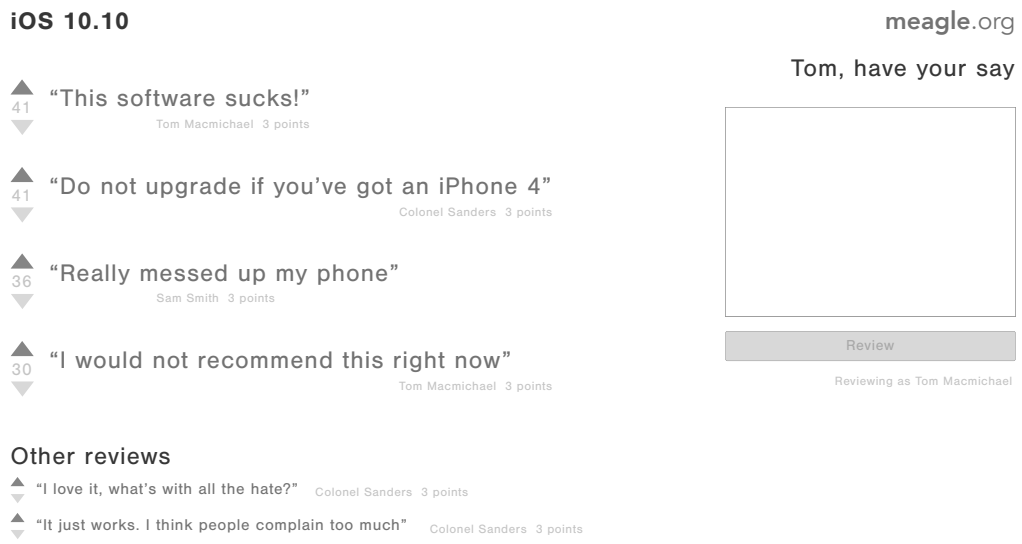

Figure 4.7: Mockup of the review page designed in the application Sketch

4.2.5 Basic crowdsourcing

In this sprint the ability to add versions of software was added to the site, however the pieces of software themselves still needed to be manually added. Users were also given the ability to contribute a release date of a particular version of software.

When users submitted a software version or a suggested release date, this would be immediately visible to other users. Users could then vote on whether they thought it was an accurate contribution (discussed further in Section 4.2.9). On a software version page, users were presented a table above reviews that would show whether the version of software they were looking at had been released and whether the data attached to it was confirmed as real. These checks looked at the `confirmed_real` property of a `Version`. However, this needed to be updated manually at a database level. This table can be seen in Figure 4.9.

Figure 4.8 shows an example page where a user is given the option to suggest the release date for a piece of software. The user could also vote on a release date suggested by other users.

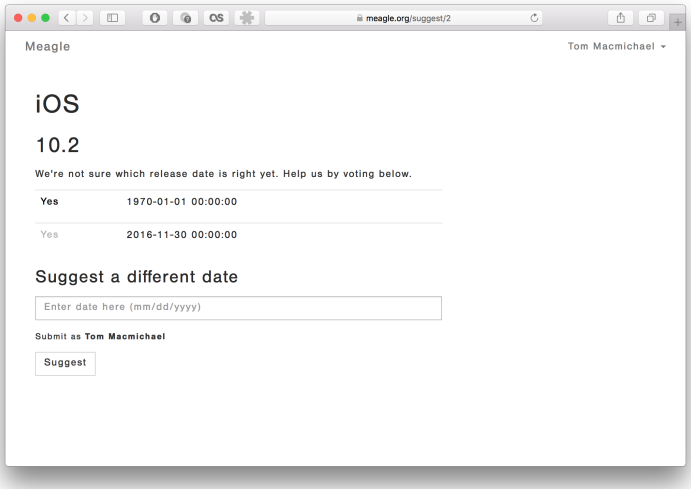

Figure 4.8: Screenshot of crowdsourcing of software release dates in Sprint 1

4.2.6 Writing reviews

In this sprint, users were also given the ability to write reviews on a given piece of software as shown in Figure 4.9. Reviews submitted by users would be instantly displayed on the page if the form was validated successfully (Section 4.2.7). The top reviews would be listed in a large font, with less popular reviews below as determined in the initial mockup.

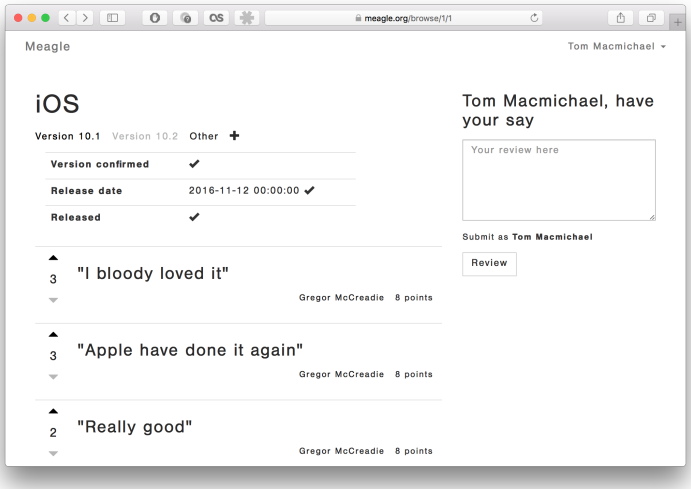

Figure 4.9: Software version pages with reviews in Sprint 1

4.2.7 Form validation

As the site now allowed user input (for voting, for writing reviews, and software release dates), form validation was required. Form validation is the process of

checking input submitted by a user and ensuring it is safe and correct before entering it into the database. With form validation, the request made with user input is routed to a specific validation class that checks all the required data is there and is okay.

Laravel's validation framework [30] was used to validate the request. Figure 4.10 gives an example set of rules for a particular form. In this example, if a `description` or `title` is not submitted, the request will fail and redirect back to the previous page. It also includes the requirement that the description has to be at least 80 characters long. These validation rules are extensive and can include database checks to ensure, for example, a new user has a unique email address that is not already registered.

After the validation rules pass, the data can then be safely inserted into the database.

```
public function rules()
{
    return [
        'description' => 'required|min:80',
        'title' => 'required|min:30|max:255',
    ];
}

```

Figure 4.10: An example of form validation rules in a form validation file

4.2.8 SQL injection

Rather than write raw SQL queries, Laravel has its own query builder – a syntax for building SQL queries indirectly [64]. This query builder can be used on a database table directly, or on an Eloquent model. An example of a query using this query builder is shown in Figure 4.11.

If data submitted by a user is used directly in an SQL query, there is a risk of SQL injection[1]. However, using Laravel's query builder, any data passed into the query builder (for example, user-submitted data), is "prepared" using PDO parameter binding: a process for preventing SQL injection [65].

Usually, when SQL queries are executed, the entire SQL query is interpreted as one string and executed. This means any user data that *includes SQL itself* has the potential to be executed as part of the query.

For example, an SQL query might take in a username and password submitted by the user:

[1]SQL injection is a technique to arbitrarily run SQL database queries on a vulnerable website, potentially causing loss of data or data breaches.

```
SELECT FROM users WHERE username = '$submitted_username' AND
    password = '$submitted_password'
```

If the `$submitted_username` was `'--` the SQL query would become:

```
SELECT FROM users WHERE username = '' --' AND password = 'any
    password'
```

The submitted username, `'--`, becomes part of the query – in SQL this syntax ignores anything past the dashes, meaning the password check is ignored. This is a simple example, but fundamentally the user can inject any SQL code they want to.

However, with PDO binding, the SQL query is sent separately from the user data. When executed, the user data is only considered as *data* and never executed as part of the query – even if the data includes SQL code. As Laravel's query builder does PDO binding for all data passed into it, SQL injection is not possible.

```
// $username contains a username submitted by a user
// Laravel's query builder builds a PDO-bound SQL query, using the
    user's input
$user = DB::table('users')->where('name', $username)->first();

// Print the name of the user extracted from the database table User
// This would print John, if John existed in the database
echo $user->name;

```

Figure 4.11: Example code showing how Laravel's query builder can be used

CSRF Protection

CSRF, Cross-Site Request Forgery, is a type of attack whereby users are forced into performing an unintended action on a site they are currently authenticated on.

This is best explained through an example scenario. In this example a website might accept data in the form of the following URL: `http://website.com/-?action=add_comment&message=_hello`, which adds a comment to the website with the message `hello`. The user is logged in at website.com and so the website posts this comment as the user which makes this request. An attacker wants this user to post a malicious comment on the site, as this user. The attacker simply crafts a URL which contains the data it wants the user to post and forces this user to visit this URL – they do so by sending an email to the user, which loads this URL as an image. The vulnerable user opens their email and their browser makes a request to this URL – it does not matter that this URL is not really an image; the browser will still load this URL. The user is logged in on this website,

and so the attacker's crafted message is posted from this user – and the user is completely unaware. This is how a CSRF attack happens.

To avoid this, CSRF tokens can be used. When a user visits a page, a CSRF token is generated by the website – this is a one-time randomly generated string that cannot be guessed – and stored in the user's session[2]. The user would then make a request, such as posting a comment on a website. When a genuine user posts a comment, the CSRF token is sent with it. To use the previous example, the request might look like: `http://website.com/?action=add_comment&message=-_hello&csrf_token=fsdfks8432423kfsdsdfsd1d`. The website then makes sure that the token stored in the user's session matches the one sent with the comment. If it does not, then the request fails and the comment is not posted. This prevents batch requests – as the token can only be used once. Also, as tokens are associated with the user's session, there is no way an attacker can generate a token for another user.

Laravel includes CSRF protection – any form that is submitted undergoes a CSRF token check, to verify the token submitted matches the one in the user's session. The code snippet in Figure 4.12 is used to place the user's CSRF token in a hidden field in a form. When the form is submitted, this CSRF token is submitted. [30].

```
<form method="POST" action="/profile">
    {{ csrf_field() }}
    ...
</form>

```

Figure 4.12: An example code snippet showing how a hidden CSRF input field is embedded in a form in Laravel

4.2.9 Basic points and voting

One of the user stories was the ability to give feedback on reviews left by other users. The voting user interface chosen was in the form of an upvote/downvote button, with the number of votes showing in the between. Reviews are then ordered by the number of upvotes they have received. The flow of interaction is shown in Figure 4.13.

This user interface was inspired by the design on StackOverflow for giving feedback on other users' answers. This design is advantageous as it allows quick feedback from users, while also bringing the most popular reviews to the top of the page, and the worst reviews to the bottom. By showing the number of votes a review has next to it, it is obvious to users how the reviews are ordered, as well as how many people agree with a review.

[2]Users that are logged in are associated with a unique "session". This unique session can be used to store information, such as the previous page visited by a certain user.

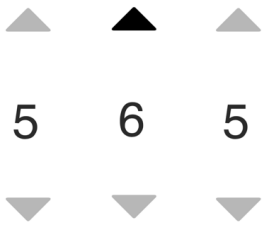

Figure 4.13: Left: before voting, middle: after pressing the upvote, right: after pressing the upvote again - deleting the vote

In this release, users were also given "points": where points were defined as the numerical sum of the votes towards them. For example, if there was a +1 vote for a user, and a -1 vote for a user, they would have 0 points.

Implementation

In this release, three ways to gain and lose points were introduced. As shown in the architecture diagram in Figure 4.2, there were three different types of votes:

- `Vote`. Every review could be voted on using the upvote and downvote arrows next to a review. When a user upvoted a review, the author would gain one point. Similarly if they downvoted a review, the author would lose one point. At a database level, upvotes were stored as a positive integer and downvotes were stored as a negative integer. An example of this is shown in Figure 4.9.
- `VersionVote`. Rather than use an upvote/downvote system, users were asked to vote whether a piece of software existed or not. The implementation at a database level was essentially the same: where "no" is a downvote and "yes" is an upvote. As with regular `Votes`, authors of these suggested versions would receive or lose points depending on the votes they received.
- `SuggestedReleaseDateVote`. The implementation of this was slightly different: users were presented with several release dates for a piece of software, that had been suggested by other users. Users then could vote on the one they thought was correct. Here, no negative votes could be recorded. Instead, the suggestion which met a certain threshold would just be confirmed as correct. This is shown in Figure 4.8. Again, authors of the suggestions would be given a point every time a user voted on their suggestion.

4.2.10 Other features

User profiles. These profiles displayed how many points a user had and any reviews they had written. These can be seen in Figure 4.14.

Verified users. Users registering from certain domains such as ed.ac.uk have a badge next to their reviews to increase the trustworthiness of it. The concept behind this was to highlight reviews from people who know the most.

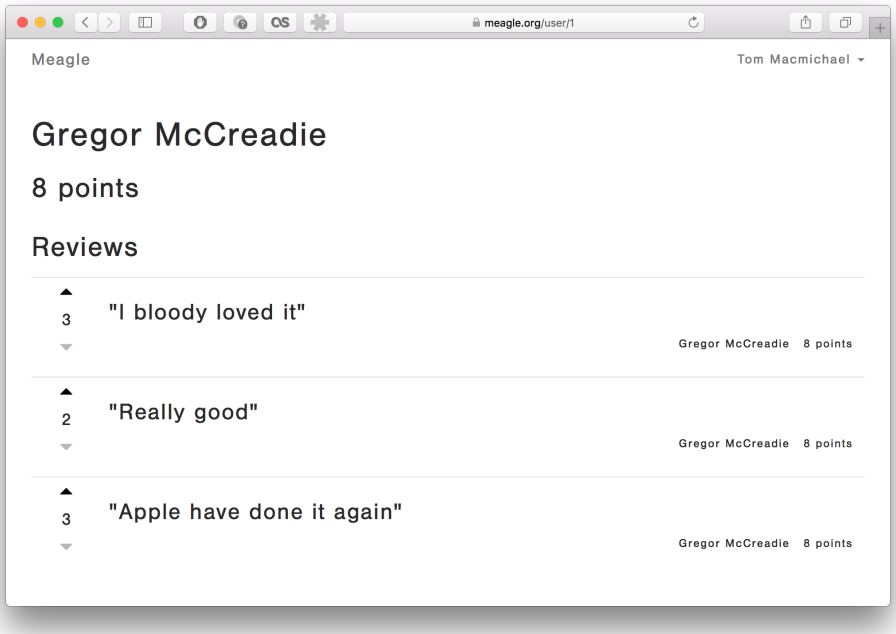

Figure 4.14: Profiles in Release 1

4.2.11 Evaluation and feedback

To evaluate this release a selection of random potential users with varying backgrounds were asked to use the site.

