Final report

Improving the Plugify Admin Panel

Authors
B.A.P. de Jonge
M.J.J. Oudshoorn
M. van de Ruit

Final report

Improving the Plugify Admin Panel

Authors
B.A.P. de Jonge
M.J.J. Oudshoorn
M. van de Ruit

Coordinator:	O. Visser,	TU Delft
Coaches:	Assist. Prof. dr. ir. G. Gousios,	TU Delft
	Ir. J Hejderup,	TU Delft
Client:	Ir. K. Grigorjancs,	Plugify

TU Delft

Preface

This report is written by Bart de Jonge, Mark van de Ruit and Menno Oudshoorn for the bachelor project of the Computer Science program at Delft University of Technology. For the past 10 weeks, we have worked on improving the administration panel and customer service workflow of Plugify.

We would like to thank everyone at Plugify for their enthusiasm and support, with a special thanks to Karens Grigorjancs, our official Plugify supervisor and daily source of help, Peter Bosma, for his input, feedback and thoughts on the systems we built, Harriet Greve, for testing our application and providing us with useful feedback, and Oscar Westerhof for his assistance in building the new Plugify dashboard.

We would also like to thank our TU supervisors, Georgios Gousios and Joseph Hejderup, for their continued support and feedback throughout the project.

Delft, June 2017

Summary

In this report, we present our work on improving the administration software and automating parts of the customer service work for Plugify, a young Dutch startup that provides an online music act booking platform. We first analyze Plugify's needs, and then discuss how we design, develop, and iterate the systems they require. We leverage and customize pre-existing administration software named ForestAdmin to provide us with a stable administration panel to create and modify data. Furthermore, we rework Plugify's existing internal notification system to be smarter. Then, we create a small number of custom web pages for Plugify's customer service to more easily process relevant data. Finally, we provide Plugify with a system for metrics in an intelligent way. We also discuss how we perform unit, integration and usability testing, to ensure, among others, the stability, reliability, usability, performance and completeness of the systems we have built.

Contents

1 Introduction **1**

2 Research **3**

2.1 Overview . . . 3
2.2 Problem definition and analysis . . . 3
2.3 Project goals . . . 4
2.4 Requirements analysis . . . 5
2.4.1 User interviews . . . 5
2.4.2 Current system analysis . . . 6
2.5 Requirements definition . . . 8
2.5.1 Functional requirements definition . . . 8
2.5.2 Non-functional requirements definition . . . 10
2.6 Approach . . . 11
2.6.1 Process . . . 11
2.6.2 Tools and techniques . . . 12
2.7 Research conclusion . . . 13

3 Background knowledge **15**

3.1 Plugify background . . . 15
3.1.1 The Plugify funnel . . . 15
3.2 ForestAdmin background . . . 16
3.2.1 Default ForestAdmin features . . . 16
3.2.2 Extending ForestAdmin with custom functionality . . . 17

4 Design **19**

4.1 Architecture design . . . 19
4.1.1 Back end architecture . . . 19
4.1.2 ForestAdmin architecture . . . 20
4.2 Feed design . . . 21
4.2.1 Base feed initial design . . . 21
4.2.2 Messages feed considerations . . . 22
4.2.3 Notifications feed considerations . . . 23
4.2.4 Request feed design considerations . . . 23
4.3 Forest team design . . . 23

5 Implementation **25**

5.1 Smart actions . . . 25
5.2 Dashboard and metrics . . . 26
5.2.1 Fetching and processing the data . . . 26
5.2.2 Layout and excel output . . . 27
5.2.3 Speed improvements . . . 28
5.2.4 Predictions . . . 28
5.2.5 Graphs . . . 28
5.3 Notification logic . . . 28
5.3.1 Notifications for new users and artists . . . 29
5.3.2 Notifications for requests that don't seem to be converting to a booking . . . 29
5.3.3 Miscellaneous notifications . . . 30
5.3.4 Properties and actions . . . 30
5.3.5 Implementation . . . 31

5.4 Feeds . . . 31
5.4.1 Limitations and similarities . . . 31
5.4.2 Initial layouts . . . 32
5.5 Other features . . . 33
5.5.1 CRUD operations . . . 33
5.5.2 Settings . . . 34
5.5.3 Warnings . . . 34

6 SIG Feedback 35
6.1 Issues for delivering the code . . . 35
6.2 First SIG feedback . . . 35
6.3 Processing the SIG feedback . . . 36

7 Testing and QA 37
7.1 Unit testing . . . 37
7.1.1 Model testing . . . 37
7.1.2 Controller testing . . . 38
7.1.3 Forest collection testing . . . 39
7.2 Manual testing . . . 39
7.3 Usability testing with RITE . . . 40
7.3.1 Rules . . . 40
7.3.2 Participants . . . 40
7.3.3 Measurements . . . 41
7.3.4 Procedures . . . 41
7.3.5 Test results . . . 41
7.4 Usability testing through measurements . . . 43

8 Final Product Evaluation 45
8.1 Functional requirements assessment . . . 45
8.1.1 Dropped or modified functional requirements . . . 45
8.1.2 Missed functional requirements . . . 46
8.2 Project goals assessment . . . 46
8.3 Ethical implications . . . 48

9 Conclusion 49

10 Discussion and recommendations 51
10.1 Improvements and additions . . . 51
10.2 Regarding ForestAdmin . . . 51

A UML diagrams 53

B Interview questions 57

C Project description 59
C.1 What do we do? . . . 59
C.2 What will you do? . . . 59
C.3 Which technologies do we use . . . 59
C.4 Company description . . . 60

Bibliography 61

1 Introduction

Plugify [4] is a Dutch start-up that provides an online platform where users can book all kinds of live music acts. As a user, you can make a request for an artist, after which artists and users can have a chat, come to an agreement, receive an invoice and make a booking. The whole process needs to be monitored by the customer service of Plugify, so they can intervene when needed, for example when an artist does not respond to requests fast enough.

For this, Plugify has an administration panel (AP), which has a notification system that notifies the customer service of issues that may arise among users and artists. However, the existing AP is outdated and does not contain all the functionality that Plugify desires. Therefore, the project's aim is to improve the current administration panel. Following from this, the main goals are to improve on existing features such as the notification system, and to extend the panel with new features. Another goal of the project is to improve the reliability and maintainability of the software.

This report is structured as follows: chapter 2 contains the research report, in which the problems Plugify is running into, and possible solutions, are analyzed. Chapter 3 provides the necessary background information. Chapter 4 discusses design choices that were made during the project, and chapter 5 shows the implementation of the different features. Chapter 6 is devoted to analyzing the feedback from the Software Improvement Group (SIG), and how the issues in the code that they pointed out were handled. In chapter 7 we discuss how the system was tested, and based on this we evaluate the final product in chapter 8. We conclude in chapter 9, and discuss and give recommendations in chapter 10.

2 Research

2.1. Overview

The first two weeks of this project are dedicated entirely to the research phase, in which we formally define and discuss the details of what this project encompasses. This chapter summarizes this phase. In section 2.2, we first define and analyze the problem presented to us by Plugify. In section 2.3, we separate the key goals of this project. Afterwards, we analyze the current system and its shortcomings in section 2.4 and write down the requirements that follow in section 2.5. section 2.6 highlights the approach we want to take to solve the problem succesfully.

2.2. Problem definition and analysis

The current administration panel was designed with Plugify's needs and requirements at the time kept in mind. As Plugify is an active platform with a rapidly growing customer-base, however, its requirements for data-management are rapidly shifting.

Currently, Plugify is heavily occupied with managing its community. Plugify's Customer Happiness Officers (CHOs), for example, are responsible for maintaining day-to-day contact with users of the platform, and resolving issues that occur. They make phone calls, assist users and artists, monitor communications, manage customer data, and generally ensure that Plugify's community remains healthy and satisfied. The AP has a notifications feed, which CHOs manage, processing each notification individually. Notifications are triggered by a number of different events, such as users communicating, or new bookings being made.

As Plugify's customer base is growing so much, it is vital that the CHOs keep working at an efficient pace. According to Plugify, however, their work is becoming quite a cumbersome process, due to the panel. The panel has a sometimes unintuitive, overly manual and sluggish design, is missing critical usability features, is riddled with minute bugs, and performs slowly.

Furthermore, Plugify has managed to collect large sums of data over time, and is trying to extract useful information from said data, mostly by hand. It has a dire need for automating these processes and making the extracted information more accessible. This information can help Plugify optimize and improve their products.

Plugify feels that the administration panel, in its current form, no longer serves its needs and requirements as well as it could, making an update desirable. From this, we identify a number of issues with the panel:

- The panel is missing critical features that Plugify needs.
- The panel often presents information unclearly, due to a sometimes counter-intuitive user interface.
- The panel is riddled with small software bugs.

- The panel performs slowly, due to long average loading times.
- There are tasks that have to be performed manually in the panel that could be automated.

2.3. Project goals

From the problems that were previously defined, we can derive a number of goals for the project. These are defined with solving the aforementioned problems kept in mind.

These goals are important, and must be considered throughout the entire project. To support this, we use the Goal-Question-Metric (GQM) approach [1], which allows us to measure progress towards achieving these goals. This means that, for every conceptual goal we define, we also define a set of accompanying questions to assess said goal, and a set of metrics to help answer these questions in a measurable way.

The derived goals, questions and metrics are displayed below:

- Feature-completeness
  - Goal: Ensure the system is operational for the next phase in the startup, serving Plugify's needs.
  - Questions: Are all features Plugify's members need present in the system? Are any features Plugify's members need missing?
  - Metrics: Proportion of missing features vs implemented features.
- Usability
  - Goal: Improve the usability of the system for its users.
  - Questions: Is the performance and efficiency of CHOs and other users improving? Is the satisfiability of CHOs and other users with the systems improving?
  - Metrics: Time taken per task user has to complete, number of clicks taken per task user has to complete, and (subjectively) general satisfiability of users with the system.
- Maintainability
  - Goal: Improve the maintainability of the system for developers.
  - Questions: Is the system maintainable? Is it well structured? Does it follow common software design principles?
  - Metrics: Percentage of code duplication, average cyclomatic complexity, maximum cyclomatic complexity, test coverage, and (subjectively) number of other code smells.
- Reliability
  - Goal: Improve the reliability of the system for its users.
  - Questions: Is the system reliable? How are bugs and failure scenarios tackled? Is the system well tested?
  - Metrics: Test coverage, number of encountered errors while using the product in a production environment
- Performance
  - Goal: Improve the runtime performance of the system.
  - Questions: How performant is the system? How fast does it render content for CHOs? What is the performance during stress and high load?
  - Metrics: loading times of system, loading times of views that require large amounts of data.

2.4. Requirements analysis

It is vital that we derive requirements allowing us to achieve the project goals efficiently. Hence, we first perform a number of interviews with actual users of the administration panel, to get more detailed information on previously identified issues. Afterwards, we perform an analysis of the current system, to get a detailed overview of how it is assembled and what the shortcomings are.

2.4.1. User interviews

In this section, we perform a semi-structured interview [10], in order to identify shortcomings of the current product and better tailor to the needs of product users. This implies that we define a set of questions covering topics we wish to explore, but allow actual questions and answers to divert during the interview, if they elaborate more on the topics. As we interview CHOs and other users of the administration panel, the interview structure is concerned almost entirely with their work and actual use of the software. For CHOs, this is currently mostly the processing of notifications, so we dedicate a large part of the interview to this topic. The actual structure is defined in Appendix B.

The interviews loosely followed the described structure, and were performed with two users of the AP. These users are a CHO, who uses the panel daily, and an administrator, who uses the panel from a business-perspective. As the interviews followed the defined structure only loosely, we simply provide the points identified during the interviews, even if these do not answer any of the above questions, as long as they are concerned with the topics covered.

CHO Interview

The following issues, remarks and suggestions were identified by the CHO during the interview:

CH1: It is difficult to distinguish processed notifications from unprocessed notifications.

CH2: Snoozing or rejecting a notification could be easier and faster.

CH3: Some notifications, requests specifically, are always snoozed. This can be done automatically.

CH4: Views in the AP need to be constantly refreshed to update information.

CH5: Notifications page displays very little information about the notification in question, making notifications hard to identify at times.

CH6: Checking whether a user's whose request was closed, had any other pending requests, is rather devious.

CH7: Loss reasons for clients are rarely filled in.

CH8: Notifications for communication between an artist and a client after they have already made a booking, may be unnecessary.

CH9: The AP redirects users a lot. After one notification, six tabs may be open in the browser.

CH10: Many fields the user has to enter for notifications, should have default values, as they are almost always the same.

Administrator interview

The following issues, remarks and suggestions were identified by the administrator during the interview:

A1: There are many options per message: can be reduced to accepting, rejecting, marking as request, snoozing, and automatically warning an artist.