A common theme in the feedback was that having just one text field for the review was confusing as it was unclear as to what they were supposed to contribute; whether it should be a sentence or a paragraph of text. While users liked the simplicity of the design, it was still difficult to use – particularly hard to navigate between pages.

Users found it hard to quickly identify whether a piece of software had been reviewed positively or negatively as reading through the entire review was necessary.

4.3 Sprint 2

This sprint started on January 30th 2016.

4.3.1 Design brief

The aim of this release was to incorporate more crowdsourcing and automation into the site to allow it to be self-moderating. Previously, although users could vote on data, contributed data such as software versions needed to be manually confirmed or deleted – this process needed to be automated. From the feedback in the previous release, users wanted more flexibility than just a single text box when writing a review. The user stories for this release were:

- "I want to be able to find software on the site"
- "I want more information regarding a piece of software"
- "I want to be rewarded for writing good reviews"
- "I want to be able to navigate the site better"
- "I want to be able to write longer reviews"

4.3.2 Breadcrumb navigation

To satisfy the user story regarding navigation, a breadcrumb navigation was implemented. This is a fairly common concept and was something that is used on both CNET and TrustRadius, highlighted in the background research. The implementation of this on Meagle is shown in Figure 4.15. This used Bootstrap's breadcrumb class that has appropriate styling for breadcrumb navigation. The navigation was added to every page for consistency across the site.

Figure 4.15: Screenshot showing breadcrumb navigation

4.3.3 Tooltips

Bootstrap offers a tooltip plugin which allows information to appear in small popup boxes next to an element, when it is hovered over [66] as shown in Figure 4.20. This was utilised in many areas of the site to give helpful information explaining concepts. The advantage of a tooltip is that the page can appear uncluttered, yet when a user needs information it is easily accessible. This is something that TrustRadius use, as identified in the background research.

4.3.4 Tagging

A re-usable "tag" system was added allowing users to tag pieces of software with various negative and positive aspects – for example, "buggy". These tags

were added through the review form and then displayed next to each review as shown in Figure 4.16. The tags were colour-coded so that it was easier to determine whether a review was generally favourable to a piece of software or not. This implementation of tags was inspired by both Glassdoor and StackOverflow: StackOverflow allows questions to be tagged to help organise data on the site and to help users find related questions, Glassdoor asks users to give pros and cons for a company. Meagle's tagging system allows the "pros and cons" of a piece of software, but in the form of reusable tags. This feature was added to address user's concerns of wanting to quickly see if a piece of software was good or not.

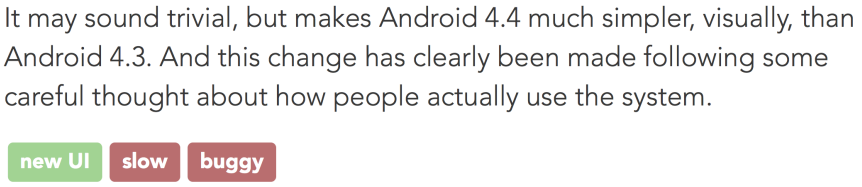

Figure 4.16: Screenshot showing tags below a review

4.3.5 Updated architecture

Previously, a new table and model were made for the different types of votes in the system: whether they were votes on reviews (`Votes`) or votes on suggested release dates (`SuggestedReleaseDatesVotes`). While this kept relations clean, and separated the data in the database, it introduced duplicated code and was not scalable should other, different, items need to be voted on in the future.

In this sprint, these different types of votes were combined into a single table called `Votes` and linked to their parent type (such as `Review`) using a polymorphic relation. This works by storing an extra piece of information for each entry in the database which stores the type of model being voted on.

Abstract Votable Objects

To reduce duplication of code a new abstract class called `Votable` was introduced. Any model that needed the ability to be voted on could simply extend this class and inherit all the methods required for voting. PHP also has the ability to be strictly typed: this is where a function can require a certain type (or a certain type of class) is passed to it. Using an abstract class also meant that other parts of the codebase could require their methods were passed a class of type `Votable` - which could be a review or it could be a question: allowing similar classes to be treated as if they were the same. Strictly typing classes has the advantage of ensuring that the correct data is passed to it as well as making the code more readable.

The updated structure is shown in Figure 4.17.

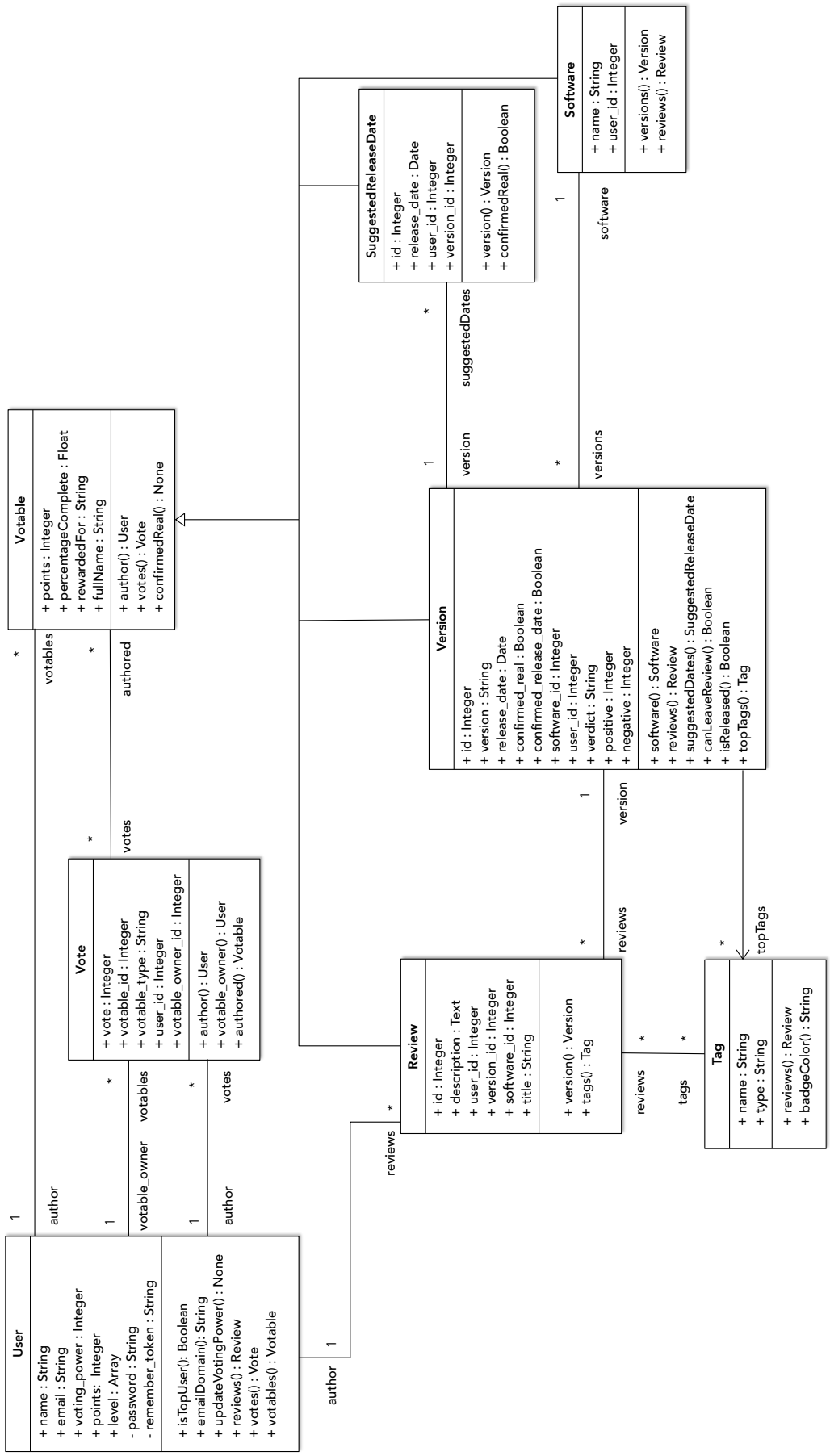

Figure 4.17: UML diagram of Eloquent classes after Sprint 2

Models extending the `Votable` class:

- `Review`
- `Software`
- `Version`
- `SuggestedReleaseDate`

Implementing these changes required the creation of several new classes and database migrations.

4.3.6 How points are calculated

In the previous sprint the concept of points was introduced, however the way the points were calculated was not necessarily clear to a user. To reduce confusion, the following pieces of information needed to be displayed to the user:

- How they achieved / lost points.
- When they achieved / lost points.
- How many points they achieved / lost.

As all points derive from the votes towards a user, all that was required to show how points were calculated was displaying these votes. This was implemented in the form of a table on the user profile page, shown in Figure 4.18. Previously, the `Votes` database table included the `id` of the person that voted, the `id` of the object being voted on, and the votable type (such as `Review` or `Version`). However, to populate this table a list of all the votes towards a user's contribution was needed. To easily implement this, a new column was added on the database table called `votable_owner_id`, which stored the `id` of the user which owned the object that was being voted on.

To populate the table, all votes that have a `votable_owner_id` equal to a specific user's `id` could be easily requested. This would return a list of `Vote` objects, with their relation (a `Votable`).

All `Votable` objects have the accessor attributes `getFullName` and `getRoute` - these can be overwritten by the extended classes, such as `Review` or `SuggestedReleaseDate`. This allows classes that extend the `Votable` class to share attributes but implement them differently. For example, `getRoute` on a `SuggestedReleaseDate` should link somewhere differently to `getRoute` on a `Review`. The table itself can then link to a specific route and display a specific name, without being concerned about the implementation of these for the different classes, as shown in Figure 4.19.

How my points were calculated

Points are gained either when a user makes a positive contribution (such as a good review) or when a user votes on something, which in turn the community then votes in the same direction on.

Points rewarded	Rewarded by	Rewarded on	Rewarded for	Time
10	Systems	For correctly voting on "I think this awful awful, awful"	Voting	1 month ago
1	Tom	I think this software update is fine but not great	Contribution	1 month ago
1	Simon	I think this software update is fine but not great	Contribution	1 month ago
1	Tom	Why has my review been deleted?	Contribution	1 month ago
1	Bertie Wilson	I think this software update is fine but not great	Contribution	1 month ago
1	Kayode Oduyemi	I think this software update is fine but not great	Contribution	1 month ago

Figure 4.18: Screenshot showing a table on a user's profile showing how their points were calculated

```
<a href="{{ vote->votable->route }}">
    {{ vote->votable->fullName }}
</a>

```

Figure 4.19: Snippet from **points_history.blade.php** showing how the view could be re-used for all `Votables`

4.3.7 Consensus boxes

This concept was designed specifically for Meagle and was not found in background research. The idea behind this was a way of making the crowdsourcing nature of the platform more transparent. "Consensus boxes" are boxes that display how close the site is to confirming a piece of data; that is, how close users are to reaching a consensus. If 10 users vote that a piece of software is real and 10 vote that it is not, then the consensus box will be empty: we are no closer to reaching a consensus than when no-one had voted. Figure 4.20 shows this.

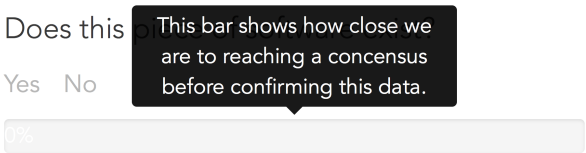

Figure 4.20: Screenshot showing an example consensus box for a piece of software

Reusing views

In the **views/segment** directory, Blade views that could be reused easily were placed. A **consensus_box.blade.php** file was created, shown in Figure 4.21, that takes a `Votable` object and displays the `percentageComplete` attribute; an attribute that was added to the `Votable` class, determining how close a verdict is to reaching a consensus.

```
<div class="progress-bar" role="progressbar"
aria-valuenow="{{ $votable->percentageComplete }}"
aria-valuemin="0" aria-valuemax="100"
style="width: {{ $votable->percentageComplete }}%">
    {{ $votable->percentageComplete }}
</div>

```

Figure 4.21: Snippet from a Blade view of a consensus box which can be used by many Votables

4.3.8 Self-moderating platform

To implement this feature, automatic removal of unwanted data points as well as automatic confirmation of correct data points was introduced.

Events

Laravel events allow the implementation of an observer pattern and decoupling of code. In the observer design pattern, "subjects" maintain a list of dependents, "observers", which are notified if there are any changes to a subject [67]. This was used to observe the votes taking place on Meagle, and take appropriate actions if necessary.

`UserVoted` is an event that is created whenever a user votes. After this happens, Laravel alerts any part of the application "listening" to this event: in this case, that is the `UserVotedListener` listener. This is a class that determines which users should be rewarded or punished after this vote. These listeners are registered in a separate part of the application, when it is first initialised.

Jobs

Jobs are pieces of code that are executed in a separate thread from the rest of the application at a separate time from when they were created, therefore they are not synchronous and do not block the page load speed. For example, if a user registered on a website, a developer might wish to send an email to this user. The developer could choose to call an email API and send the email which may

take several seconds to execute. As PHP is synchronous, the user viewing the page would have to wait for this to execute before the rest of the page would load. Implemented using a job, the developer can place the task to be executed at a different time on a different thread. It is a way of deferring tasks that do not immediately need to be executed.

When a job is created it is placed in a virtual queue for execution. Laravel offers several ways to implement this, with the simplest being an extra table in the database called `jobs` that keeps track of the jobs that need to be executed. A job can also have a specified priority so that less important tasks do not block time-critical tasks.

This system was utilised for the rewarding and punishing of users. When a user votes on the site, the `UserVoted` event is fired and the `UserVotedListener` is executed. The listener itself decides whether the object being voted on has reached a threshold for deletion or confirmation. This then executes the `confirmReal()` method on the object being voted on or delete it. Concrete classes, such as `Review`, all inherit this method: they can also extend it to perform different actions when they are confirmed as real, making the code not only reusable but extensible. It then creates a job to reward and punish users who voted correctly and incorrectly, where "correctly" is defined as "voting in the same direction as the majority". This rewarding and punishing is then executed at a later time.

The queue runner is run as a daemon on the server. A daemon is a computer program that runs as a background process [68]. It constantly checks for new additions to the queue and then executes the jobs that have been added. Generally the jobs are executed straight away and so there is minimal delay.

Rewarding and punishing

Rewards and punishments are given in the form of a `Vote` towards this user. The system creates a new `Vote` of +10 for good behaviour and -10 for negative behaviour. These votes then appear on a user's profile. The process from a user voting to a job being executed is shown in Figure 4.22. This concept of rewarding users who voted with the majority and punishing those that did not was a concept designed specifically for Meagle.

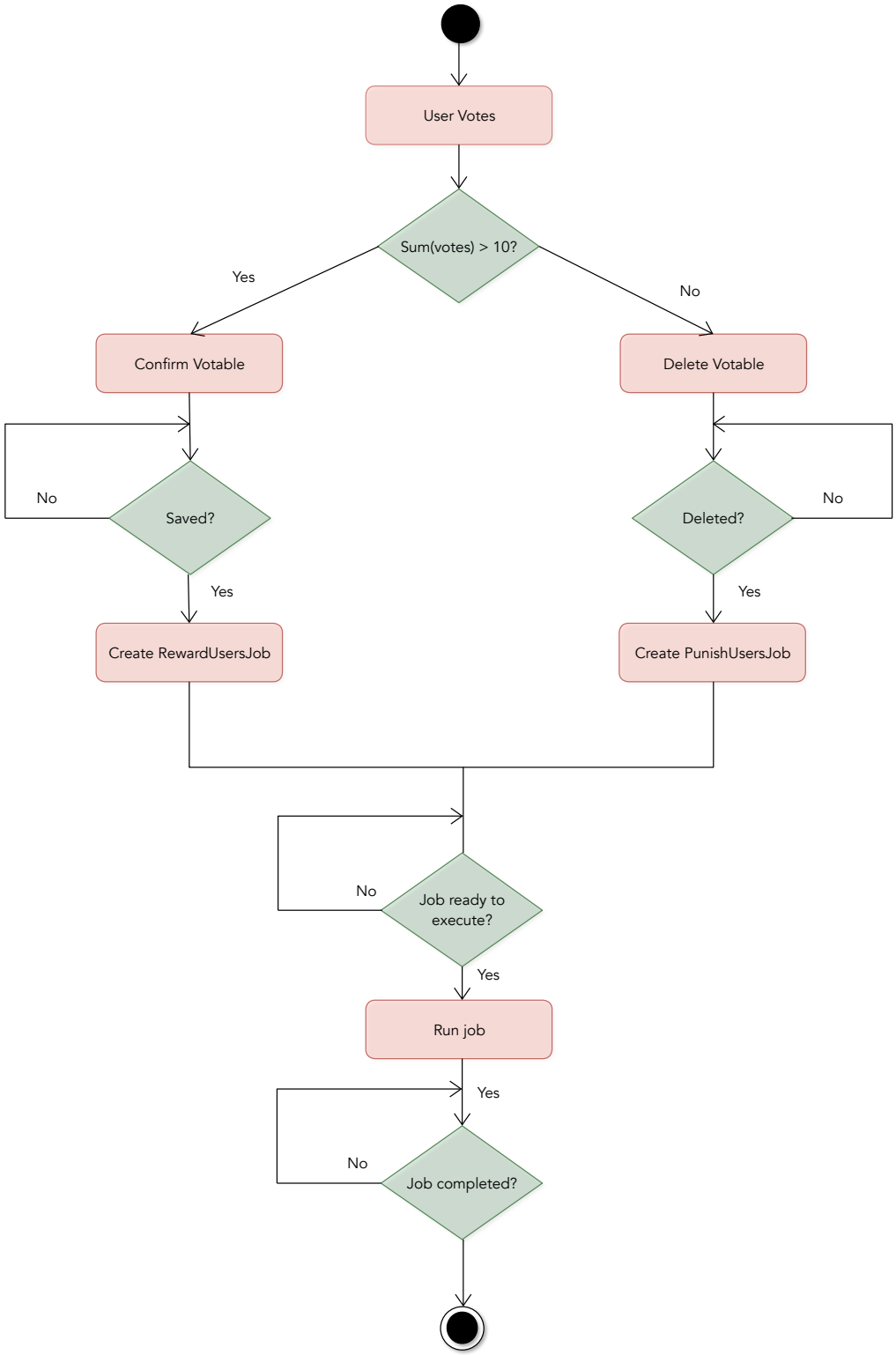

Figure 4.22: Activity diagram showing how users are rewarded and punished using jobs

4.3.9 New user interface

The site had large improvements in the user interface in this version. This included redesigning the site to make it look more professional. To do this, several mockups were designed as shown in Figure 4.25 and Figure 4.23, with the actual implementated version shown on the right of the respective images.

Elements such as error messages when form validation failed, were added. Previously, users would not know whether the form had saved successfully or not. The flow of user interaction with the website was also improved to reduce the number of clicks required to perform various tasks.

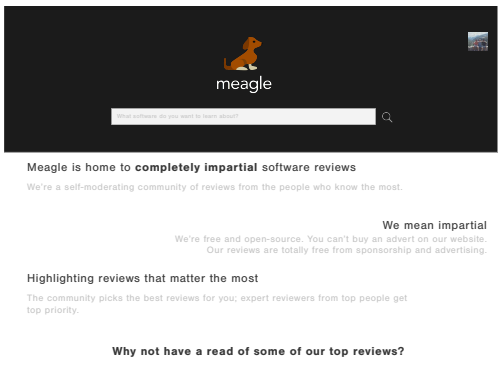

Figure 4.23: Mockup of the homepage in Sprint 2 designed using Sketch

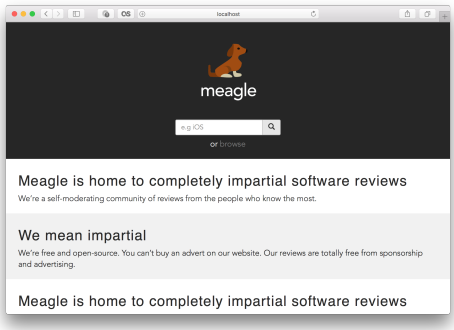

Figure 4.24: Actual homepage after Sprint 2

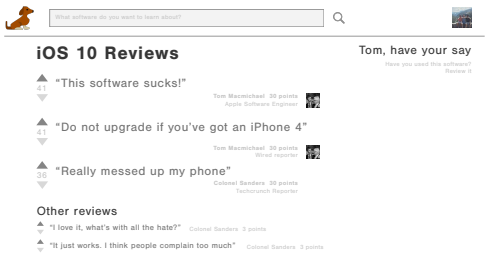

Figure 4.25: Mockup of the reviews page in Sprint 2 designed using Sketch

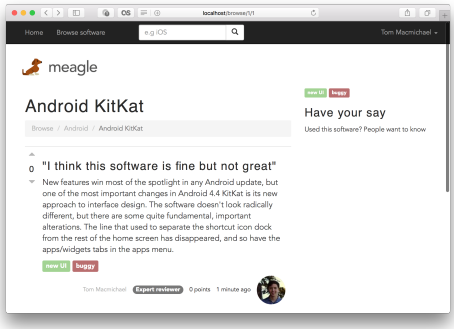

Figure 4.26: Actual reviews page after Sprint 2

4.3.10 Minor features

- **Ability to search**. This allowed users to search by version name or software name.

- **Longer reviews**. Previously reviews only had a body and no title. Comparing with other platforms, this felt inappropriate. Users did not know how much to write, leading to varying review size. A new text field was added: allowing a title and a body for every review.
- **SEO**. Metadata, such as page titles and descriptions, were added to each page so that it could be easily indexed by search engines.

4.3.11 Updated tests

When developing the search box, a functional test was written that would test to see whether the search functionality was performing correctly, including form validation. The test, shown in Figure 4.27, visits the homepage and performs an empty query and verifies the form validation error is shown. These types of tests are a powerful way of verifying that the entire stack is performing correctly: the frontend, the routing, the forms, the search, and the validation.

```
public function testEmptySearch()
{
    $this->visitRoute('home')
        ->type('', 'query')
        ->press('search')
        ->see('The query field is required.');
}

```

Figure 4.27: Test which checks whether the search functionality on the homepage is performing correctly

4.3.12 Evaluation and feedback

This sprint introduced several new concepts such as consensus boxes and tags. It also introduced a more sophisticated crowdsourcing system with automation. After speaking with the focus group, the majority of feedback was positive. The main piece of feedback was that users wanted a place on the site where they could find information if they were stuck or did not understand something. These users also suggested that the site should include a single rating for a piece of software, allowing for quick glances without having to read all the reviews.

4.4 Sprint 3

This sprint started on 23rd February 2017.

4.4.1 Design brief

Using the feedback from the previous release, the following user stories were created. It was also decided that the crowdsourced nature of the site needed to be enhanced, so that users with more points could have greater influence on the site and users that who have behaved badly on the site would have little to no influence.

- "I want to find helpful information on how to use the site"
- "I want to be able to see information about how good a piece of software is without reading through all the reviews"
- "I want to be able to search using partial terms"
- "I have lots of points - I should be able to delete and add content more easily"

4.4.2 Crowdsourced questions and answers

One of the user stories was the ability to find helpful information on the site. As a crowdsourced platform, it seemed appropriate to implement the help page using crowdsourcing. This is something that none of the platforms looked at in background research appeared to have.

Utilising the new abstract `Votable` class described in Section 4.3.5, it was simple to implement a crowdsourced help page. A new `Question` class was created that extended the `Votable` class. This is shown in the updated UML diagram in Figure 4.34. Database migrations were created to update the database schema.

Reusing views

As every `Votable` class shares the same methods and properties, the views previously created for displaying information about other `Votables` could be re-used. This meant the views created for voting on a piece of software could immediately be used again for `Questions`. This resulted in less duplication of code and consistency across the site as all voting boxes worked in the exact same way.

As explained in Section 4.3.5, the parent `Votable` class's `confirmReal()` method is called whenever a `Votable` reaches a particular number of (positive) votes. As the `Question` class is extending this, it inherits this functionality also. This meant that when a `Question` reaches a particular number of votes, it is confirmed and the voting is closed.

At a database level, a `Question` is different from a `Review`, however this inheritance property allows them to be treated as if they were the same – a principle of object-orientated programming languages.

A form to add questions was also added, which required an additional form validator and controller to take user input and store it in the database. The final

result is shown in Figure 4.28. A help button was added in the bottom-right of every page, always visible. If a user clicks then they are take them to this page.

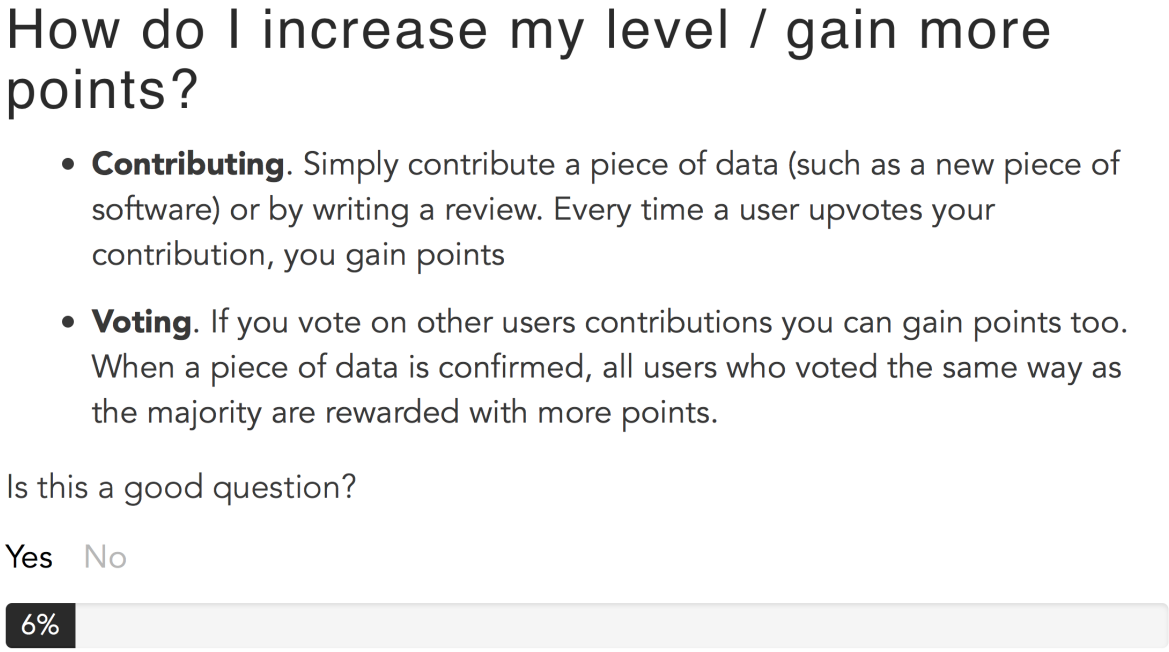

Figure 4.28: Diagram showing a crowdsourced question

4.4.3 User levels and voting power

Previously, any user on the site had an equal vote (equivalent to 1). The issue with that was that malicious users could make multiple accounts and repeatedly vote on the same item and eventually it would be confirmed as real. On StackOverflow this is not so much of an issue for several reasons: answers on StackOverflow are objective – there is not much to gain by continually upvoting something bad other than to annoy other users; secondly, authors of questions confirm the answer rather than other users; and finally, users with greater reputation have greater privileges, allowing them to revert any false choices by authors and punish bad users [11].