A2: The AP is missing the ability to group together requests that belong together.

A3: The AP is missing the ability to mark requests as unique, or not grouped together.

A4: Notifications can be heavily simplified. There are too many of some kinds, and too few of others.

A5: There should be notifications for detected upsell possibilities, such as when a booking is completed.

A6: There should be notifications for detected long-term inactivity, from users and artists.

A7: There should be notifications for detected potential loss of a client, such as when all of a user's requests are rejected.

The points raised are all considered when defining the requirements in section 2.5.

2.4.2. Current system analysis

In this section a general overview of the existing administration panel is given. First, we will look at the structure of the code and its different compartments. Next, we will analyze the current performance and usability of the system. Finally, the reliability and maintainability of the system will be looked into, by means of analyzing the test coverage and code quality. The goal of this analysis is to provide a broad overview of how different parts of the system work together, as well as to help define non-functional requirements and provide an easy way to confirm improvements in different areas. The UML diagram belonging to this section can be found in Appendix A.

Code structure

The overall class diagram of the current system can be seen in figure A.1. The system uses an Angular 2 front end and a Ruby on Rails 5.0 back end. The different Angular components perform API calls that are handled by the responsible Ruby controller. The controllers fetch necessary data from the PostgreSQL database, and return it as JSON to the Angular components. For readability, not all components, controllers, and relations are shown, but this should provide a good overview of how the different parts work together.

Performance and usability analysis

We will now analyze the speed and ease of performing some actions that are often done in the existing administration panel. These are

- Checking if a new request is unique, and otherwise deciding what other unique request it belongs to
- Handling a 'new message' notification
- Handling a '1 open or all rejected' notification

We will check the following things

- The amount of clicks needed to perform the action
- The total (average) time waiting for pages to load
- A (subjective) judgement whether the action is intuitive to perform

Checking if a new request is unique, and otherwise deciding what other unique request it belongs to
The sequence of actions to do this is as follows:

1. Click 'new request' notification on notifications page
2. Click the name of the user to go to the user page. This page opens in a new tab

3. Check the other requests of this user. Decide if this request is unique or not

4. Go back to notification tab. Write judgement in text box.

5. Set snooze date (this takes two clicks most of the time, sometimes more if the snooze date lies far ahead).

6. Click 'save and snooze'

7. Close user tab

8. Go back to notifications overview

This action requires 9 clicks. The total average time spent waiting for loading pages is 22.9 seconds. The action is quite unintuitive, as it requires the user to go the the page of the user that made the request. It would be better if the other requests of that user were already shown on the notification page.

Handling a 'new message' notification
The sequence of actions to do this is as follows:

1. Click 'new message' notification on notifications page. This gets you to the corresponding request page

2. Scroll down to read the message. Decide if you want to accept or reject the message

3. Click corresponding button.

4. Go back to notifications screen.

This action requires 3 clicks. The total average time spent waiting for loading pages is 17.3 seconds. A flaw that was found when performing this action was the fact that the message the notification was about is not highlighted. This makes it difficult to see what message is the relevant one sometimes. Also, it would be better if the messages were shown before other info to avoid having to scroll down all the way. It is currently not possible to set a snooze time when rejecting a message.

Handling a '1 open or all rejected' notification
The sequence of actions to do this is as follows:

1. Click '1 open or all rejected' notification on notifications page.

2. Click name of user to get to user page, so you can see other notifications and/or requests of this user.

3. Possibly click on other requests. We assume checking 1 other request here.

4. Click on notification tab, fill in text field. Choose snooze date, and click 'save and snooze'.

5. Close other tabs that were opened in the process

This action requires 11 clicks. The total average time spent waiting for loading pages is 23.1 seconds. Just like the first action, this action is unintuitive because the requests of a user are not shown on the notification page. It would be better if these are shown on the notification page.

Software engineering analysis
We will now analyze the reliability and maintainability of the system. This is done by looking at test coverage and static code analysis. The results are summarized below

- **Test coverage:** There are no tests at all written for the front end. The back end has a few tests, but the coverage for all AP-specific code is 13.2%. We therefore find this part of the code to be tested quite poorly, and not up to par with the back end code as a whole, which has a test coverage of 80%.
- **Code duplication:** We ran Simian, a popular and versatile code-duplication detection tool, on the front-end code. By default, it marks any duplications of at least six lines as actual duplications. With these settings, it detects 1714 duplicate lines of code, spanning 150 instances of actual duplications. Most of these concern instances of HTML files that are remarkably similar.
- **Front end linter:** We ran the Angular2 linter *ts-lint* on the front end code. 40 warnings were generated, however, they all complained about exceeding the maximum line length of 140 characters. There were no other warning types. This shows that the front end code is written according to the standards of the language.
- **Back end linter:** We ran the Ruby linter *Rubocop* on the back end code. In the parts of the code that are specifically written for the AP, there were 50 offenses of varying types. This shows that there is some improvement possible in static code quality.
- **Rubycritic:** Rubycritic is a Ruby static analysis gem. It assigns different ratings to each file, while also indicating code smells, duplications and complexity. The rating for AP-specific code is about a C on average, which shows improvement can be made to the static quality of the code. There are an average of 13 code smells and 42 duplications per file, and the average complexity is 106. This shows that there is quite some room for improvement in the static code quality.

2.5. Requirements definition

Based on the user interviews and system analysis performed in **??**, we now specify the requirements for the system. We divide these into functional and non-functional requirements.

2.5.1. Functional requirements definition

Functional requirements are requirements that describe what the system should do. The functional requirements are derived from the interview remarks in subsection 2.4.1. We divide these into four parts: existing functionality, notifications, custom actions and the dashboard. Some basic existing functionality should continue to exist. The notifications section covers all kinds of feeds and triggers to manage different aspects, such as artists, users, requests, and bookings. It also contains some automation and data collection. The custom actions section contains special actions that are not necessarily part of a standard CRUD operation, such as combining two user entries because they are actually the same user that registered in two different ways. The dashboard is for data visualization, to provide more insight in how the company is doing.
The functional requirements are as follows:

1. Existing functionality
   (a) It should be possible to perform standard operations such as create, update, delete, on data such as users, artists, and requests
   (b) Access to the administration panel should be restricted to Plugify employees only.
   (c) Applicable pages should have a search, filter and sort functionality, to provide easy lookup of data.
   (d) It should be possible to assign an admin to a conversation/notification/request. It should be possible to show only notifications of a certain admin.

2. Notifications
   (a) A *message feed* should exist.
      - The message feed should contain all messages and conversations that are not associated with a booking yet. These consists of both messages with and without a request
      - It should be possible to accept a message with a single button click so that it will not appear as a new message again.
      - It should be possible to label a message as a request. This should only be possible when a request does not exist yet
      - It should be possible to reject or revoke a request.
      - It should be possible to snooze the conversation for a certain amount of time, so that the user will be reminded again at that time. This snooze may be interrupted by certain triggers, such as the creation of a booking.

   (b) A *new request feed* should exist.
      - This feed should contain all requests that are possibly unique.
      - It should be possible to match a request to another unique request, if the request belongs to this unique request
      - It should be possible to label a request as a unique request.

   (c) A *notification feed* exists
      - This feed lists all notifications that are not yet handled.
      - The necessary actions differ per notification. The different notification types are discussed below.

   (d) Different notifications should be generated
      i. New customer notification
         - This notification should trigger when a customer enters his or her phone number.
         - It should be possible to add customer information that is retrieved when calling the customer, such as the type of customer and how the customer found the platform
         - It should be possible to signal that there are no upsell possibilities for this customer.

      ii. New artist notification
         - This notification should trigger when an artist receives his or her first request
         - This notification should trigger when an artist receives a request and it has been three months since someone from Plugify last spoke to this artist is nu 'since artist got a request'
         - It should be possible to indicate when this artist was last spoken to, and what was discussed.

      iii. Possible deletion notification
         - This notification should trigger when an artist has received two warnings for leakage; trying to book around Plugify's back.
         - This notification should trigger when there have been three other warnings, such as
           – Not responding within 24 hours
           – Mentioning travel costs or VAT
           – Mentioning the Plugify service fee
           – Negative comments about Plugify
         - The given warnings need to be kept track of and displayed
         - It should be possible to delete a warning.

        iv. Potential loss notification
            - Triggers when the last request is revoked
            - It should be possible to indicate why this loss occurred
            - It should be possible to indicate which other artist was booked on what platform, if applicable
        v. Fear of missing out notification
            - Triggers when only a single request is open
            - Triggers when the last request is revoked, and nothing has been done with automatically sent suggestions
            - It should be possible to indicate what action was taken
            - It should be possible to set a snooze for such a notification
            - It should be able to assign a status, for example to not bother anymore.
        vi. New booking notification
            - Triggers when a new booking occurs and upsell was not turned off (see new customer notification).
            - It should be possible to indicate what other bands were recommended and when
            - It should be possible to snooze such a notification

3. Custom actions
    (a) It should be possible to perform certain custom actions, such as, but not limited to:
        - Merging two user entries into one if they are actually the same user
        - Sending a message in a conversation in the name of the user or artist
        - Rejecting or revoking a request
        - Rejecting a quotation
    (b) These actions need to be intuitive to perform
    (c) These actions need to be available at the correct place
4. Dashboard
    (a) The dashboard should contain several graphs, concerning, but not limited to:
        - Number of unique requests
        - Percentage of unique requests that are booked/quoted/still open
        - Total revenue (per booking)
        - Number of new users/artists
    (b) It should be possible to adjust the start and end point of the graph
    (c) It should be possible to adjust the interval of measurements in the graph
    (d) It should be possible to filter results, for example filtering new artists on only DJs

2.5.2. Non-functional requirements definition

Non-functional requirements specify how the system works, and serve as quality attributes for the system. Some of these requirements are derived from the interviews and system analysis, others are a result of best practices that we learned during our bachelor studies. The non-functional requirements are as follows:

1. The system should comply with Plugify's current software system and data models.
2. The system should be maintainable, meaning that it has well written, well documented code that follows common programming standards. Therefore
    - The number of code smells should be as low as possible, with a maximum of two per file

- The amount of code duplication should be as low as possible

3. The system should be reliable, and therefore well-tested. Therefore
    - The existing tests should keep passing
    - New unit-tests will be created for new features in the back end
    - The back end test coverage must be at least 80% so it complies with the test coverage in the existing back end
    - The front end will be tested by user testing, through RITE. Front end logic should also be tested well.
4. The system should be fast, meaning it has short loading times, and responds quickly even under heavy load
5. The system should be secure and accessible only by those granted access.
6. The system should have an easy to use interface that can be understood by employees with a non-technical background.

2.6. Approach

In this section we discuss our approach to this project. First, we discuss how we will ensure a reliable development process happens, so requirements are met, in subsection 2.6.1. Second, in subsection 2.6.2 we discuss the tools and techniques we will use in this project to make sure requirements are met and the process is as smooth as possible, and why we choose them specifically.

2.6.1. Process

A stable development process is vital for creating a good system rapidly. Therefore, we will adhere to a number of points, to ensure an effective development process happens.

- **Agile**. We will use agile development in the form of SCRUM, with a sprint duration of 1 week. A sprint plan will be made before every sprint, and after each sprint we will reflect on the planning and performance during that sprint. At the end of each sprint, a stable version of the system must exist. An agile development process provides room for early testing, and is adaptable to changing requirements. The latter allows us to tweak requirements to ensure the product we deliver perfectly fits the needs of Plugify's employees.
- **Continuous integration**. To safeguard the reliability of our system, we will use continuous integration. This means we will instantly notice if some tests fail, resulting in those tests being fixed shortly after It will not be possible to merge new code into the existing master branch if the continuous integration checks fail.
- **Working on location**. We will work in the office of Plugify four days a week. This entails good communication, which is vital in adhering to the needs of the company. It also provides easy user testing possibilities.
- **Version control and pull-based development**. A version control system is vital in writing collaborative software. Pull-based development ensures all new code is reviewed thoroughly before being added to the definite code-base. This means the overall software quality will be higher, and it is likely that less bugs will be present.
- **Rapid Iterative Testing (RITE)** [9]. To ensure that we meet the requirements, we will use RITE as soon as a sufficient working version has been established. This means we let our users test the product and have them engage in a verbal protocol. We then update the product as soon as possible, after which we do another testing round. This should improve the overall user experience.