On Meagle, a slightly different approach was taken with a new concept of "voting power". Instead of allowing top users the ability to edit, delete and change others' contributions, a more seamless approach was chosen. The concept instead was to give users with more points the power to upvote and downvote something as if it were done by more than one user. This effectively gives top users the ability to confirm or delete data quicker, with individual users having far less influence.

This was implemented by adding a new attribute to the `User` class called `voting_power` with a default value of `1`, and stored in the database as an additional column. The different voting powers were configured into levels: given a certain amount of points, a user would be given a particular voting power. These different

buckets were defined within `config/crowd-sourced.php`, allowing easy removal or addition of extra levels. Every time a user votes, their vote was simply their voting power. For example, if their voting power is 20, then every time they upvote something it counts as 20 basic user votes, and similarly if they downvote something this counts for -20 votes. If a user has *negative* points, then their voting power is 0 meaning they have no influence in the site.

This was implemented using a new observer class on the `Vote` class, following the observer design pattern [67]. Any time a `Vote` is created or deleted, the observer is notified. This is then used to call the `updateVotingPower()` method on a user. This in turn calculates a new voting power for a user and updates their voting power and points in the database. This is shown in Figure 4.29. Two extra configuration variables were also added: `confirm_at` and `reject_at`. These define how many votes are required before a piece of data is confirmed as real, or deleted.

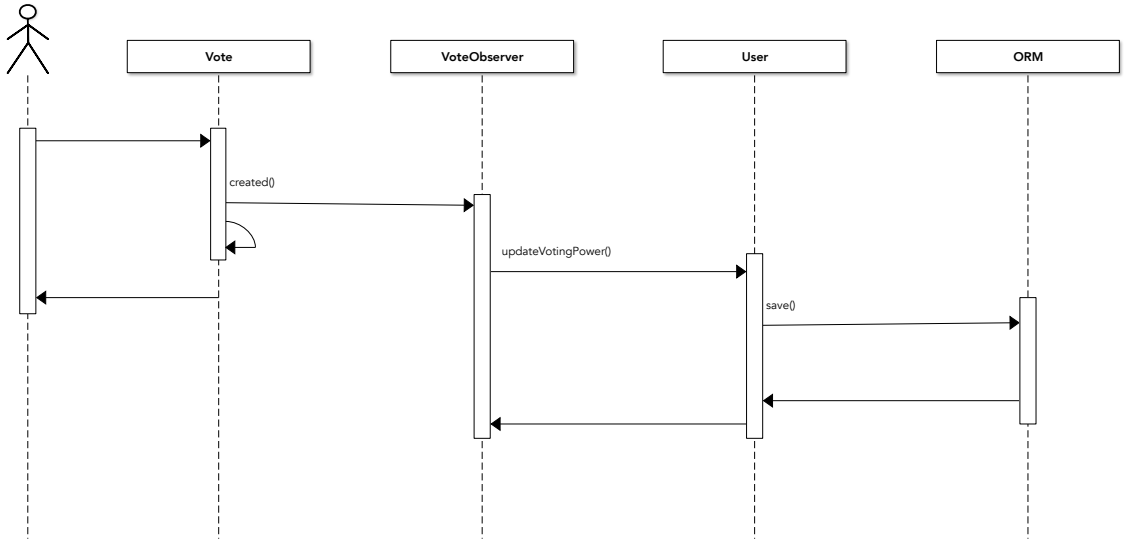

Figure 4.29: Sequence diagram showing flow of data as a user votes

User levels

To make it clearer which reviews were written by "top" users (those with higher voting power) and those from newer users, the concept of "user levels" was added. Next to every review, the user's level was included. User levels give context to users as they show a user's level relative to the highest possible level, rather than just an arbitrary number. StackOverflow has a similar concept, where the more reputation you have the more privileges you have – at certain stages user's gain a new privilege. However, they do not display this information as a "user level", so users unfamiliar with the site do not necessary know how respected a user is relative to the community.

- written by Tom, Level 1/5 1 month ago

Figure 4.30: Screenshot showing user level next to the author of a review

4.4.4 Verdict box and top tags

One of the user stories in the design brief was *"I want to be able to see information about how good a piece of software is without reading through all the reviews"*. There are several ways to implement this, as discussed in Section 2. An approach similar to Glassdoor was chosen, where the top negative and positive comments repeated in reviews are highlighted as shown in Figure 2.12. As users are able to add negative and positive tags to their reviews from the previous sprint, this was utilised in this update to show the most frequently used tags in the top-right of a software page as shown in Figure 4.31.

All the other sites reviewed in Section 2 also have an easy quantifiable measure of what users think of a piece of software, generally in the form of a star rating. This was something that was avoided when developing Meagle, as the goal was to offer something unique, with a focus on content rather than on an arbitrary number of how good a piece of software is.

Glassdoor take the approach of asking users whether they have a positive, negative or neutral outlook on a particular company. This approach seemed better as it is less subjective – "5 stars" varies from user to user, whereas "positive or negative" is fairly consistent. Rather than ask users another question, the information given in the form of positive and negative tags from the previous release was used. This lead to the creation of a "verdict box" – summarising whether users had given more negative tags than positive ones. The box is coloured red and green to highlight what percentage of reviews are negative and positive respectively. Text above the bar summarises the meaning for clarification.

To implement this, several Laravel accessor attributes were added. This allowed for the syntax of `$version->verdict`, `$version->percentagePositive`, `$version->percentageNegative`. These percentages represent what proportion of tags, attached to reviews for this version of software, are positive and negative respectively. If the percentages are almost equal, the `verdict` attribute returns "no consensus". Any reviews that have been deleted do not count towards the verdict.

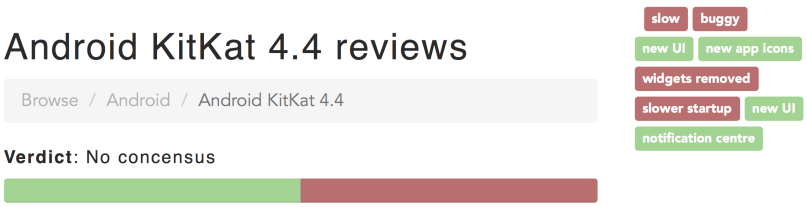

Figure 4.31: Screenshot showing verdict box and top tags for Android KitKat 4.4

4.4.5 Markdown text editor

Another design requirement was to make reviews more readable, engaging and thorough. This required allowing users to customise the HTML of their displayed content. However, allowing any HTML content poses a security risk as it allows injection of arbitrary code onto the page which can lead to session-stealing [69]. Instead it was decided that the site should use markdown instead. Markdown is a lightweight alternative to HTML that supports basic formatting with a clean syntax. It is used on StackOverflow, Wikipedia and many other sites [10].

Rather than build a markdown editor from scratch, the frontend dependency manager setup was used to install an open-source markdown package known as simplemde [70]. Installing this dependency involved running the command (`bower install simplemde --save`). This updated the "bower.json" file to reflect the new package. The automation file was then updated to copy the files from this dependency into the public directory, on deployment. This left the git repository free from the dependency files, while ensuring on deployment the dependency would be available.

The markdown editor, shown in Figure 4.32, was added to both the review page as well as the question contribution page. At a database level, the data is stored in markdown. The next challenge required displaying markdown on the page to the user. Here, another open-source package that converts markdown into HTML was used. The package operates within Laravel Blade views [71]. This was installed using the composer dependency manager (`composer install laravel-markdown`).

Users can now add formatting to their reviews, the site is still protected from code injection, the database is free of HTML, and the git repository is clean from dependencies (as only two files were updated). Also, any users who has Javascript disabled (and therefore cannot see the markdown editor) can still submit a formatted review, using markdown's syntax which is generally considered simpler than HTML.

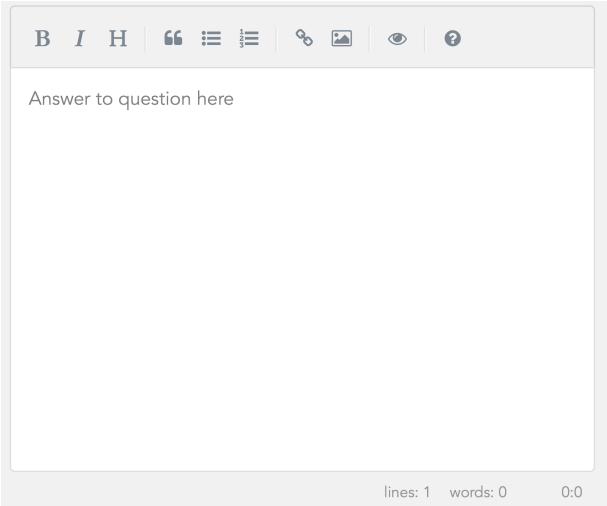

Figure 4.32: Screenshot showing markdown editor

4.4.6 Minor features

Improved search. Previously the search could only find a piece of software, given part of that software name **or** part of that version name; given both the version name and the software name together would give no results. This was a top issue highlighted by stakeholders in the previous release. In this release the query scope was expanded so that it could handle more difficult queries (such as as Android Kitkat) where Android is the software name and Kitkat is the version name.

User area. A user area was added as a homepage after a user logged in, and a central place where they could find information regarding their account such as the number of points they have and their level. In future versions the user area could be used for account management, such as changing email.

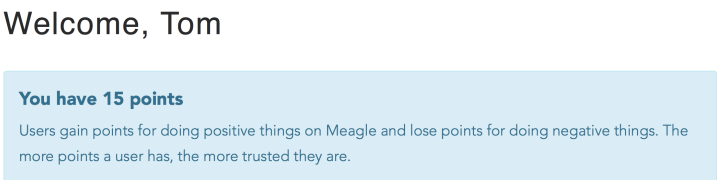

Figure 4.33: Screenshot from the user area page showing a user a welcome message and the number of points they have

4.4.7 Updated architecture

As this release included some changes to the models used by the site, there were some changes to the architecture. These are shown in Figure 4.34. Database migrations were created to update the database schema to reflect the changes to the models.

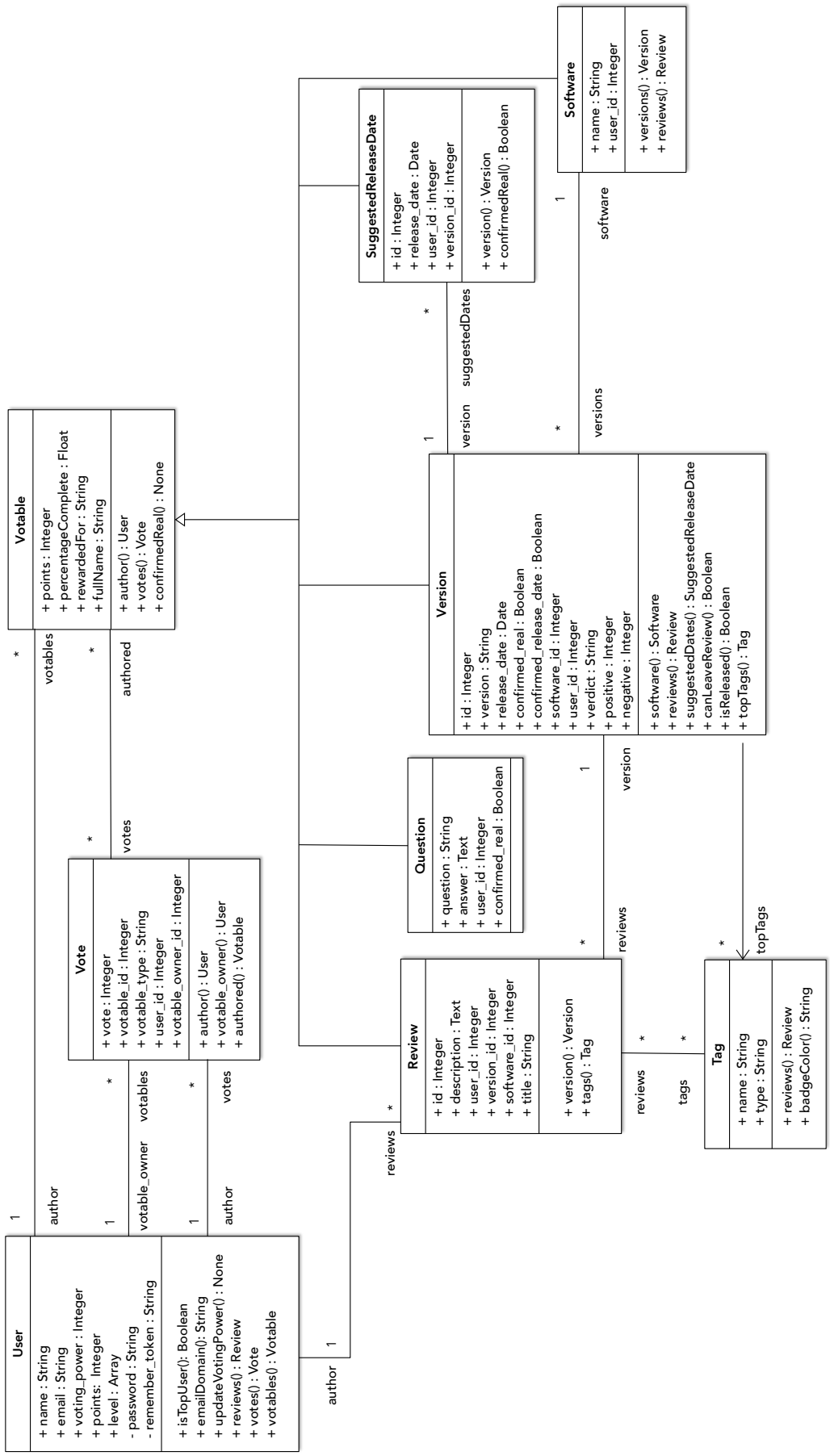

Figure 4.34: UML diagram of Eloquent classes after Sprint 3

4.4.8 Updated tests

To ensure voting power and user levels were working correctly several unit tests were written.

- **GetLevelTest.php** Given a particular number of points, ensure the - `$user->level` property returns the correctly level.
- **GetVotingPowerTest.php** Given a particular number of points, ensure the `$user->voting_power` method returns the correct voting power.
- **UpdateVotingPowerTest.php** Given a new set of points, after saving the user, its voting power should be updated.

These tests ensured the site correctly performed. An example test is shown in Figure 4.35. The user object is mocked using the Mockery framework [55], allowing isolation the particular method being tested.

```
public function testVotingPower()
{
    $this->userMock->points = 55;
    $voting_power = $this->userMock->getVotingPower();
    $this->assertEquals(10, $voting_power);
}

```

Figure 4.35: Test which checks whether the search functionality on the homepage is performing correctly

Chapter 5

Final evaluation

After the final sprint a survey was designed that took users through the site's features, asking them to perform tasks and comment on the ease of use and overall design of the site. There were 14 responses from the survey, including members from the focus group that were used throughout the development. The structure of the questions was in the form of a statement given, in which respondents were asked how strongly they agreed with a statement.

5.1 Responses

5.1.1 Basic tasks

Respondents were asked to perform basic tasks such as making an account and logging in. Overall, users did not seem to have an issue with these tasks, however one user did report it was hard to find "Android KitKat". After asking this individual further, it was established that they had used the mobile website which does not feature a search box on every page. This could easily be resolved by adding a search icon on the mobile site, allowing quicker access to search.

The results for this section are in Figure 5.1.

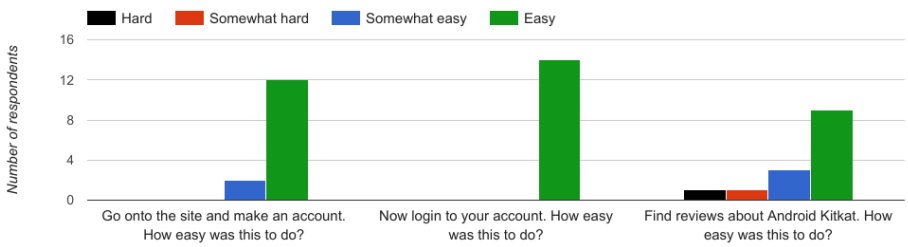

Figure 5.1: Responses for the basic site tasks

5.1.2 Upvote / downvoting

Respondents were told the following statement: "Upvote / downvote a review by pressing one of the arrows, then answer the following question". This section focused on whether the interface for votes was clear enough and not overly simplified.

While the majority understood what the number next to the review meant, there were a few who did not. The biggest cause for confusion was the number in between the voting icons. A solution to this could be in the form of showing new users a tutorial when they first use the site. An open-source tool for doing this is called intro.js, shown in Figure 5.3. It could be used to highlight and explain features when users first use the site.

The results for this section are in Figure 5.2.

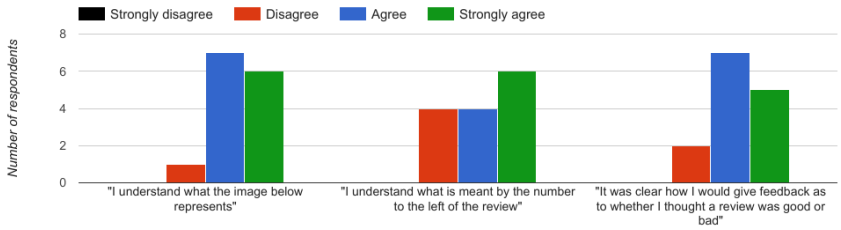

Figure 5.2: Responses regarding the voting user interface

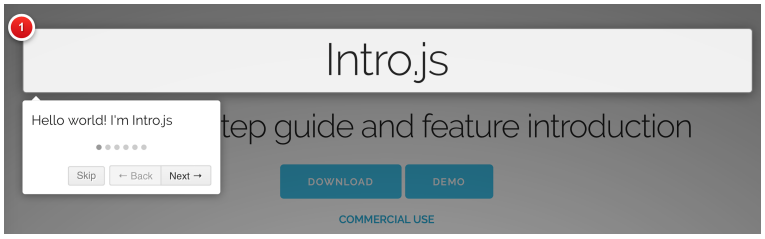

Figure 5.3: An example of intro.js being used as a tutorial on a website

5.1.3 Contributing

This section looked at how clear contributing data on Meagle is and whether users understood the process. It asked users to add a piece of software and then asked whether they understood why no-one could write reviews for it straight away.

The greatest issue here was understanding why the software they added was not immediately reviewable. Figure 5.5 shows the error message displayed to users when viewing a piece of software that has just been added. A comment would be to more explicitly state "this software cannot be reviewed yet" and perhaps try and simplify the reason why. Currently users are directed to the crowdsourced process page, which may be overwhelming for some users.

The results for this section are in Figure 5.4.

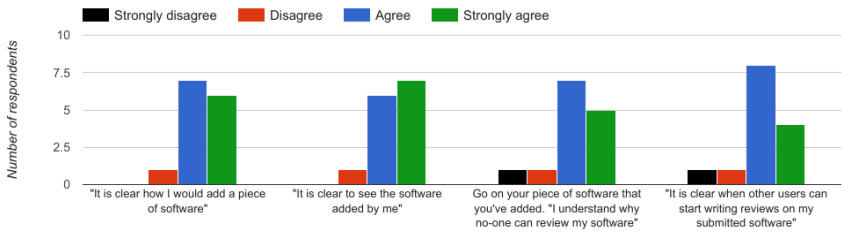

Figure 5.4: Responses regarding contributing data

This software has not been confirmed as real yet. Software versions are crowd-sourced on Meagle. To find out more about the process of review, read here.

Figure 5.5: Screenshot of an error message shown to users who have just added a piece of software

5.1.4 Navigation

This section focused on the site navigation and search. Several users found it difficult to find their user profile. The way the link to the user profile was implemented was by displaying the user's name in the top-right of the screen. This is a fairly common approach used by Facebook, Twitter, Github, StackOverflow and many other sites. However, it is worth noting that on mobile devices the link is hidden within a menu (as shown in Figure 5.7). A possible solution to make it clearer would be by the inclusion of a profile icon – either an uploaded photo of the user, or a generic user icon. This would make the site more consistent with other social networking platforms.

The results for this section are in Figure 5.6.

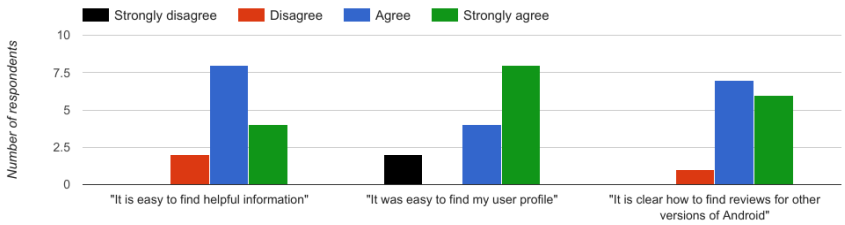

Figure 5.6: Responses for navigation related questions

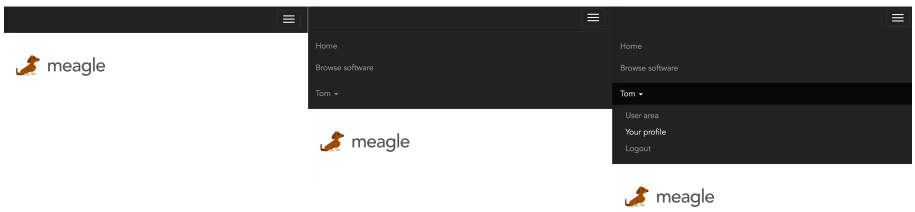

Figure 5.7: Screenshots showing how to access the user profile on a mobile device

5.1.5 Reading reviews

This section focused on the display of the reviews on the site. The most contentious issue was the ability of a user to determine how trustworthy a review was. This was a frustrating, yet important, piece of feedback; the implementation of user levels had aimed at solving this problem. A potential solution would be change the terminology from "level" to "trustworthiness". Another possibility would be extend the "verified users" feature, which displays a badge next to users who have verified their email address from a trusted source.

The results for this section are in Figure 5.8.

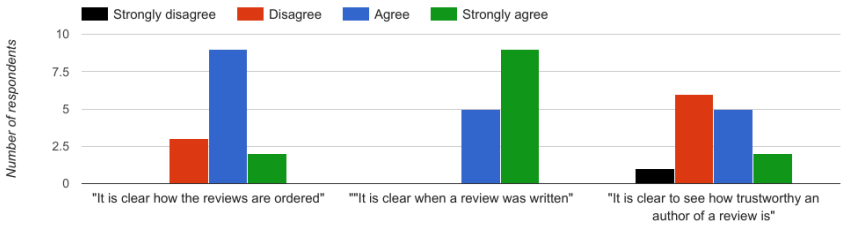

Figure 5.8: Responses for the questions regarding reviews

5.1.6 Analysing

This section looked to see whether users could analyse the reviews for a piece of software easily. Users generally did not have an issue reading the software summaries, however there were still a few who did not think it was clear what the problems with the software were. One way to resolve this would be to expand the "top tags" section, introduced in sprint 2. Rather than use tags to highlight the negative aspects of a piece of software, an approach more similar to Glassdoor could be used: highlighting top "negative" sentences at the top of a piece of software, rather than just negative tags.

The results for this section are in Figure 5.9.

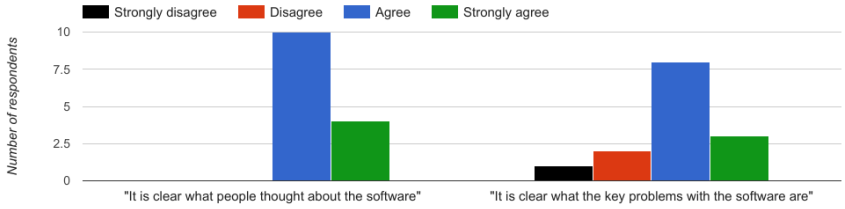

Figure 5.9: Responses regarding analysing reviews

5.1.7 User interface

Users reported that the site was easy to navigate, and in general users liked the design of the site. The site design was mentioned in the additional feedback section over 5 times. However, some users still disagreed. This is something that could be worked on by doing design-specific research with more users, however as design is far more subjective than usability it is something that will not necessarily ever have a complete positive consensus.

The results for this section are in Figure 5.10.

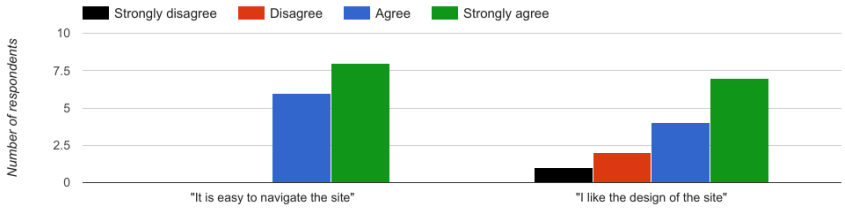

Figure 5.10: Responses for user interface related questions

5.1.8 Points

This section looked at whether users understood the points and voting power concepts. Some, but not many, users also had difficulty understanding how their points were calculated. This information can be found on a user's profile, as shown in Figure 4.18. There is a lot of information that needs to be shown to the user without overwhelming them. To help overcome this confusion, further research could be done regarding this specific issue and try and narrow down the problem.

In the feedback form at the end, some users expressed confusion regarding the difference between "points" and "voting power". Although users could find this information by hovering over the text and seeing tooltips, a way to avoid this confusion would be to keep the underlying voting power concept, but not tell users about it. This would keep the functionality of rewarding top users without overloading users with information they do not necessarily need to know.

The results for this section are in Figure 5.11.

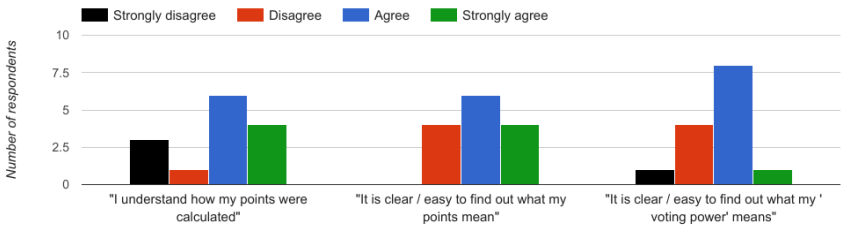

Figure 5.11: Responses for questions regarding voting power and points

5.1.9 Voting

This section asked users to use the different aspects of the voting interface. The most difficult task reported by users was deleting a vote – this involved clicking the vote icon again, turning it from black to grey. The flow of interaction is shown in Figure 5.13. One solution would be the use of tooltips to tell users how to delete their vote, rather than just rely on user intuition.