2.6.2. Tools and techniques

Current System

As our project is an extension of the Plugify software package, we need to comply with their currently used techniques. The choice between extending the current AP or building a new one from scratch is explained below, so in this section we will focus on anything not related to the front-end. Since the entire back-end of the Plugify is built using Ruby on Rails 5.0 and Ruby 2.3 we will use that for the AP part of the back-end as well. This choice ensures consistency and improves maintainability. Plugify uses a PostgreSQL 9.5 database to store their data, which means we need to use that as well. For code sharing, version control, and code review Plugify is currently using GitHub. We are happy to use this for our project, since this would be our own choice as well due to our extensive experience with the platform, and the good code review features it offers on top of the Git framework. Finally, Travis CI is used for continuous integration. This ensures all tests will be run before code can be merged, meaning the master branch can always provide a working version of the software product.

Process Tools

We will use two tools to improve our development process, which is discussed in subsection 2.6.1. The first tool is Trello, a card-based tool to get insights in the different tasks of your agile sprint, therefore improving organization and productivity. Another similar alternative for Trello would be Waffle.io which has the extended functionality of integrating with GitHub. However, we will not use our own GitHub repository for this project, as most of the code will be an extension of Plugify's current back end. Therefore Waffle.io would clutter their GitHub, something that should be avoided.

The second tool is Slack, a messaging service that has many features to support project work. We have had good experiences with this tool in the past because it enables clear communication with everyone involved in the project, such as team members and supervisors. Another reason Slack is a good choice for communication is the fact that Plugify also uses Slack to discuss projects throughout the company, which means using Slack ourselves provides consistency for a smoother workflow.

Front end framework

The final decision explained in this section covers the front end of the AP software. This is arguably the most important choice, as the framework used for the back end is already established to comply with Plugify's system. A number of plausible options are present, which, together with their main strengths and weaknesses, are discussed below:

- Improving and extending the current front end, which was developed in Angular 2. This seems to be the best choice at first glance, as we would not have to start from scratch, and most of the basic features have been implemented. These advantages however, are countered by the fact that while the current AP implements most features, it suffers from some issues as discussed in subsection 2.4.2. The design lacks polish, pages take long to load and actions are unintuitive and take a lot of clicks. It would take up a hight portion of the available development time to fix these issues.

- Creating a new front end using a front end framework like Angular 2, React or Polymer. As we have experience developing front ends using such frameworks, particularly Polymer, this would allow us to generate a front end that perfectly matches the client's needs. However, generating a front end from scratch takes a large amount of development work, and would cause us to focus less on the automation and notification parts in the back end.

- Creating a new front end using Ruby on Rails. Creating a front end in Ruby on Rails, the same framework that is used for the back end, would allow us to develop a working version rapidly. However, styling such a front end appears to be challenging, among other issues because no one of our team has experience with developing front ends using Ruby on Rails. Another disadvantage, when compared to front end frameworks like Polymer or Angular2, is that Ruby on Rails does not by default support updating views without reloading the page.

- Setting up an existing AP front end framework such as ForestAdmin, BlurAdmin, or ActiveAdmin. Such a framework generates the standard data manipulation possibilities, and then allows for custom actions, views, and settings to be used while developing a user experience that fits your business. An advantage is that we can make a suitable user interface with little development time and focus on more complicated back end features such as automation and notification triggering. The logic of such a front end framework is already well tested. A disadvantage would be that it is less customizable than developing a front end from scratch, and it, in the case of ForestAdmin, is usually hosted by ForestAdmin itself [7], which introduces an unwarranted dependency on the software vendor.

After reviewing and discussing the options with our client, we conclude that using an existing AP framework is currently the most beneficial choice for the final product. The current AP suffers from a number of issues we would have to address before starting on any of the requested new features. The right framework, however, provides us with an intuitive interface, and addresses many of the original issues with the AP, allowing us to focus on new features such as automation. The question that remains is then which framework best fits our requirements. After analyzing the possibilities, we conclude that ForestAdmin and ActiveAdmin suit our needs, due to their reliable service and easy setup possibilities. ActiveAdmin has been around longer than ForestAdmin [2]. However, ForestAdmin is a rapidly growing service that constantly adds new and exciting features. Furthermore, we had talk with a member of ForestAdmin, to alleviate our concerns. Hence, we are convinced that ForestAdmin is a reliable service that suits our needs, and choose to use it during the remainder of the project.

2.7. Research conclusion

We can now conclude the research phase by giving an overview of our proposed solution to the problem. The administration panel of Plugify, in its current form, no longer serves the needs of the company. It is slow, has missing features and actions are often unintuitive to perform. Therefore, the goals of the project were to make a system that meets Plugify's new requirements and has a high usability, maintainability, reliability, and performance. Based on interviews with some of the future users of the system, and an analysis of the existing administration panel, we have derived a set of functional and non-functional requirements. To achieve the project goals, we will develop a new administration panel, using ForestAdmin as a front end framework, and extending the existing Ruby on Rails back end to suit the new requirements. As one of our goals was to achieve a high usability, we will use RITE for quick user evaluation, so we can find flaws in the design of the system quickly. A stable development process is ensured by using SCRUM and continuous integration. This also helps in achieving a high reliability in our system.

3 Background knowledge

This section provides the necessary background knowledge to fully understand the project and this report. It explains the basic funnel of Plugify, the different steps that are taken from a user doing a request, to a booking being made. It also gives a basic explanation of Forest Admin, the front-end framework that is used to build the new administration panel.

3.1. Plugify background

Plugify as a company is already explained in chapter 1, subsequently in this chapter we will go into further details on how Plugify operates, and what the general process is that preceeds a booking. Plugify has a small employee base (10-15 people) that is divided into several teams. One of the aims of this project is to improve the efficiency of the Customer Happiness Office (CHO) team. The main job of the CHO team is to make sure the Plugify funnel is operating smoothly, bookers and requesters understand the platform and each other, and that all actions taken on the platform comply with Plugify's Terms of Service (TOS) [11]. Hence, it is important to understand the different stages that the Plugify funnel is composed of.

3.1.1. The Plugify funnel

In Plugify's case, the funnel encloses the entire process between a user visiting the platform for the first time, a performance, and (hopefully) a review. The steps a user will encounter in this funnel, in the most common order, are:

- Search for an artist
- Sign up for the platform
- Create a request
- Discuss the details with the artist
- Receive a quotation from the artist
- Confirm the booking, and make the payment
- Listen to the artist
- Leave a review

As users are uncertain whether an artist is available, Plugify encourages them to create a request for multiple artists, to have the highest chance at a successful booking. As only one of these requests can result in a booking, Plugify marks them as a group with a single ”master-request”. This is useful for both the CHO team, as they know what requests are coupled, and for business intelligence, as looking at the number and price of master requests provides a valuable insight in the transaction value that is generated on the platform.

Because Plugify is growing, the CHO team is unable to go through every request manually every day to check that everything is still going smoothly. Therefore, as part of the previous Admin Panel, a notification system was implemented to provide updates to the CHO team when certain events occur, so they can check in to see if everything is alright. The triggers for funnel-specific notifications in the old notification system are:

- A user's phone number is updated - this allows the CHO team to welcome them to the platform and ask if they have feedback or need anything.
- A new request is created.
- A new message is added to a request's conversation - this provides possibilities to check if the request is still going smoothly, and whether the contents of the message are in line with Plugify's TOS. That last point is to detect and prevent leakage, which occurs when an artist tries to process a request outside of Plugify, so he/she does not have to pay the commission.
- A quotation is generated.
- All requests/quotations in a master request are rejected/withdrawn - this means a user was not able to find an artist, and he/she might need some help.
- A booking is created.

Even with this notifications system, the CHO team still has to perform many actions per request. Therefore a major goal of the new AP is to improve this system. An improvement that could be made is to generate notifications only when a master request is stuck, for example due to one party not responding. This means the data for the individual requests must be analyzed in greater detail, and in smarter ways.

3.2. ForestAdmin background

As discussed in chapter 2 we decided to use the ForestAdmin framework [7] to rapidly build an AP front end. This section will explain how ForestAdmin works, what it provides by default, and how it can be personalized. This is important as it shows what front end work we have and have not done ourselves.

ForestAdmin is divided into two parts:

1. The actual front end which is served by the ForestAdmin servers. This front end first fetches the table layouts, which are stored on one of the ForestAdmin servers as well, as is discussed below. The client then fetches the data from the Plugify servers.
2. An open source back end framework which crawls the data model and sets up the standard CRUD API endpoints, including getting lists of records, full details of one record, and updating a record. It also makes a copy of the data layout and sends it to the ForestAdmin server for improved loading times. This framework is available for multiple back end options like Express, Django and Ruby on Rails, however the latter is the one that is best supported. The default CRUD implementations already support Ruby on Rails validations, and display an error message when these fail.

3.2.1. Default ForestAdmin features

These two parts combined give ForestAdmin the default features of showing data, editing data, creating data and deleting data. By default all defined collections and their fields are shown, but irrelevant (or secured) collections or fields can be hidden using the layout editor in the ForestAdmin front end. ForestAdmin supports development and production environments by default, but more can be added manually. ForestAdmin also supports different teams, so certain collections can only be viewed and edited by certain people. Finally, ForestAdmin has a dashboard page on which you can show single value, percentage, pie, or time based charts of your data. Simple charts based on a direct field in your data can be added in the front end.

3.2.2. Extending ForestAdmin with custom functionality

As every company operates differently, ForestAdmin provides some ways to customize the AP. First of all, for fields, collections and charts, smart versions can be added. These smart versions are displayed identical to default ones in the front end, but their content can be determined dynamically in the back end. This means we can add aggregated fields, add charts that depend on multiple fields/data entries, and show custom collections that are not immediately available from the database. All of these improve the user interface and data visualization, and therefore increase the efficiency of the employees.

Secondly, ForestAdmin allows for the creation of so-called smart views. These smart views are custom views that are defined on a collection and built in EmberJS. They allow us to fully customize how interaction with the data is handled.

4 Design

This section provides a detailed overview of the design choices made to fulfill the requirements for this project. It details the code and architecture design process for the system in section 4.1, discusses the UX design process behind the set of *feeds* that are present in the AP in section 4.2, and finally brings to light the design choices behind the team layouts in ForestAdmin in section 4.3.

4.1. Architecture design

This section discusses the overall architectural design of our polyglot implementation of the new AP. Since our implementation spans both the Ruby on Rails back end and the ForestAdmin related implementations, they are discussed separately. A total overview of the system can be found in figure A.2. This overview shows how the ForestAdmin front end, Ruby on Rails back end, PostgreSQL database and Exact API communicate. For some parts of the system an example is given, rather than all components. For example, the communication between the front end artists page and the Forest::ArtistsController displays the general structure for all smart actions. The communication between the front end notification feed and a number of ForestLiana controllers display the fact that the front end feeds get data from different controllers (to fetch related data), and are also able to call smart actions. The actual interaction with the database is extracted in Ruby on Rails, via the use of ActiveRecord. Therefore the Rails back end as a whole communicates with the database, instead of every controller individually. The overview also shows that both the DashboardController and the MetricsController get data from the Exact API. However if one of them fetches the data, it is cached to improve loading times on both related pages.

4.1.1. Back end architecture

As Plugify already has a fully operational Ruby on Rails back end, and the code for the new AP should be merged into this project, we were not able to design the architecture from scratch. However, Ruby on Rails has detailed guidelines/best-practices for the architecture of an api application, and the existing Plugify back end follows these principles really well. Models and controllers are nicely separated to create a Model-View-Controller (MVC) model, with the front end website as view. Other than that, there are a number of jobs, to be run periodically or on a certain trigger, and helpers, for shared functionalities.

We tried to follow this pattern as best as we could, however some ForestAdmin specific code had to be placed in predefined locations for ForestAdmin to recognize it. This concerns the code for smart actions and fields, as well as CRUD overrides. To still follow the Ruby on Rails design patterns we divided that code up into models and controllers as well. All test code was separated following the same principles.

4.1.2. ForestAdmin architecture

The architecture of the ForestAdmin part of our implementation concerns both the ForestAdmin data flow, the ForestAdmin MVC model, and our own implemented custom smart views.

ForestAdmin data flow

The ForestAdmin data flow is not something we designed ourselves, but it is a key point for Plugify as it concerns the security of their data. As discussed in section 3.2 the ForestAdmin servers store a copy of the layout of the Plugify data model, which is generated by the ForestLiana Ruby gem. That data layout is fetched by the client after the general ForestAdmin front end has been loaded. After this the client fetches the data from the Plugify server, which runs on the Amazon Web Services (AWS) platform. This means only a copy of the data layout is ever stored on a ForestAdmin server, and the data is, secured with our own secret keys, strictly sent from the Plugify servers to the clients, never passing through the ForestAdmin server. Therefore ForestAdmin's general data flow satisfies Plugify's need for securing their data. A visualization of both the initialization process and general ForestAdmin architecture is given in figure 4.1

ForestAdmin MVC model

As discussed above, our custom ForestAdmin code has to be placed in predefined locations, which slightly hinders a perfect MVC model. However we managed to still divide the code into parts that make sense. First of all, all custom front end code, in the form of smart views, is separated as discussed in subsection 5.4.1. Secondly, all controllers that process smart actions, declare smart charts, or declare smart collections, are grouped together in the Forest namespace of the existing controller package. Controllers that overwrite existing ForestAdmin functionality, like overwriting the general CRUD operations for custom logic and checks on updates, are grouped into the controller section of the lib/forest_liana package. These controllers can not be moved to the general controller package as that would prevent forest_liana from recognizing the overwrites. Finally, the extensions of the ForestAdmin collections that declare smart fields and segments, are grouped together into the collections section of the lib/forest_liana package, again to make sure forest_liana recognizes the extensions.