The results for this section are in Figure 5.12.

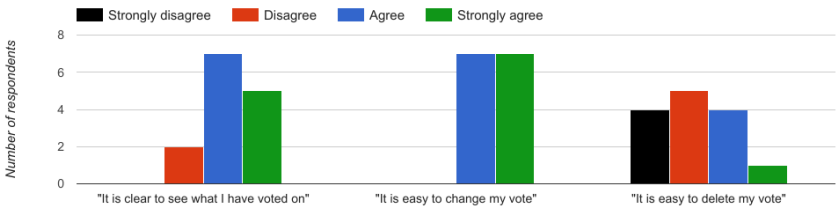

Figure 5.12: Responses for questions regarding voting

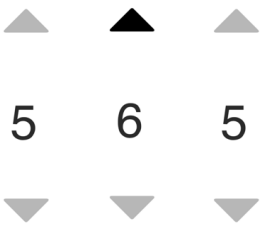

Figure 5.13: Left: before voting, middle: after pressing the upvote, right: after pressing the upvote again - deleting the vote

5.2 Summary

The feedback received was overwhelmingly positive. Users were able to contribute data to the site, read reviews, and navigate the site. Most of the concerns with minor and could easily be resolved using the suggestions outlined in the previous sections.

5.3 Usability

Meagle.org passed Google's mobile-friendly test which analyses webpages on a number of criteria including page speed, button size, readability and layout [72]. This is unsurprising as the site was built on Bootstrap's responsive framework, which is optimised to pass these criteria.

5.4 Performance

Pingdom is a commercial uptime and performance monitoring tool and was used to test and compare the page load time against comparable platforms. Its speed test uses real instances of Google Chrome browsers on numerous computers around the world to identify bottlenecks and measure page speed [8].

When tested, Meagle had a smaller page size and loaded faster than every other site mentioned in the related work section, shown in Figure 5.14. This is partly due to the fact Meagle contains no images other than the logo, but also due to the techniques outlined in this report such as a minification and combination of stylesheets, utilising Cloudflare's CDN for caching, Laravel as a framework, and Heroku's servers.

The tool also displays a Google PageSpeed Grade, which is on a grading scale from A to F, where A is excellent and F is extremely poor. This is an easy way to compare different sites as the grade considers various different aspects of the site, and then offers suggestions on how to improve it. Although Meagle's grade was B, there was only one suggestion given for improvements – to host assets, such as images as stylesheets, on a cookieless domain. This is because currently whenever this content is requested, cookie data[1] is sent with the request, which increases the time it takes to load. As this content is static it does not need cookie data; it will never change, regardless of what data is sent to it. However, as the cookies were set on meagle.org, any request to meagle.org will include include this data. Google's suggestion is to have all static content on another domain,

[1]Cookies are pieces of data stored in a user's browser, associated with a domain. The browser sends this data every time a request is made to that domain. A website can set cookies in a user's browser by sending cookies to users who visit the website. Cookies are used on Meagle as a way of remembering who is currently logged in.

such as static-content.com – any requests to this domain would not receive the meagle.org cookie data.

In all cases, the homepage of the site was used for testing.

	Google PageSpeed Grade	Page Size	Load Time
Meagle	**B**	**187.2kB**	**632ms**
CNET	C	2.7 MB	2.34s
TrustPilot	C	633.4kB	1.62s
AlternativeTo	C	899.6Kb	2.83s
TrustRadius	B	1.7 MB	1.9 s
Wired	C	5.2 MB	60s
TechCrunch	C	2.9 MB	4.7s
StackOverflow	C	560 kB	1.65s

Figure 5.14: Table showing comparison between homepage load times of various software review sites

5.5 Cost

Another way to evaluate Meagle is to look at the total cost of building and running it. The only initial cost was $3.17, which was to register the domain from 1&1.com.

CircleCI, GitHub, Cloudflare are all free to use. The database was being run off a "micro" package at Amazon RDS. This is free for the first 12 months, after which it is no more than $13 a month[2].

While developing, the site was on Heroku's free package. The only downside of this package was that if no-one has accessed the site recently, there was a slight delay when someone tried to access it. After a while the site was upgraded to a "hobby" package at Heroku, at a total cost of $7 a month, which has no delay.

If the site had increased demand these packages would likely need changing. However, in its current form the site costs very little to run, despite using sophisticated technology and development practices.

[2]This is the cost of running the database 24/7 for a month. As Amazon charge by the hour, and the database is unlikely to be used constantly, the expected cost could actually be less.

Chapter 6

Conclusion

Meagle.org is a complete software review platform. It has crowdsourcing abilities to build a database of software versions and their release dates, as well as containing a self-moderating community of reviews.

This project outlined how to setup a continuously integrated software platform following agile methodology, secured using the latest technologies, easily scalable, with a maintainable and extensible codebase.

To summarise, the following modern web development practices were used throughout the project:

- Continuous integration (CircleCI)
- Version control (Git)
- Code-to-deploy (push to GitHub, build on CircleCI, release on Heroku)
- Testing (phpunit)
- Dependency management (Bower, Composer, NPM)
- CSS pre-processing (LESS)
- Workflow automation (Gulp)
- Javascript and CSS minification on deployment (Gulp)
- Responsive design (Bootstrap)
- Forced SSL (Cloudflare)
- Secure framework as a foundation (Laravel)
- Scalable relational database (Amazon RDS)

6.1 Final features

The final feature set included:

- **Responsive design**. Shown in Figures A.1 and A.2.
- **Browse**. Shown in Figure A.3.
- **Crowdsourced software**. Shown in Figure A.5.
- **Formatted reviews with tags**. Shown in Figure A.6.
- **User area**. Shown in Figure A.7.
- **User profile**. Shown in Figure A.8.
- **Crowdsourced help page**. Shown in Figure A.9.
- **Crowdsourced release dates**. Shown in Figure A.11.
- **Search**. Shown in Figure A.12.
- **Self-moderating community**. The site moderates itself, through upvoting and downvoting on reviews. As well as through rewarding and punishing users who vote correctly and incorrectly.

6.1.1 Innovations

Meagle.org seemingly is the only site of its kind: one place where people can write reviews about, or find information on, any specific version of software. While completing background research, a site that had the exact same functionality as Meagle could not be found. Other sites either had good information on software, in the case of AlternativeTo, but not a specific update; or, sites had good reviews on software, but in a more biased way, such as CNET.

While the site borrowed ideas from other platforms, there were several innovations specific to Meagle while developing the platform: such as consensus boxes, voting power, rewarding the majority with points as means of moderation, and crowdsourced help pages.

These features, combined with Meagle's clean, advert-free user interface, make it a unique platform.

6.2 Future work

6.2.1 As Meagle.org

As the site is fully published at Meagle.org, anyone in the world can register and continue to contribute to Meagle. As a crowdsourced community it can

continue to be a self-maintaining platform. Some additional features that could be implemented are:

- **Adding comments to reviews.** This would not be too difficult to implement and is a relatively consistent feature on other similar platforms.
- **Enhancing the breakdown of reviews** so that users could find reviews from authors similar to them: for example, reviews by authors who also upgraded from their currently operating system.
- **Crowdsourcing other data about software.** This could be added to build up a more sophisticated database rather than just rely on the software release dates.
- **Allowing companies to respond to reviews.** This is something that was possible on most of the other platforms. One of the key issues with implementing this is verifying that it is actually a representative from the company responding, and not an impersonator.
- **Extending the verified users concept.** Meagle currently displays a badge next to a user's review if their email address was from a certain domain, such as a university. This could be extended so that these users start with at a greater level / voting power.

6.2.2 As an open-source platform

Meagle is a self-maintaining crowdsourcing platform, as well as a self-moderating community platform. As such, it can be used as one, or the other, or as both. It can be used for crowdsourcing some particular, objective, goal; or, it can be used as a self-moderating community of opinion.

While built specifically for software reviews, with a few small changes, users could deploy a voting community of their own on any specific topic. It is extensible, transparent and well documented. As the source code is available freely on github.com [3], anyone can download a copy of the code and develop their own crowdsourced platform or their own community review platform. This system of voting, with confirmation and deletion of data, rewarding and punishing users could be applied in various scenarios.

A few possible use cases are:

- **YourTaximeter.com** – a site that manually maintains a database of over 1,200 taxi tariffs set by local authorities across the United Kingdom, which are regularly updated [73]. Meagle's underlying system could be used to crowdsource and maintain this data instead.
- **IsItBusy.in** – this site uses public data to predict busyness of buildings. Incorporating Meagle into the platform would allow the site to improve its estimations using crowdsourced data from people in the buildings, as well as the ability to add new places.

- **Festifind.com** – a music festival search engine which manually curates information regarding festivals, such as ticket price, and artists playing. Meagle's crowdsourcing could be used to maintain, improve, and add this data, reducing the amount of manual work required by the site's developers. In parallel, Meagle's community-moderated reviews could be used for festival reviews.

6.2.3 Extending the project

All of the source code for Meagle is available on GitHub [3]. To extend the project, users can create a pull request – this is the process of proposing code changes into a repository. If approved by a "collaborator"[1], it would be merged into the codebase, which would immediately trigger a build at CircleCI and start the release process.

People can also fork[2] the repository and start developing their own version of Meagle.

[1] A collaborator is a person with read and write access to a repository who has been invited to contribute by the repository owner [74].

[2] A fork is a copy of a repository. Forking a repository allows the ability to freely experiment with changes without affecting the original project [74].

Appendix A

Screenshots of the final site

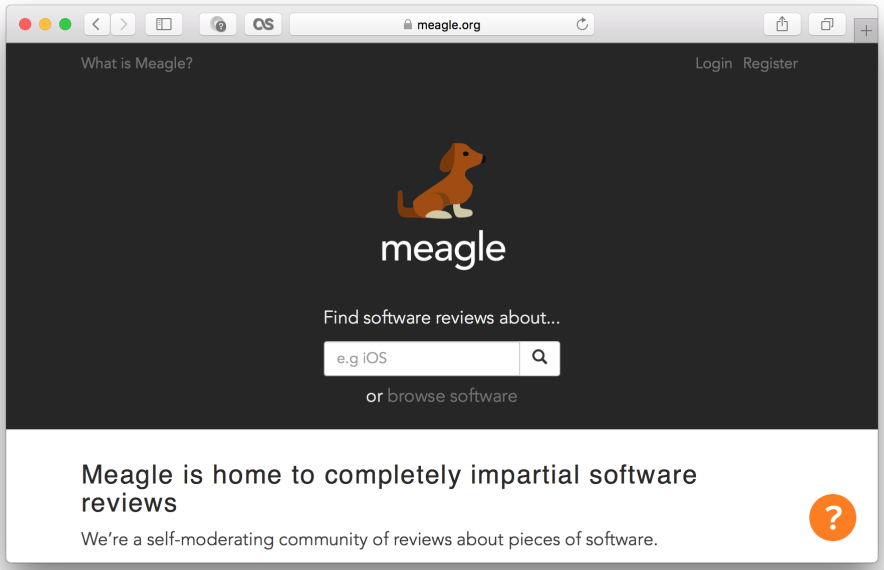

Figure A.1: Screenshot showing the site homepage

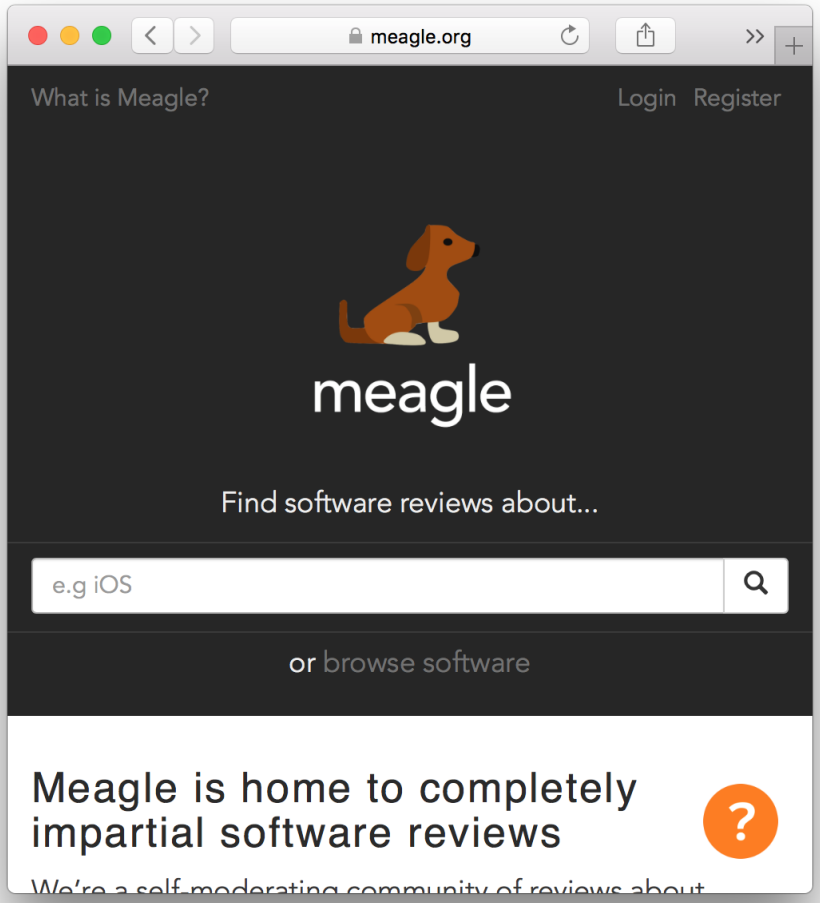

Figure A.2: Screenshot showing the site homepage on a different screen size

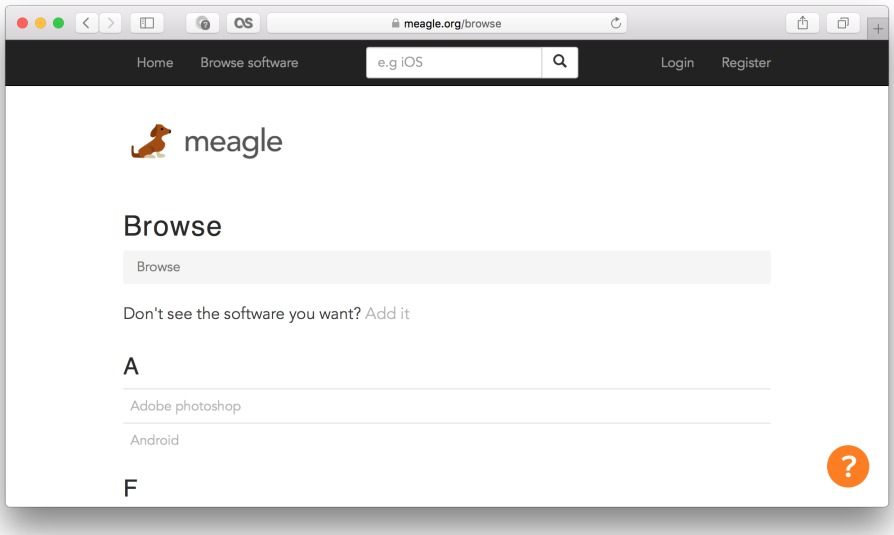

Figure A.3: Screenshot showing the "browse software" page

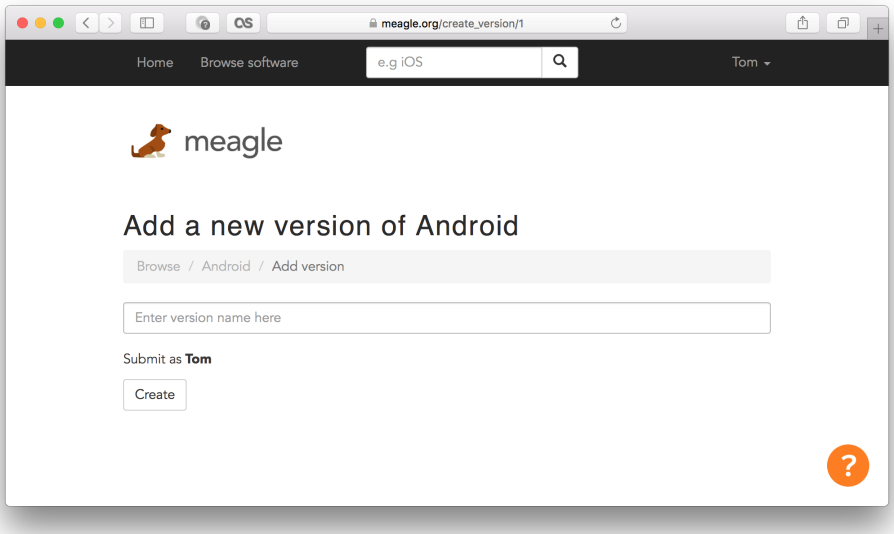

Figure A.4: Screenshot showing the form to add a version of software

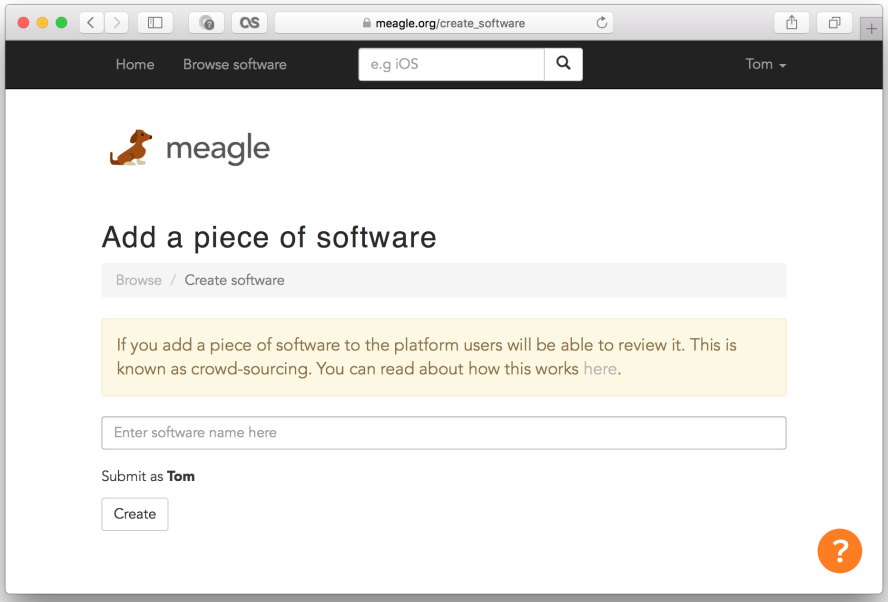

Figure A.5: Screenshot showing the form to add a piece of software

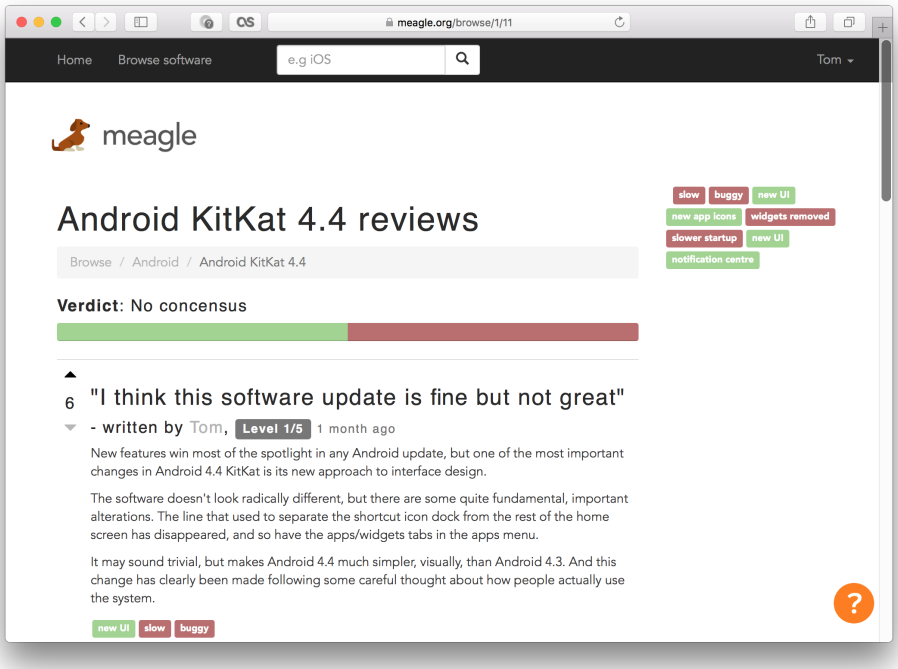

Figure A.6: Screenshot showing a review page

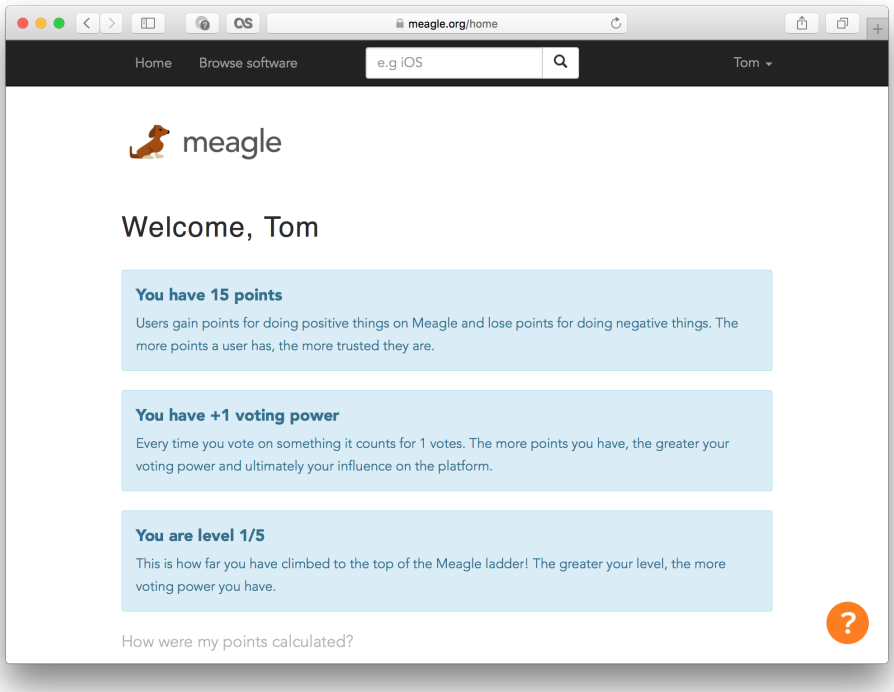

Figure A.7: Screenshot showing the user area

Figure A.8: Screenshot showing a user's profile

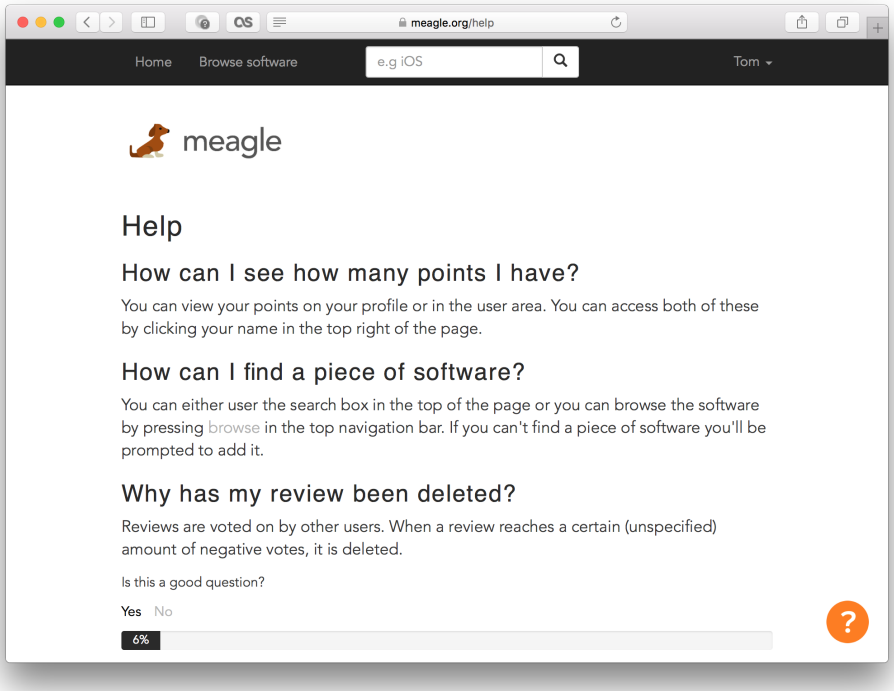

Figure A.9: Screenshot showing the crowd-sourced help page

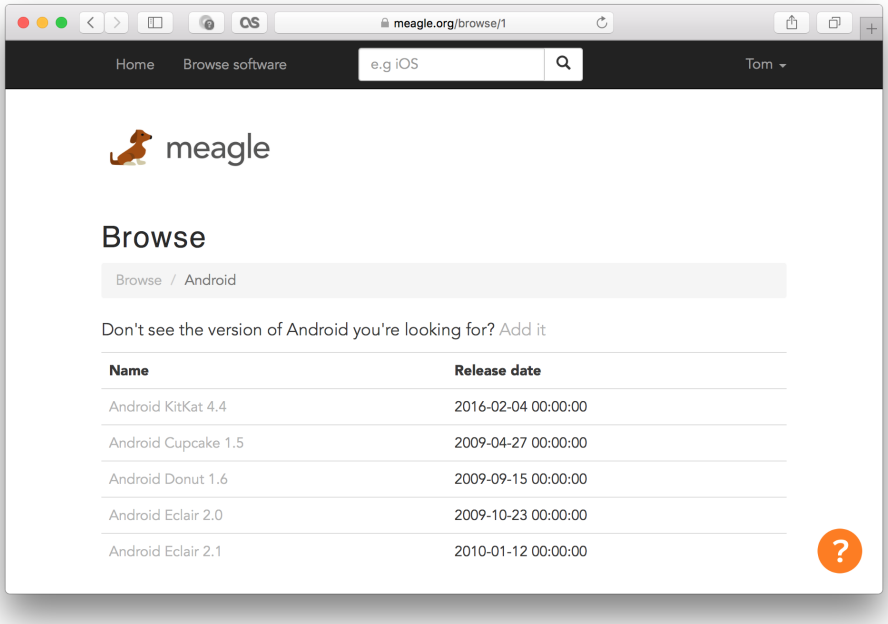

Figure A.10: Screenshot showing different Android versions on the “browse software” page