The coupling between the ForestAdmin front end and our back end code happens through a url format specified by ForestAdmin. In config/routes.rb all of these routes are mapped to the corresponding controllers for smooth processing of smart actions and smart chart requests.

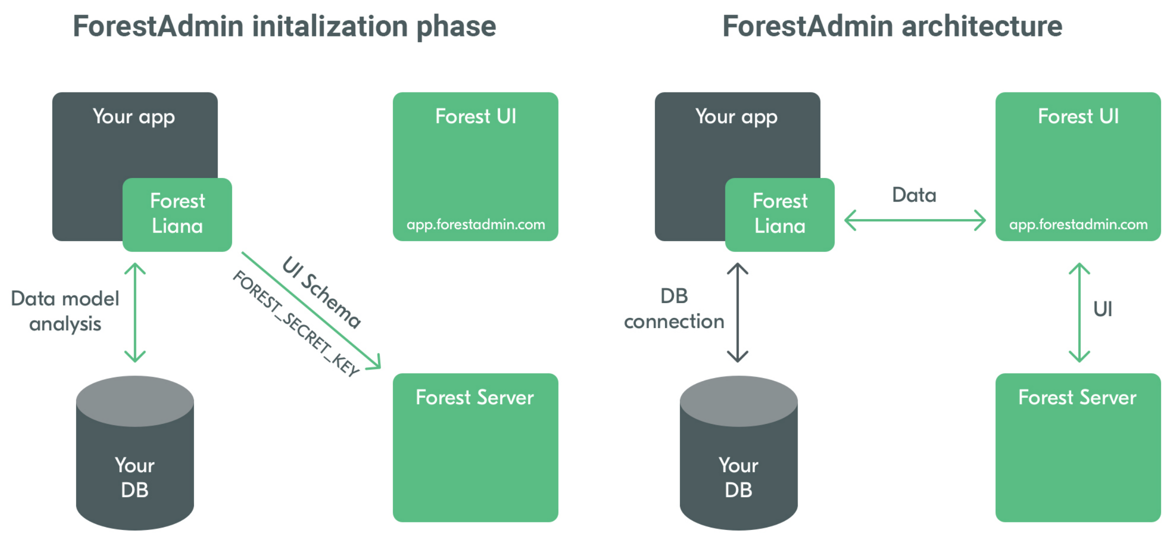


Figure 4.1: ForestAdmin initalization process and architecture
Source: http://doc.forestadmin.com/developers-guide/

4.2. Feed design

In this section, we detail the initial design for the three feeds, as defined in the functional requirements in chapter 2. We provide details and motivations behind basic design choices such as content layout and grouping, location of vital functionality, and the intentional creation of similarities between feeds.

For clarity, the three feeds are summarized below:

- A message feed, which allows users to process messages between Plugify's users, approving them, archiving them for a set time, or rejecting their associated requests.
- A new request feed which allows users to match separate incoming requests by one user on the same day together, or separate requests that were already matched, if necessary.
- A notification feed, which lists all as-of-yet unprocessed notifications thrown by the notification system, together with all the necessary attached information and the capability to modify that information.

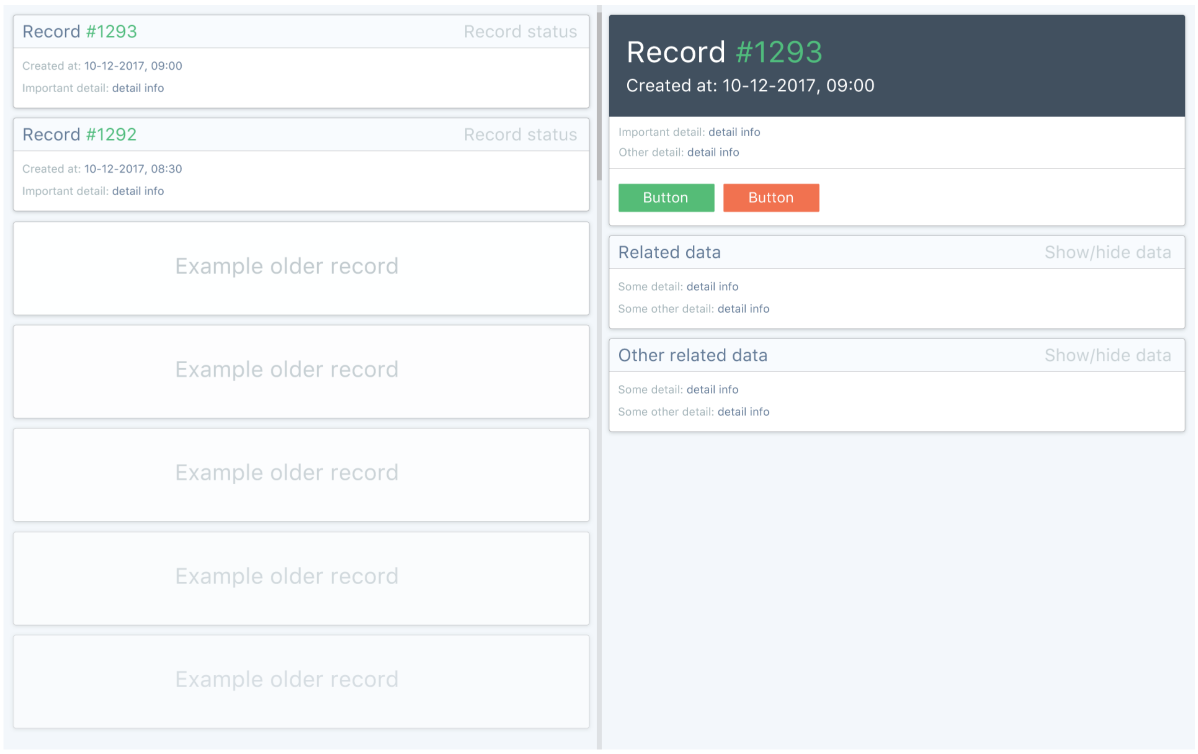
As the feeds clearly share core similarities, we provide an initial design for a *base feed* that fulfills shared layout requirements, and then define more specific requirements per feed with regards to content representation.

4.2.1. Base feed initial design

When we analyze the feeds' requirements, a set of similarities between the feeds is identified. For each feed, the following holds:

- The feed must display a number of records at the same time. (messages, requests, notifications)
- The records displayed most prominently by the feed must ordered in a pre-determined manner.
- For each record, related data must be shown. (Every message relates to a request, every request relates to a group of other requests, every notification has a supply of attached information that is necessary to process the notification.)
- For each record, a status must be shown. (Messages are approved, disapproved, and their requests rejected or nonexistent. Notifications can be closed, archived, et cetera.)

Starting from these similarities, we provide an initial layout design of a feed, depicted in figure 4.2, where content is structured into two major sections, each taking up half of the feed's space. To the left, a list of most recent records is displayed, each of which can be selected. Upon selection of a record, to the right, a list of details regarding this record is displayed. Details can include relevant information about the record, related data that may be of purpose to the user, as well as necessary actions the user can undertake that influence this record, such as modifying, adding, or deleting the record or its related data.



Record #1293 Record status

Created at: 10-12-2017, 09:00

Important detail: detail info

Record #1292 Record status

Created at: 10-12-2017, 08:30

Important detail: detail info

Example older record

Example older record

Example older record

Example older record

Example older record

Record #1293

Created at: 10-12-2017, 09:00

Important detail: detail info

Other detail: detail info

Button Button

Related data Show/hide data

Some detail: detail info

Some other detail: detail info

Other related data Show/hide data

Some detail: detail info

Some other detail: detail info

Figure 4.2: The basic feed layout

All three feeds will share this same layout design, as this allows us to more efficiently develop the feeds. However, as the feeds differ on certain points, we must take note of some considerations per feed.

4.2.2. Messages feed considerations

The messages feed shows new unapproved messages between artists and users, and allows an administrator to process them. In order to do so, the administrator requires direct access to relevant information. We identify key information that must be displayed in the feed:

- The message itself.
- Other messages in the conversation, to provide context.
- Information about the user, such as their contact information.
- Information about the artist, such as their contact information and whether they have received warnings in the past.

Furthermore, the administrator must be able to process messages quickly. We identify key actions the administrator must be able to undertake:

- Approving a message. After this, the message is processed and disappears from the feed.
- Marking a message as being a request.
- Marking a message as representing a withdrawal by a user, implying that they withdraw from the request they initially made for the artist. After this, the message is processed, and disappears from the feed.
- Marking a message as representing a rejection by an artist, implying that they reject the request a user initially made for them. After this, the message is processed and disappears from the feed.

- Adding a message to the message's conversation, which can be set as represented by an artist or a user.

4.2.3. Notifications feed considerations

The notifications feed shows all as-of-yet unclosed or unsnoozed notifications in order of the most recently created notification. Administrators must be able to rapidly process incoming notifications.

As every notification concerns a certain event, administrators must have access to key information regarding said event, which we identify below:

- The type of notification thrown.
- When the notification was thrown. This can impact the deadline for processing the notification.
- The assigned administrator for the notification.
- Relevant related information for the notification. This can include the following: a user, an artist, a request of number of requests, a conversation or number of conversations, a quotation, and a booking.

Furthermore, to process a notification, administrators must be able to perform a number of key actions, which we identify below:

- Closing a notification. After this, the notification is processed, and disappears from the feed.
- Snoozing a notification, for a configurable amount of days and hours. After this, the notification is processed (for a while), and disappears from the feed.
- Assigning an administrator to the notification. This can be done manually, if necessary.

4.2.4. Request feed design considerations

The request feed shows groups of user requests in most-recent order. If a user makes four requests on the same day, it will appear as a single group. As the feed's purpose is to look at groups of requests only, it will not show any groups consisting of just a single request. Next to these groups, administrators require access to the following information:

- The number of requests made inside a group.
- Per request, relevant information about the request, such it's date, time, duration and type of artist.

Furthermore, to monitor and adjust these request groups, administrators must be able to perform a number of key actions, which we identify below:

- Splitting off a request or number of requests from another group of requests as these are separate and count as multiple requests.
- Merging together two or more requests into a single group as these are related and count as one request.

4.3. Forest team design

As ForestAdmin supports teams, which allow us to customize the experience for sets of people, we can ensure that certain content is only visible to users with the proper rights, and can ensure that the right content is visible to the right user at a glance.

Plugify currently has at least two kinds of internal users: administrators and CHOs. We define teams for each of these. Administrators have, for one, complete access to all data, as well as the right to modify the team layouts. CHOs have limited access, as we hide data fields that are of no importance to them.

Furthermore, we define a third team, named All, to which every member of Plugify has direct access. For now, this team provides access to collected metrics only, as most of Plugify's members require access to this data, but not the rest of the data.

5 Implementation

In this section, we discuss the implementation of the system. We explain the implementation of the different components, implementation choices we considered and difficulties we encountered.