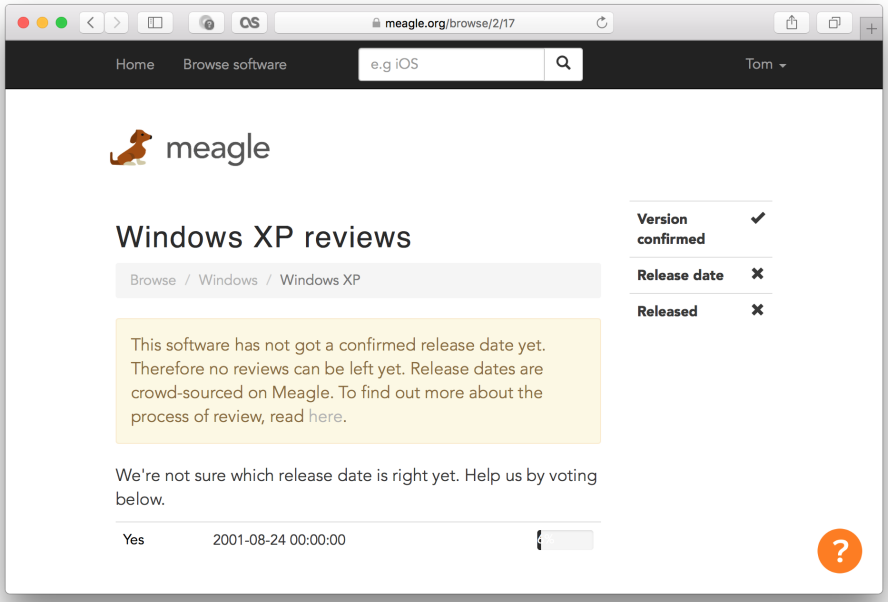

Figure A.11: Screenshot showing the form to suggest a release date

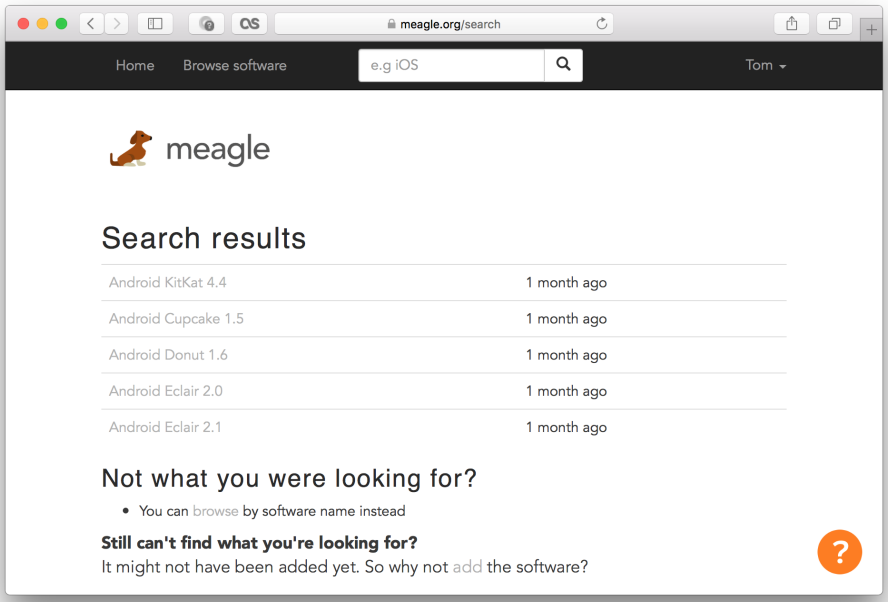

Figure A.12: Screenshot of example search results for “Android”

Bibliography

[1] "Average number of apps vs time spent - business insider." http://uk.businessinsider.com/average-number-of-apps-vs-time-spent-2016-5. (Accessed on 03/22/2017).

[2] K. Vaniea and Y. Rashidi, "Tales of software updates: The process of updating software," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pp. 3215–3226, ACM, 2016.

[3] "maccery/meagle-php: A modern software rating system, with transparency and beauty in mind." https://github.com/maccery/meagle-php. (Accessed on 03/28/2017).

[4] J. Howe, "The rise of crowdsourcing," *Wired magazine*, vol. 14, no. 6, pp. 1–4, 2006.

[5] A. Doan, R. Ramakrishnan, and A. Halevy, "Crowdsourcing systems on the world-wide web," *Communications of the ACM*, vol. 54, no. 4, pp. 86–96, 2011.

[6] D. Anastasiou and R. Gupta, "Comparison of crowdsourcing translation with machine translation," *Journal of Information Science*, vol. 37, no. 6, pp. 637–659, 2011.

[7] "recaptcha: Easy on humans, hard on bots." https://www.google.com/recaptcha/intro/. (Accessed on 04/04/2017).

[8] "Website speed test." https://tools.pingdom.com/. (Accessed on 03/28/2017).

[9] M. Levene, *An Introduction to Search Engines and Web Navigation*. Wiley, 2011.

[10] "What is markdown?." http://whatismarkdown.com/. (Accessed on 03/28/2017).

[11] "What is reputation? how do i earn (and lose) it? - help center - stack overflow." http://stackoverflow.com/help/whats-reputation. (Accessed on 03/28/2017).

[12] P. Belton, "Navigating the potentially murky world of online reviews - bbc news." http://www.bbc.co.uk/news/business-33205905, June 2015. (Accessed on 03/30/2017).

[13] B. Rubin, "Amazon continues crackdown on alleged fake reviews," Apr 2016.

[14] R. Loyola, "Apple will allow developers to respond to app store user reviews — macworld." http://www.macworld.com/article/3161216/mac-apps/apple-will-allow-developers-to-respond-to-app-store-user-reviews.html, January 2017. (Accessed on 04/04/2017).

[15] S. D and K. Runyan, *Introduction to Agile Methods*. Pearson Education, 2014.

[16] I. Stamelos, *Agile Software Development Quality Assurance*. IGI Global research collection, Information Science Reference, 2007.

[17] C. Cobb, *The Project Manager's Guide to Mastering Agile: Principles and Practices for an Adaptive Approach*. Wiley, 2015.

[18] D. Feitelson, E. Frachtenberg, and K. Beck, "Development and deployment at facebook," *IEEE Internet Computing*, vol. 17, no. 4, pp. 8–17, 2013.

[19] GitHub, "Mastering issues github guides." https://guides.github.com/features/issues/, April 2014. (Accessed on 03/28/2017).

[20] M. Issac and Q. Hardy, "Facebook to shut down parse, its platform for mobile developers - the new york times." https://bits.blogs.nytimes.com/2016/01/28/facebook-to-shut-down-parse-its-platform-for-mobile-developers/, January 2016. (Accessed on 03/28/2017).

[21] C. Metz, "Facebook buys its way into the heart of the app world — wired." https://www.wired.com/2013/09/facebook-and-parse/all, December 2013. (Accessed on 03/28/2017).

[22] C. Metz, "Facebook introduces hack, the programming language of the future — wired." https://www.wired.com/2014/03/facebook-hack/, March 2014. (Accessed on 03/31/2017).

[23] O. Livneh, "How we made editing wikipedia twice as fast wikimedia blog." https://blog.wikimedia.org/2014/12/29/how-we-made-editing-wikipedia-twice-as-fast/, December 2014. (Accessed on 03/31/2017).

[24] "Stack overflow developer survey 2016 results." http://stackoverflow.com/insights/survey/2016. (Accessed on 03/29/2017).

[25] "php/php-src: The php interpreter." https://github.com/php/php-src. (Accessed on 03/29/2017).

[26] "Php: rfc:releaseprocess." https://wiki.php.net/rfc/releaseprocess. (Accessed on 03/29/2017).

[27] "Web application frameworks." https://github.com/showcases/web-application-frameworks. (Accessed on 03/29/2017).

[28] "Release notes - laravel - the php framework for web artisans." https://laravel.com/docs/5.4/releases. (Accessed on 03/31/2017).

[29] K. Roebuck, *Object-Relational Mapping: High-impact Strategies - What You Need to Know*. Lightning Source, 2011.

[30] "Installation - laravel - the php framework for web artisans." https://laravel.com/docs/5.3. (Accessed on 04/03/2017).

[31] I. Thomson, "Cloudbleed: Big web brands 'leaked crypto keys, personal secrets' thanks to cloudflare bug the register." https://www.theregister.co.uk/2017/02/24/cloudbleed_buffer_overflow_bug_spaffs_personal_data/, February 2017. (Accessed on 03/22/2017).

[32] "Explainer: what is 'dns', why does it matter and how does it work? — technology — the guardian." https://www.theguardian.com/technology/2010/dec/03/dns-ip-ddos-explained. (Accessed on 04/04/2017).

[33] A. Lai, "Step 1: How does cloudflare work? cloudflare support." https://support.cloudflare.com/hc/en-us/articles/205177068-Step-1-How-does-Cloudflare-work-, March 2017. (Accessed on 03/28/2017).

[34] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean ghtorrent: Github data on demand," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 384–387, ACM, 2014.

[35] "Case studies & customer success - amazon web services." https://aws.amazon.com/solutions/case-studies/all/. (Accessed on 03/22/2017).

[36] P. Mutton, "Digitalocean becomes the second largest hosting company in the world — netcraft." https://news.netcraft.com/archives/2015/05/01/digitalocean-becomes-the-second-largest-hosting-company-in-the-world.html, March 2015. (Accessed on 03/28/2017).

[37] N. Harteau, "Announcing spotify infrastructures googley future — news." https://news.spotify.com/uk/2016/02/23/announcing-spotify-infrastructures-googley-future/, February 2016. (Accessed on 03/28/2017).

[38] "Heroku." https://heroku.com. (Accessed on 03/22/2017).

[39] "Heroku security — heroku." https://www.heroku.com/policy/security. (Accessed on 03/31/2017).

[40] "Industry leaders choose circleci - circleci." https://circleci.com/customers/. (Accessed on 03/22/2017).

[41] "Who is using jenkins? - jenkins - jenkins wiki." https://wiki.jenkins-ci.org/pages/viewpage.action?pageId=58001258, 2016 June. (Accessed on 03/22/2017).

[42] T. Dierks, "The transport layer security (tls) protocol version 1.2," 2008.

[43] Z. Ait Bahajji and G. Illyes, "Google online security blog," *HTTPS as a Ranking Signal*, 2014.

[44] "Secure socket layer (ssl)/tls — cloudflare." https://www.cloudflare.com/ssl/. (Accessed on 03/28/2017).

[45] T. Makino, C. Jung, and D. Phan, "Finding more mobile-friendly search results," *Google Webmaster Central Blog*, 2015.

[46] E. Marcotte, *Responsive Web Design*. Book Apart, A Book Apart, 2011.

[47] "Bootstrap the world's most popular mobile-first and responsive front-end framework.." http://getbootstrap.com/. (Accessed on 04/04/2017).

[48] N. Lanxon, "Internet explorer 8 released today: Extensive hands-on report - cnet." https://www.cnet.com/uk/news/internet-explorer-8-released-today-extensive-hands-on-report/, March 2009. (Accessed on 03/29/2017).

[49] "Getting started — less.js." http://lesscss.org/. (Accessed on 03/22/2017).

[50] D. Odell, "Build tools and automation," in *Pro JavaScript Development*, pp. 391–422, Springer, 2014.

[51] "framework/composer.json at 5.4 laravel/framework." https://github.com/laravel/framework/blob/5.4/composer.json. (Accessed on 04/03/2017).

[52] "Design patterns mvc pattern." https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm. (Accessed on 03/29/2017).

[53] "Phpunit the php testing framework." https://phpunit.de/. (Accessed on 03/29/2017).

[54] M. Olan, "Unit testing: test early, test often," *Journal of Computing Sciences in Colleges*, vol. 19, no. 2, pp. 319–328, 2003.

[55] "Mockery mockery docs 1.0-alpha documentation." http://docs.mockery.io/en/latest/. (Accessed on 03/27/2017).

[56] "What are unit testing, integration testing and functional testing? — codeutopia." https://codeutopia.net/blog/2015/04/11/what-are-unit-testing-integration-testing-and-functional-testing/. (Accessed on 03/29/2017).

[57] "Mocking - laravel - the php framework for web artisans." https://laravel.com/docs/5.3/mocking. (Accessed on 04/03/2017).

[58] "Application testing - laravel - the php framework for web artisans." https://laravel.com/docs/5.3/application-testing. (Accessed on 03/29/2017).

[59] I. Dunlap, *Open Source Database Driven Web Development: A Guide for Information Professionals*. Chandos Information Professional Series, Elsevier Science, 2006.

[60] A. Hern, "Talktalk hit with record 400k fine over cyber-attack," *The Guardian*, 2017.

[61] N. Provos and D. Mazieres, "A future-adaptable password scheme.," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 81–91, 1999.

[62] J. Lockhart, *Modern PHP: New Features and Good Practices*. O'Reilly Media, 2015.

[63] "Sketch - professional digital design for mac." `https://www.sketchapp.com/`. (Accessed on 03/27/2017).

[64] "Database: Query builder - laravel - the php framework for web artisans." `https://laravel.com/docs/5.4/queries`. (Accessed on 04/03/2017).

[65] "Php: Pdo - manual." `http://php.net/manual/en/book.pdo.php`. (Accessed on 04/03/2017).

[66] "Javascript bootstrap." `http://getbootstrap.com/javascript/#tooltips`. (Accessed on 03/28/2017).

[67] J. Sweat, *PHP Architect's Guide to PHP Design Patterns*. Marco Tabini & Associates, 2005.

[68] "In unix, what is a daemon?." `https://kb.iu.edu/d/aiau`. (Accessed on 04/04/2017).

[69] P. Hope and B. Walther, *Web Security Testing Cookbook*. Cookbook Series, O'Reilly Media, 2009.

[70] "Javascript markdown editor - simplemde." `https://simplemde.com/`. (Accessed on 03/27/2017).

[71] "haleks/laravel-markdown: Laravel markdown integrate markdown "curly" braces inside the blade template engine, adding view extension and also giving the possibility of extending commonmark.." `https://github.com/haleks/laravel-markdown`. (Accessed on 03/25/2017).

[72] "Mobile-friendly test - google search console." `https://search.google.com/search-console/mobile-friendly`. (Accessed on 03/24/2017).

[73] "About us - yourtaximeter — uk taxi fare calculator — taxi numbers." `https://yourtaximeter.com/about`. (Accessed on 03/28/2017).

[74] "Github glossary - user documentation." `https://help.github.com/articles/github-glossary/`. (Accessed on 04/04/2017).