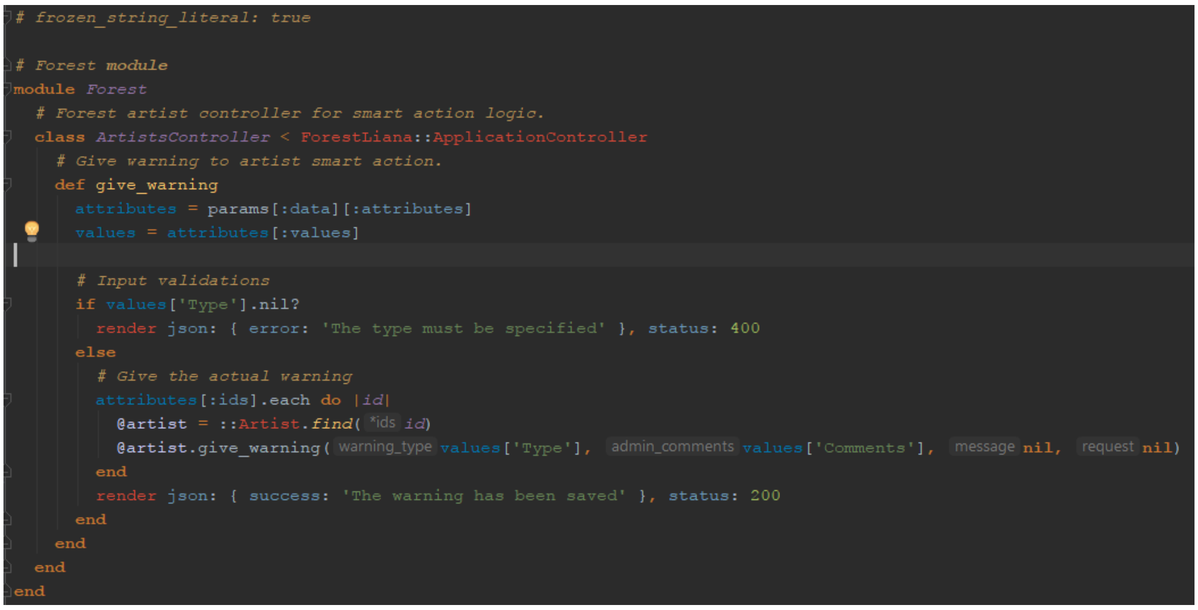
5.1. Smart actions

Since every business needs more logic than just creating, deleting and editing, ForestAdmin allows the definition of so-called 'smart actions' on collections. These smart actions are defined in the back end, and can include an input form when necessary. An example of an implementation of a smart action can be found in figure 5.1. We have implemented a number of smart actions to facilitate Plugify's needs:

- **Artist: give warning** - This action is used to give a warning to an artist, when he/she does not comply with Plugify's code of conduct. Examples are: talking negatively about Plugify or asking for traveling costs. The required fields in the input form are: the type of warning, and the reason this warning was given.
- **Conversation: this is a request** - As users of the platform can make contact with an artist by either creating a request or asking for more information, this is a way to mark these requests for more information as a actual request in the back end. This is then used for generating notifications and more accurate business intelligence data. The action requires some input for defining the date, occasion and location of the performance.
- **Conversation: send message** - This action gives CHO employees the possibility to send a message in a conversation. The required form inputs are: the body of the message, if the message must be sent as user or artist, and whether to email the recipient.
- **Message: reject request with message** - This actions marks a request as rejected, with the body of the message as reject reason. The form allow the initiator of the action to specify if the user and/or artist should receive an email notifying them of this rejection.
- **Notification: edit notification admin** - This action can be used to edit the admin of a notification. The form can be used to select an admin using a dropdown. In the back end logic, the admin of the corresponding user is also updated to keep the data consistent.
- **Quotation: reject quotation** - This action can be used to reject a quotation, and the corresponding request if specified in the input form. The form also allows specifying the reject reason and whether the user and/or artist should receive an email.
- **Request: reject request** - This action can be used to reject a request. The input form can be used to specify the reject reason, the rejection date (default today), if the request should be rejected as user or artist, and whether the user and/or artist should receive an email.

- **User: impersonate** - This is an action that is a duplication of an action in the old AP. It can be used to log into the platform as a specific user, to check what the see when they open the platform. This is very useful for testing purposes and helping with specific problems.

- **User: merge users** - This action is actually defined on two users, and is used when a user created two accounts by accident. The action merges the most important information of the second user into the first user, if it is not already present. After this the second user is deleted.



```ruby
# frozen_string_literal: true

# Forest module
module Forest
  # Forest artist controller for smart action logic.
  class ArtistsController < ForestLiana::ApplicationController
    # Give warning to artist smart action.
    def give_warning
      attributes = params[:data][:attributes]
      values = attributes[:values]

      # Input validations
      if values['Type'].nil?
        render json: { error: 'The type must be specified' }, status: 400
      else
        # Give the actual warning
        attributes[:ids].each do |id|
          @artist = ::Artist.find( *ids id)
          @artist.give_warning( warning_type values['Type'], admin_comments values['Comments'], message nil, request nil)
        end
        render json: { success: 'The warning has been saved' }, status: 200
      end
    end
  end
end
```

Figure 5.1: Example of a smart action. Gives a warning to an artist

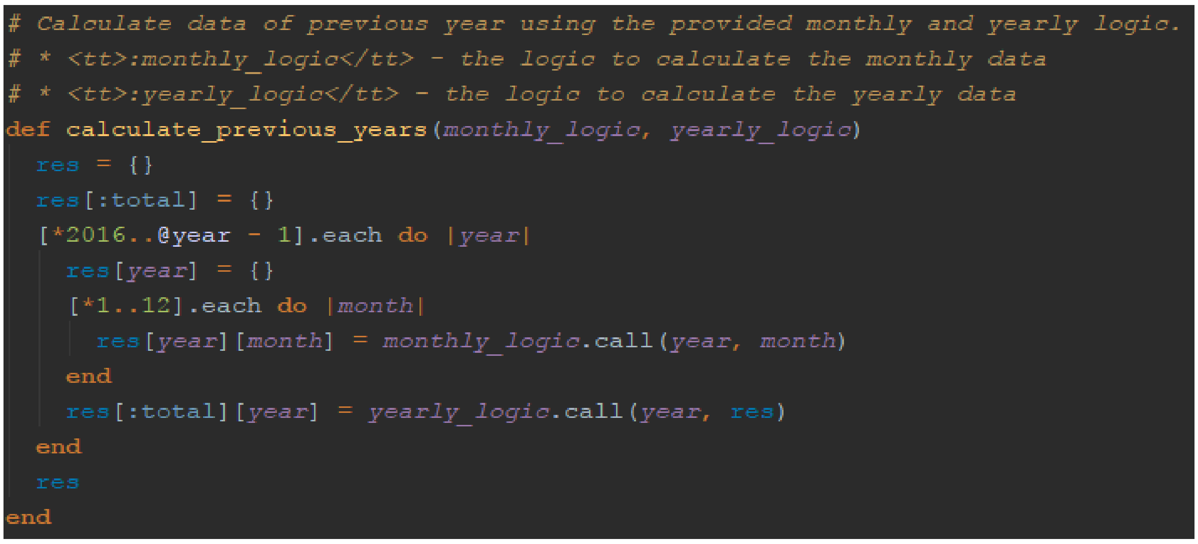
5.2. Dashboard and metrics

The metrics view and dashboard give a quick and easy overview of the performance and growth of Plugify and as a company. It displays both financial data as well as other metrics such as the number of requests and bookings over time. Predictions for the future are also being made.

5.2.1. Fetching and processing the data

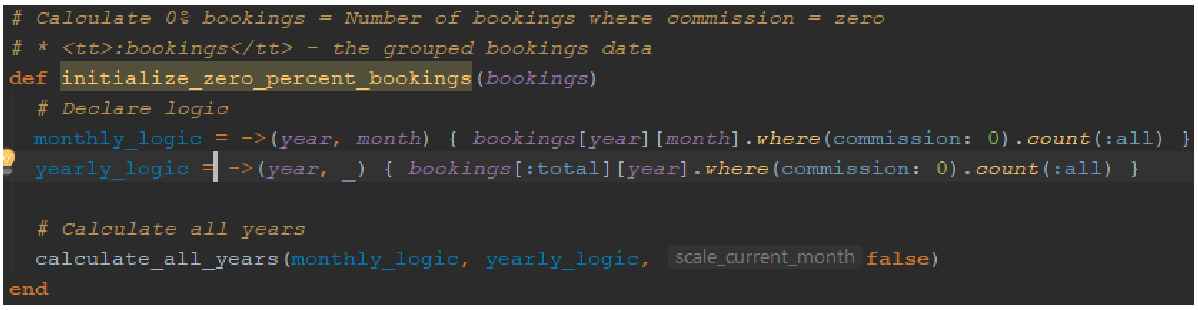
The necessary data to calculate the different metrics is stored in two separate places. Information about requests and bookings is in the Plugify database, and the financial data is in the Exact database. Exact is an accounting platform that provides a means to store and process financial data. [3]

We fetch the Exact data from the REST API, using the Elmas [5] gem. Data from the Plugify database is fetched by using regular SQL queries, aided with Rails ActiveRecord actions. We then use this data to calculate all the different metrics that are shown in the dashboard. This is done by sending the separate logic for each different metric in a lambda to a helper method that uses this logic, and returns the correct data format. Figure 5.2 shows an example of a helper method, figure 5.3 shows how this method is used to calculate a metric



```ruby
# Calculate data of previous year using the provided monthly and yearly logic.
# * <tt>:monthly_logic</tt> - the logic to calculate the monthly data
# * <tt>:yearly_logic</tt> - the logic to calculate the yearly data
def calculate_previous_years(monthly_logic, yearly_logic)
  res = {}
  res[:total] = {}
  [*2016..@year - 1].each do |year|
    res[year] = {}
    [*1..12].each do |month|
      res[year][month] = monthly_logic.call(year, month)
    end
    res[:total][year] = yearly_logic.call(year, res)
  end
  res
end
```

Figure 5.2: Helper method for the calculation of the metrics. Takes logic as an input, and uses it to calculate all different values



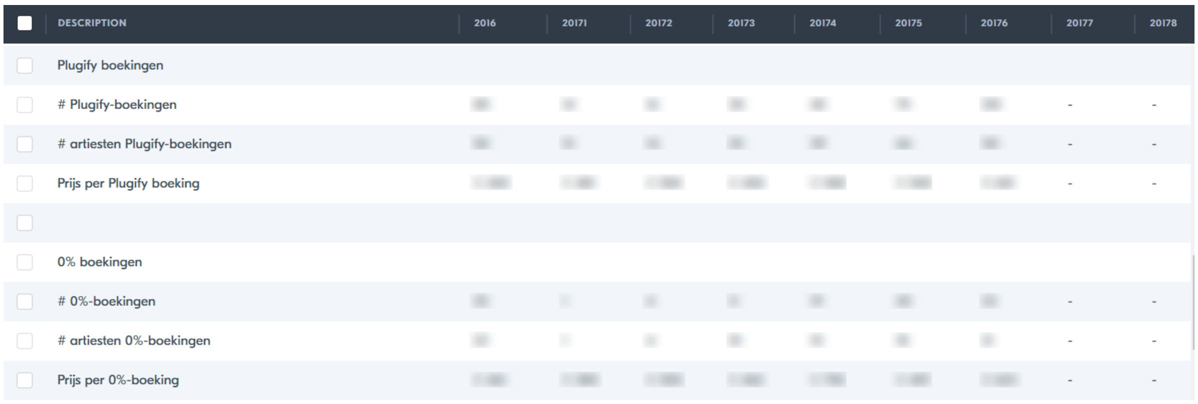
```ruby
# Calculate 0% bookings = Number of bookings where commission = zero
# * <tt>:bookings</tt> - the grouped bookings data
def initialize_zero_percent_bookings(bookings)
  # Declare logic
  monthly_logic = ->(year, month) { bookings[year][month].where(commission: 0).count(:all) }
  yearly_logic = ->(year, _) { bookings[:total][year].where(commission: 0).count(:all) }

  # Calculate all years
  calculate_all_years(monthly_logic, yearly_logic, scale_current_month false)
end
```

Figure 5.3: Example of calculating a metric.

5.2.2. Layout and excel output

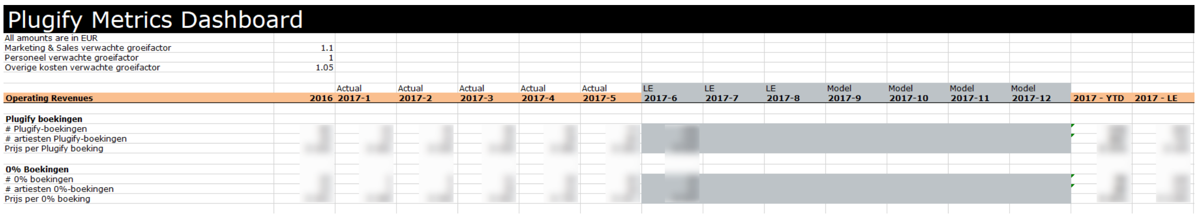
The data is displayed in Forest, by defining a smart collection 'Metrics', and treating each metric as if it was a record in a 'regular' collection. This way, each metric is displayed on its own row. The dashboard automatically adapts to show the current year. Figure 5.4 shows part of the dashboard in ForestAdmin.



DESCRIPTION	2016	20171	20172	20173	20174	20175	20176	20177	20178
Plugify boekingen									
# Plugify-boekingen	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	-	-
# artiesten Plugify-boekingen	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	-	-
Prijs per Plugify boeking	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	-	-
0% boekingen									
# 0%-boekingen	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	-	-
# artiesten 0%-boekingen	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	-	-
Prijs per 0%-boeking	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	-	-

Figure 5.4: Part of the dashboard as shown in ForestAdmin. Values are not representative of the real world situation.

Forest has very limited styling possibilities that make the dashboard a bit hard to read sometimes. However, we also have created the possibility to automatically generate a nicely formatted excel spreadsheet. This also gives the people from Plugify a means to download the data to an offline location, so they are not dependent on accessing the online dashboard every time. A part of the excel output can be seen in figure 5.5



Plugify Metrics Dashboard

All amounts are in EUR														
Marketing & Sales verwachte groeifactor	1.1													
Personeel verwachte groeifactor	1													
Overige kosten verwachte groeifactor	1.05													
		Actual	Actual	Actual	Actual	Actual	LE	LE	LE	Model	Model	Model	Model		
Operating Revenues	**2016**	**2017-1**	**2017-2**	**2017-3**	**2017-4**	**2017-5**	**2017-6**	**2017-7**	**2017-8**	**2017-9**	**2017-10**	**2017-11**	**2017-12**	**2017 - YTD**	**2017 - LE**
Plugify boekingen															
# Plugify-boekingen	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]							[redacted]	[redacted]
# artiesten Plugify-boekingen	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]							[redacted]	[redacted]
Prijs per Plugify boeking	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]							[redacted]	[redacted]
0% Boekingen															
# 0% boekingen	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]							[redacted]	[redacted]
# artiesten 0%-boekingen	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]							[redacted]	[redacted]
Prijs per 0% boeking	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]							[redacted]	[redacted]

Figure 5.5: Part of the generated excel file. Values are not representative of the real world situation

5.2.3. Speed improvements

A problem that was soon encountered was the fact that fetching and processing the data takes quite a while (> 10 seconds). To avoid very long waiting times for the user, we use a caching system to store the data. The data is fetched automatically every hour and kept in cache. This does mean that the data might be outdated a bit, but a delay of an hour outweighs the long waiting times that would otherwise be present.

5.2.4. Predictions

As Plugify is a fast-growing company, a solid prediction of the revenue and profit for the upcoming months is a key feature, that is therefore also present in the dashboard. It predicts certain values, using the most recent historical data. This gives Plugify an idea of where they will stand at the end of the year.

5.2.5. Graphs

Next to the dashboard with the numeric values, we have also utilized the dashboard page of ForestAdmin to display the most important metrics in a graph. This provides the opportunity to see how the company is doing at a glance. Also, visualizing data often gives a better picture than just looking at raw numbers. Figure 5.6 shows the visual dashboard.

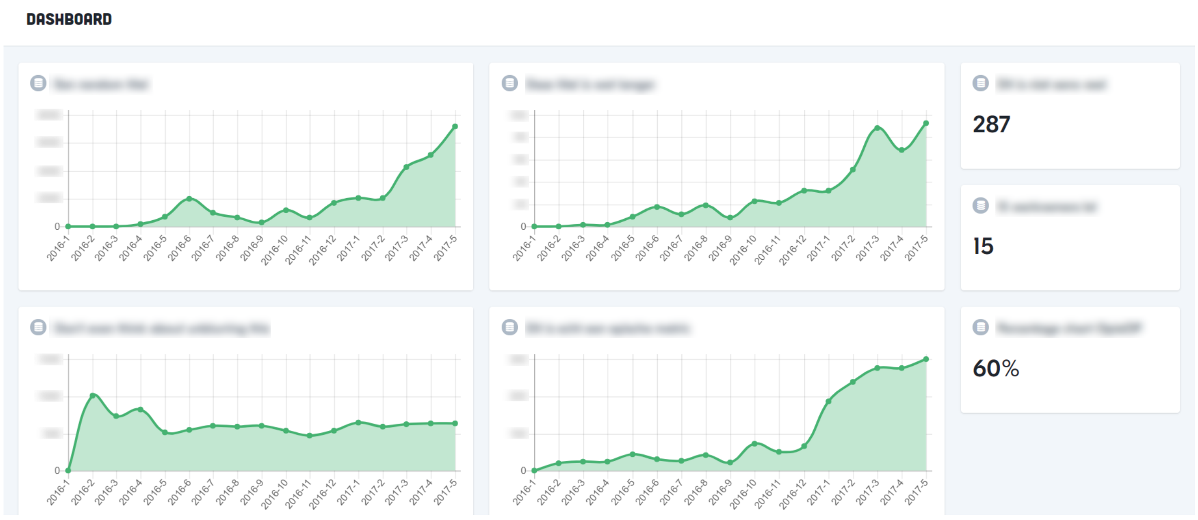


Figure 5.6: The graphic visualization part of the dashboard.

5.3. Notification logic

The process of booking an artist does not always go smoothly. To improve on the conversion rate, the Customer Happiness Team of Plugify can intervene when they notice that something might be going wrong. To keep an overview of what is going on, a notification system exists. Notifications were thrown on events such as the creation of a new request, a message being sent, or an artist rejecting a request

To improve on the efficiency of the CHO team of Plugify, the notification system has been completely revamped, to avoid unnecessary notifications that take a long time to handle. In the new notification system, there are two main categories of events for which a notification should be thrown, as well as some miscellaneous notifications.

5.3.1. Notifications for new users and artists

The first category contains notifications that are related to new users and/or artists. Plugify wants to personally call every new artist and user, to welcome them to the platform, answer any initial questions they might have, and gain information about the origin of their customers, and how they found the platform. Three notifications exist in this category:

- **First request for new artist**. This notification triggers when an artist receives his or her first request. We decided not to throw this notification when an artist registers on the platform, as the artist only becomes relevant for Plugify when he/she starts getting requests.
- **First request for artist in three months**. This notification triggers when it has been a long time (default: three months) since the artist received a request. Maybe the artist forgot the exact procedure, or things have changed on the website. Therefore, the people of Plugify want to call the artist again at this time.
- **New active user with phone number**. This notification triggers when a user with a request history adds a phone number, or when a user with a phone number does his or her first request. This notification is dependent on a phone number being present, as Plugify want to call the customer, and decided upon the fact that sending an e-mail often does not have the desired effects.

5.3.2. Notifications for requests that don't seem to be converting to a booking

The second, and arguably most important category of notifications, are the notifications that are thrown when it seems something is going wrong. The ultimate goal of Plugify is to get as many bookings through their platform as possible, so when a request seems to be stuck, for example due to one party not responding, there should be some manual intervention. There are two notifications in this category:

- **Last request withdrawn by user**. This notification triggers when the last request within a master request is withdrawn by the user. This situation likely means that the user found an artist somewhere else. Plugify likes to know their competition, and what they offer that Plugify does not, so at this point they want to call the client to ask what happened.
- **Fear of missing out**. This notification is the most diverse and sophisticated out of all notifications in the new system. This request is thrown when there is no single request in a group of master request in which everything seems to be going well. If all requests have a problem, there seems to be a high chance that a booking will not come from this group of requests. The following problems related to a request have been identified:
  - **No artist response for 24 hours**: An artist is obliged to respond to a request within 24 hours. If this does not happen, a request can get stuck rapidly, as the user can't do anything except sending a reminder message.
  - **Radiosilence**: Radiosilence refers to the situation where there has already been at least one message from the artist, but it has been a while since messages have been sent. Detecting when a request suffers from radiosilence is not trivial. When planning an event 6 months ahead, a week of silence is absolutely not a problem. However, when the performance is due next week, even a period of two days without a single message can already be a problem. To try and throw the least amount of

false positives and negatives, we decided on the following rule: A request suffers from radiosilence when there have been no messages for 20% of the time between the last message, and the performance date. To avoid notification spam for requests that have a performance date very close by, we use a minimum of 12 hours, and to avoid too many false negatives for requests that have a performance date far away, we use a maximum of two weeks. Because these values are an initial estimate, they are easily changeable in the AP, so Plugify can tweak them when they have formed a better idea on their ideal value.

- **Rejected / Withdrawn**: When a request is rejected (by the artist) or withdrawn (by the user) there is obviously a low chance that this request will result in a booking. Therefore, this is also classified as a problem case.

The fear-of-missing-out (FOMO) notifications are thrown hourly. Every hour, all requests are checked for their status. New FOMO notifications are thrown where necessary, and old FOMO notifications that now have an 'Okay' status are removed.

5.3.3. Miscellaneous notifications

Next to these important groups, there are some miscellaneous notifications that do not really belong together, but still serve an important purpose. These notifications are:

- **Artist received too many warnings**. This notification triggers when an artist has received too many warnings, for example for not responding on time. Currently, an artist can get a maximum of two warnings for leakage, or three other warnings. These values are also easily changeable in the AP.
- **Performance date passed**. This notification triggers when an active request is found that has a performance date that lies in the past. Often, an artist or user forgets to reject/withdraw a request even though they agree a booking will not be made, for example via the message system. This can lead to a lot of outdated data in Plugify's database, distorting the conversion analysis. This notification notifies the people of Plugify that they should, in all likelihood, close this request manually.
- **Negative review given**. This notification triggers when a negative review is given by a user. A negative review means a score lower or equal to 6 (on a scale of 10) for Plugify as a whole, or a score lower or equal to 2 (on a scale of 5) for an artist. Plugify wants to improve their service continuously, so when a user is dissatisfied with Plugify, some research should be performed on why this was the case, and how this can be improved. An artist can, of course, receive a negative review occasionally, but when this happens too often, Plugify should consider talking to the artist, or removing them from the platform. This way, they can guarantee a high quality of artists.
- **New booking with upsell possibilities**. This notification triggers when a new booking is made. At this point, Plugify wants to try and sell another booking. For example, if someone booked an artist for their wedding ceremony, they might also want to book someone for the party afterwards.

5.3.4. Properties and actions

Notifications are handled in the notifications feed, which is explained in section 5.4. When a notification is being handled, it can be snoozed or closed. Snoozing a notification means that you want to look at it again after a set amount of time, whereas closing a notification means you are done with it and will never see it again.

Snoozing and closing with FOMO notifications is a little more sophisticated, as request statuses change all the time. Therefore, we have implemented the following logic, to avoid too little or too many notifications:

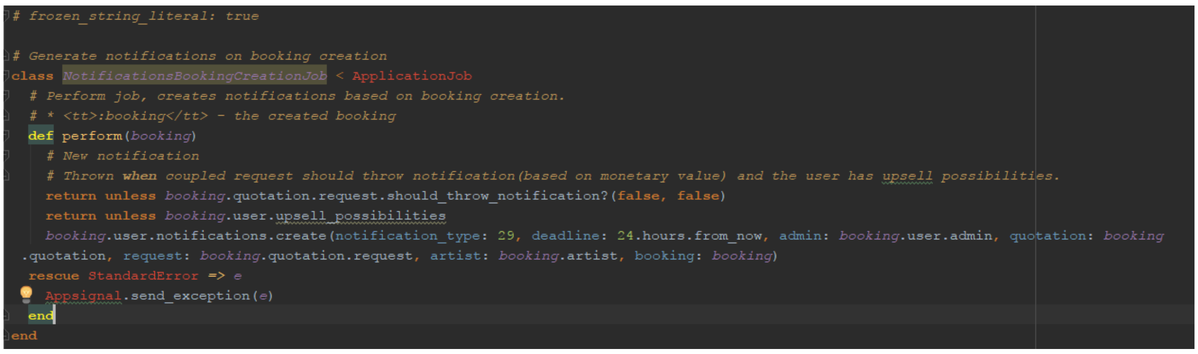
- Closing a FOMO notification means you will never get a FOMO notification for that master request again, even if the status changes from OK to not OK again. Therefore, such a notification should only be closed if you are either certain that everything will be alright and this request will result in a booking, or are convinced that this request is definitely lost.

- When snoozing a FOMO notification, an 'override' check-box is given.
  - When a FOMO notification is snoozed with override, and the status changes from bad to good, the notification is automatically closed.
  - When a FOMO notification is snoozed with override, and the notification was automatically closed, and the status changes from good to bad, it is reopened.
  - When a FOMO notification is snoozed without override, the notification will always come back, even if the status is OK.

Every notification can be assigned an admin, generally a member of the Customer Happiness Team of Plugify. Being an admin happens on a user level, so when a notification is thrown, the admin of the user that the notification is related to is automatically assigned as the admin of that notification. The admin can leave comments, to remind him/herself of the situation the next time the notification comes by, for example after the snooze period has passed.

5.3.5. Implementation

Notifications are thrown in the notification jobs. These jobs are called on certain ActiveRecord callbacks. For example, when a booking is created, a job runs that checks if a 'new booking' notification should be thrown. An example of such a job can be found in figure 5.7



```
# frozen_string_literal: true

# Generate notifications on booking creation
class NotificationsBookingCreationJob < ApplicationJob
  # Perform job, creates notifications based on booking creation.
  # * <tt>:booking</tt> - the created booking
  def perform(booking)
    # New notification
    # Thrown when coupled request should throw notification(based on monetary value) and the user has upsell possibilities.
    return unless booking.quotation.request.should_throw_notification?(false, false)
    return unless booking.user.upsell_possibilities
    booking.user.notifications.create(notification_type: 29, deadline: 24.hours.from_now, admin: booking.user.admin, quotation: booking
 .quotation, request: booking.quotation.request, artist: booking.artist, booking: booking)
  rescue StandardError => e
    Appsignal.send_exception(e)
  end
end
```

Figure 5.7: Example of a notification job. This job is called whenever a new booking is created

5.4. Feeds

To implement the feeds designed in section 4.2, we use ForestAdmin's Smart Views. Smart Views are custom EmberJS components [8], which implies that they are entirely self-contained views. They have direct access to sets of paginated data which ForestAdmin provides. These pages consists of a small collection of objects and their relations, which smart views can update and delete. Furthermore, ForestAdmin provides several extensions that can be used to, among others, trigger Smart Actions, and fetch more pages of data.

5.4.1. Limitations and similarities

The Smart View's code must be provided in the form of only a single pair of HTML and JavaScript files, which implies that there are several limitations we must consider:

1. There is no way for us to create custom EmberJS components, other than the Smart View. This hampers efficient code-reuse almost entirely, as we cannot, for example, define a small component for a certain data item and then use it in multiple separate places.
2. There is no way for us to extend EmberJS, which implies custom controllers, templates, routes, models and tests cannot be used. This limits functionality we can create, limiting

us to modifying available data and calling Smart Actions only. Furthermore, this limits the testability of our Smart Views, as we cannot create or run software tests on them.

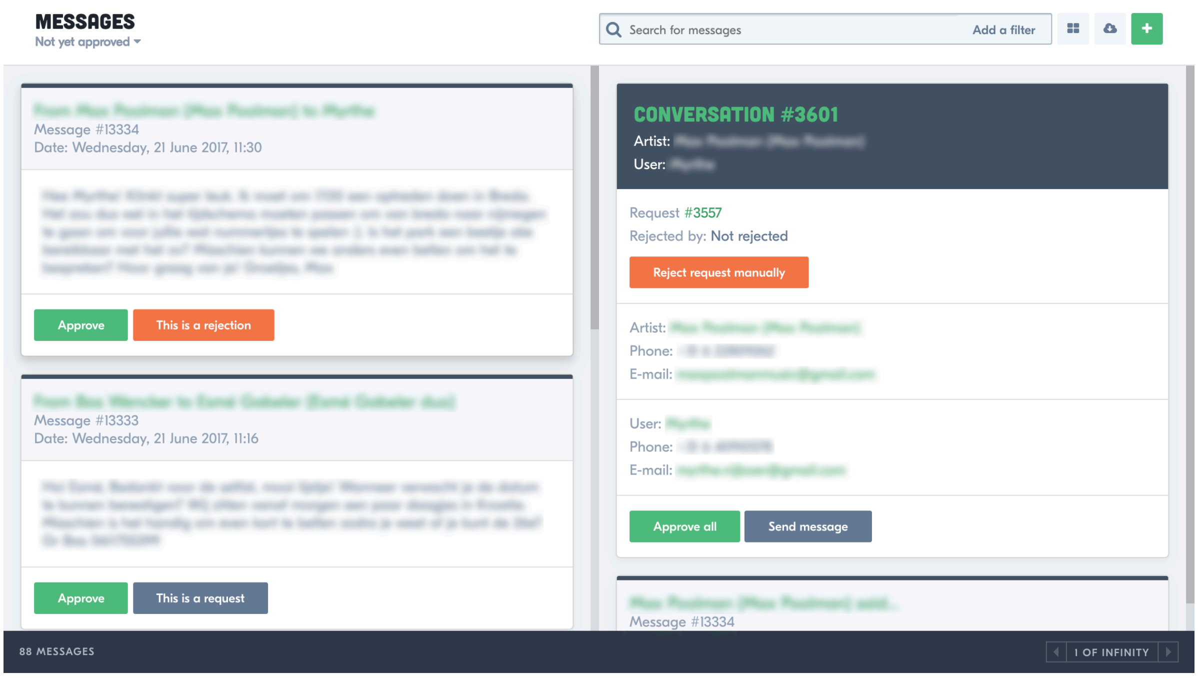
Keeping these limitations in mind, we can still somewhat prevent code duplication. As is discussed in section 4.2, all feeds share a similar two-section layout, with a list view showing paginated data to the left, and a detail view showing selected data to the right. The feeds not only share a similar layout, but can also have similar architectures. They share a number of default methods that perform actions that are entirely identical. These actions are:

- Fetching or refreshing arbitrary data from the Ember Store.
- Fetching or refreshing related information when paginated data is refreshed, or another page is fetched. This refreshes data in the list view section.
- Fetching or refreshing related information for a specific data item, such as when this is selected by a user. This refreshes data in the detail view section.

The triggers for these actions are completely identical in each feed. For example, paginated data is refreshed on a page switch or reload, while selected data is fetched when a new item is selected, and when the item is modified by a user.

5.4.2. Initial layouts

By leveraging Forest Smart Views, and considering the limitations and similarities discussed above, we provide samples of the initial working versions of the three feeds in figure 5.8, figure 5.9 and figure 5.10.



Message #13334
Date: Wednesday, 21 June 2017, 11:30

Message #13333
Date: Wednesday, 21 June 2017, 11:16

CONVERSATION #3601
Artist:
User:

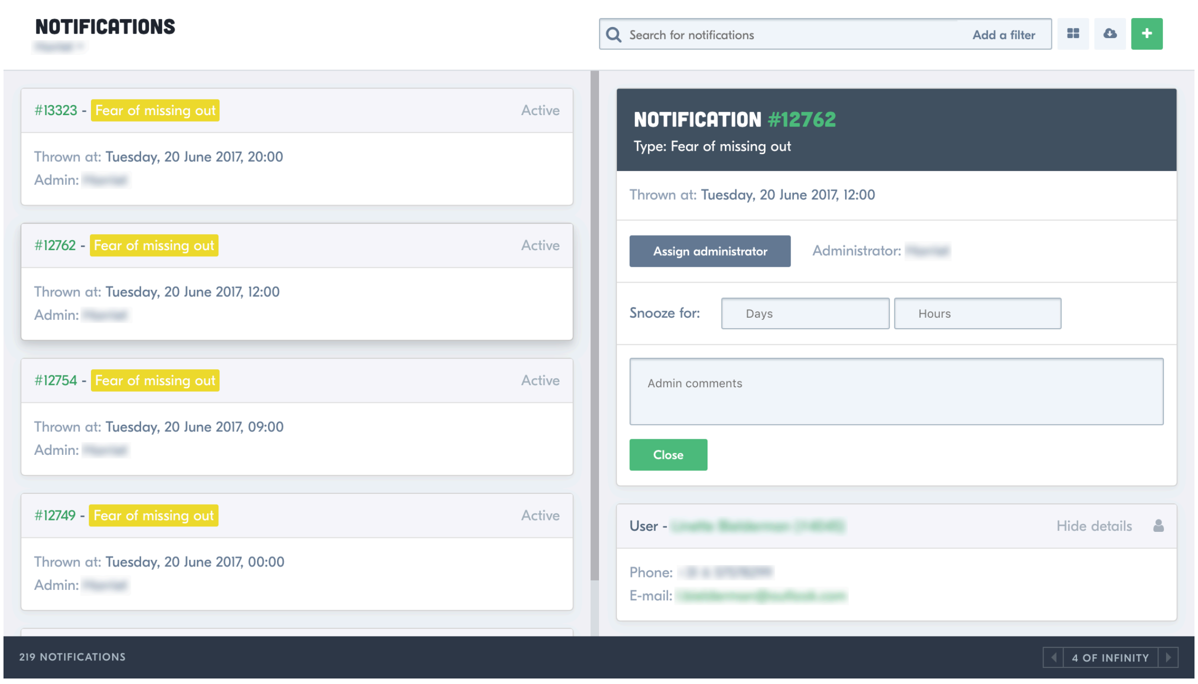
Request #3557
Rejected by: Not rejected

Artist:
Phone:
E-mail:

User:
Phone:
E-mail:

Message #13334

Figure 5.8: Message feed initial version


NOTIFICATIONS
[redacted]

Search for notifications Add a filter

#13323 - Fear of missing out Active
Thrown at: Tuesday, 20 June 2017, 20:00
Admin: [redacted]

#12762 - Fear of missing out Active
Thrown at: Tuesday, 20 June 2017, 12:00
Admin: [redacted]

#12754 - Fear of missing out Active
Thrown at: Tuesday, 20 June 2017, 09:00
Admin: [redacted]

#12749 - Fear of missing out Active
Thrown at: Tuesday, 20 June 2017, 00:00
Admin: [redacted]

NOTIFICATION #12762
Type: Fear of missing out

Thrown at: Tuesday, 20 June 2017, 12:00

Assign administrator Administrator: [redacted]

Snooze for: Days Hours

Admin comments

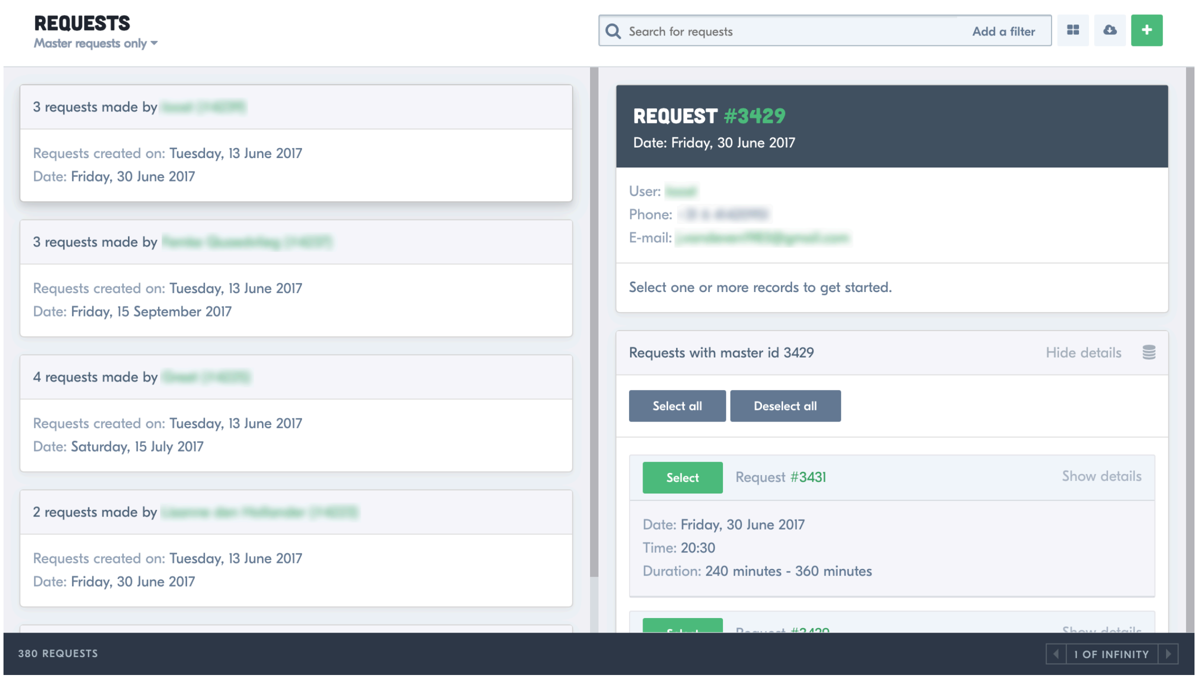
Close

User - [redacted] Hide details

Phone: [redacted]
E-mail: [redacted]

219 NOTIFICATIONS 4 OF INFINITY

Figure 5.9: Notification feed initial version


REQUESTS
Master requests only

Search for requests Add a filter

3 requests made by [redacted]
Requests created on: Tuesday, 13 June 2017
Date: Friday, 30 June 2017

3 requests made by [redacted]
Requests created on: Tuesday, 13 June 2017
Date: Friday, 15 September 2017

4 requests made by [redacted]
Requests created on: Tuesday, 13 June 2017
Date: Saturday, 15 July 2017

2 requests made by [redacted]
Requests created on: Tuesday, 13 June 2017
Date: Friday, 30 June 2017

REQUEST #3429
Date: Friday, 30 June 2017

User: [redacted]
Phone: [redacted]
E-mail: [redacted]

Select one or more records to get started.

Requests with master id 3429 Hide details

Select all Deselect all

Select Request #3431 Show details
Date: Friday, 30 June 2017
Time: 20:30
Duration: 240 minutes - 360 minutes

380 REQUESTS 1 OF INFINITY

Figure 5.10: Request feed initial version

5.5. Other features

In this section we discuss some smaller features that were also implemented during this project. While small, these features are still key to the usability of our system.

5.5.1. CRUD operations

Creating, reading, updating and deleting data is a big part of every administration panel. CRUD operations are available by default in ForestAdmin, however, there were some issues with the default behaviour as it was provided.

Enumerated values

There exist quite some enumerated values in the data model of Plugify, where a set list of options each have their own integer, that is used to store the value in the database. However, this does not provide a good user experience, as the numbers by themselves don't mean a lot to most users of the admin panel. Therefore we created *smart fields* for each of these enums, providing a simple drop-down menu with the options to the users.

Custom update logic

As the Plugify data model contains some dependencies between relations that are not specifically modeled in validations, these are not automatically checked by the default ForestAdmin update logic. The back end code of the old AP already contained some of this custom logic, and we extended this with some more logic. As ForestAdmin provides an example of how to overwrite their default CRUD operations in their documentation [7], the actual implementation was trivial. As the list of custom update logic is rather long, only a few examples are given:

- Update the admin on all notifications connected to a user, when the user's admin is updated.
- Updating the user and artist on either request or conversation, when one of them is updated in the other collection. This has to be done to ensure consistency in the data.
- Deleting new message notifications when all messages in a conversation are approved. As the notifications for new messages is not present in the new AP, this is just for backwards compatibility.

5.5.2. Settings

A small but nice to have feature that we implemented is a global settings system using the gem rails-settings-cached [12]. This allows some constants to be changed easily within ForestAdmin, such as the maximum number of warnings an artist may receive before a notification is thrown. These kinds of values do not need to be changed in code anymore, but can be changed easily in the AP

5.5.3. Warnings

We have developed a simple warnings system, that allows for the possibility of giving warnings to artists. This provides an easy way to keep track of artists that are not acting in line with Plugify's TOS, therefore keeping the overall quality of artists on the platform high. There are a set amount of warning types, and an administrator can also add comments to provide his or her own view on the situation. Some warnings, such as the 'no artist response for 24 hours' warning, are checked and generated automatically.

6
SIG Feedback

In the 6th week of the project we were required to send our codebase to the Software Improvement Group (SIG) [14]. We then had to incorporate this feedback into our system, and send our codebase again for evaluation at the end of the project.

6.1. Issues for delivering the code

When we had to deliver the code to SIG, we experienced some minor challenges. First, all of our back-end code was written in the existing back-end of Plugify. This made it hard to distinguish our code from their code. To fix this, we made a separate project, to which we moved the files that we wrote ourselves completely, or in which we made relevant changes. Second, due to limitations in Forest Admin, our front end code for the smart views contained some major flaws that we, unfortunately, could not fix. We discussed this with the people from SIG, and explained our problem in a README file. In this way, they could account for this when giving feedback.

6.2. First SIG feedback

The first SIG feedback reads as follows (translated, original Dutch version below):

The code of the system has scored 3 stars on our maintainability model, which means that the code is averagely maintainable. The highest possible score was not reached due to low scores for Unit Size and Unit Complexity.

For Unit Size, we look at the percentage of code that is excessively long. Splitting up these kinds of methods in smaller pieces results in a better understandability, testability and maintainability of every single part. Within the longer methods in this system, such as the 'Request.fomo_status'-method, separate pieces of functionality can be found that can be re-factored to their own methods. Comment-lines such as 'Minimum 12 hours, longer than 20 percent of remaining time or 2 weeks' are a good indication that an autonomous piece of functionality is present. We advice to look critically to the longer methods within this system, and split these up where possible.

For Unit complexity, we look at the percentage of code that is excessively complex. Here it holds as well that splitting up these kinds of methods results in a better understandability, testability and maintainability of every individual component. In this case, the most complex methods are also the longest one, so fixing the first problem will also fix this one.

Finally, the presence of test-code is promising, hopefully the amount of test code will also grow when new functionality is added.

Generally, the code scores average. Hopefully you can bring this level up a bit during the rest of the development phase.

Original:

De code van het systeem scoort 3 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code gemiddeld onderhoudbaar is. De hoogste score is niet behaald door lagere scores voor Unit Size en Unit Complexity.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld de 'Request.fomo_status'-methode, zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes. Commentaarregels zoals bijvoorbeeld 'Minimum 12 hours, longer than 20 percent of remaining time or 2 weeks' zijn een goede indicatie dat er een autonoom stuk functionaliteit te ontdekken is. Het is aan te raden kritisch te kijken naar de langere methodes binnen dit systeem en deze waar mogelijk op te splitsen.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Ook hier geldt dat het opsplitsen van dit soort methodes in kleinere stukken ervoor zorgt dat elk onderdeel makkelijker te begrijpen, makkelijker te testen en daardoor eenvoudiger te onderhouden wordt. In dit geval komen de meest complexe methoden ook naar voren als de langste methoden, waardoor het oplossen van het eerste probleem ook dit probleem zal verhelpen.

Tot slot is de aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

Over het algemeen scoort de code dus gemiddeld, hopelijk lukt het om dit niveau nog iets te laten stijgen tijdens de rest van de ontwikkelfase.

6.3. Processing the SIG feedback

To improve the maintainability of our system, we have primarily looked at splitting up long and complex methods. We have split up the main example of the first SIG feedback iteration: Request.fomo_status. Furthermore we have analyzed what methods were (too) long in the code we sent to SIG in the first iteration, and refactored them accordingly. This refactoring meant making our code more concise, splitting some methods up, and extracting duplicated logic from other methods. Through the remainder of the project we have kept the advices from SIG in mind and have written shorter and less complex units of code, extracting duplicated logic wherever possible.

7

Testing and QA

To ensure a reliable system that contains as little bugs as possible, a good testing process is of utmost importance during a software development project. We have employed three different ways of testing: Unit testing using Rspec, manual integration testing in the system itself, and usability testing using RITE In this chapter, we explain how we used these different forms of testing, and how they have improved the reliability of our system.

7.1. Unit testing

For unit testing we have used Rspec [13]. Rspec is an often-used testing framework for Ruby on Rails. As the existing tests for Plugify's back end were written using this framework, we have done this as well. To set up test objects with some data easily, without having to write them to the database ourselves, and manage relations with other classes, we have used the factory_girl [6] gem. This gem provides easy to set up factories for the different models in your system.

We have used Rspec to test both model classes and controller classes, as well as ForestAdmin collection classes. There are some major differences in the way these groups are tested, so we explain them separately.

7.1.1. Model testing

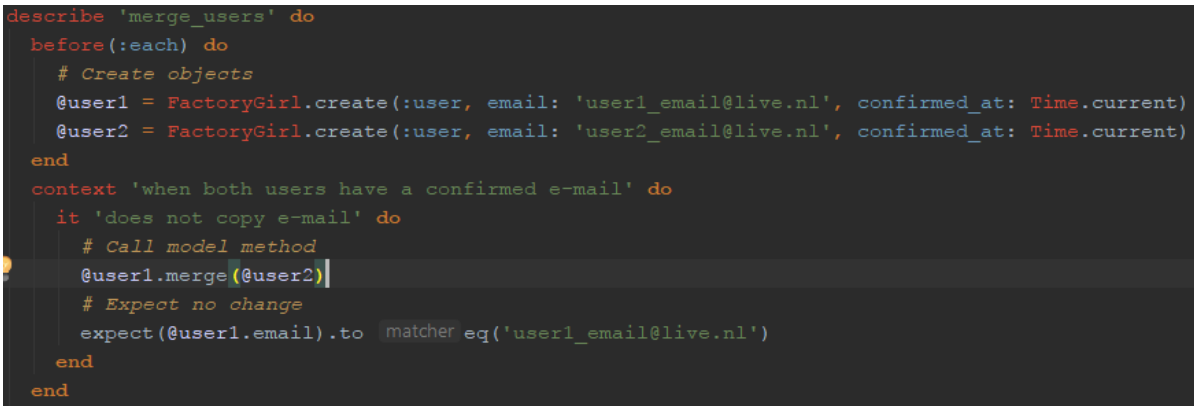
Models and model methods have everything to do with data manipulation. Therefore, tests that test model methods are often of the form:

1. Create an object, possibly with some predefined data like an e-mail address

2. Call the model method on this object

3. Check that some value has changed, or has remained equal, whichever the intended behavior.

An example of a model method test can be found in figure 7.1



```
describe 'merge_users' do
  before(:each) do
    # Create objects
    @user1 = FactoryGirl.create(:user, email: 'user1_email@live.nl', confirmed_at: Time.current)
    @user2 = FactoryGirl.create(:user, email: 'user2_email@live.nl', confirmed_at: Time.current)
  end
  context 'when both users have a confirmed e-mail' do
    it 'does not copy e-mail' do
      # Call model method
      @user1.merge(@user2)
      # Expect no change
      expect(@user1.email).to matcher eq('user1_email@live.nl')
    end
  end
```

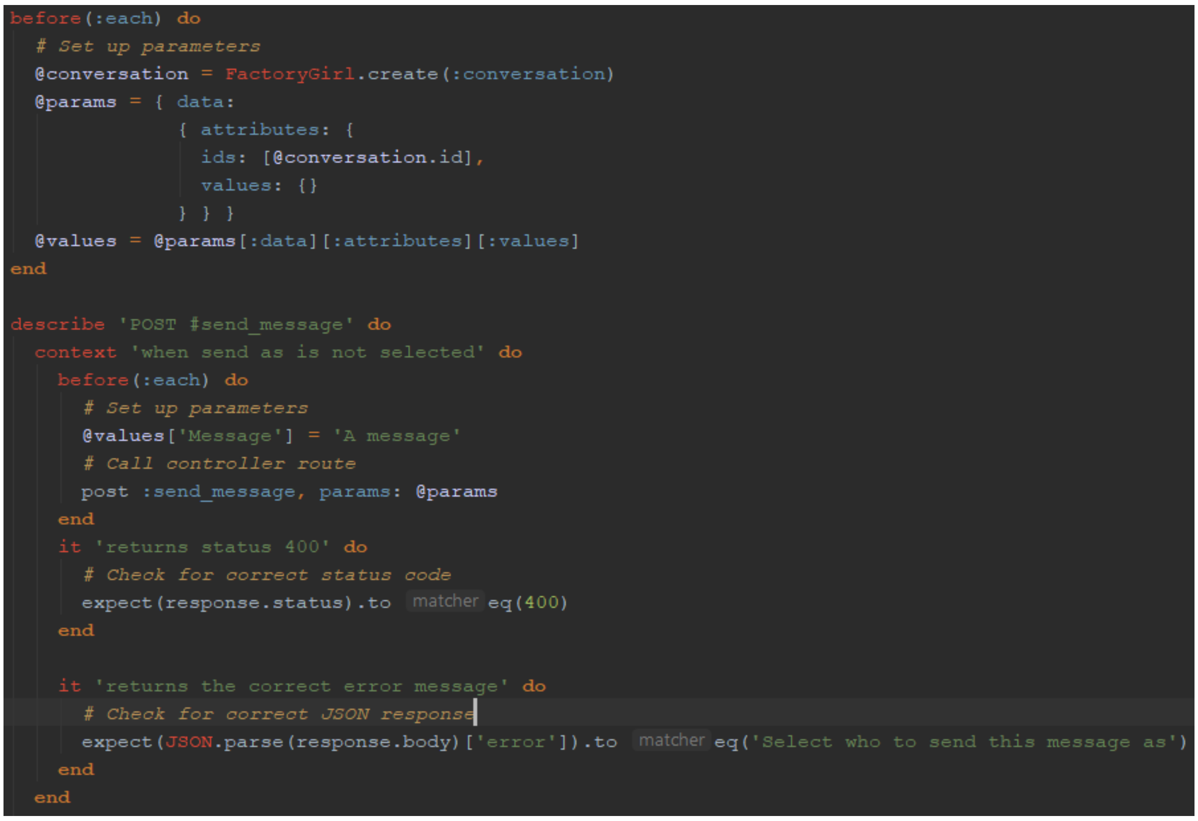
Figure 7.1: An example of an Rspec model test

7.1.2. Controller testing

Controllers exist to receive requests, handle them, and provide a response with (optional) data. Because controllers themselves generally do not manipulate data, their tests look different too. A controller test generally has (some of) the following steps:

1. Set up the response parameters, and execute the call to the controller
2. Check that a certain model method is called on the correct object, based on the given parameters.
3. Check that the correct status code is returned
4. Check that a correct JSON response is returned.

An example of a controller test can be found in figure 7.2



```
before(:each) do
  # Set up parameters
  @conversation = FactoryGirl.create(:conversation)
  @params = { data:
              { attributes: {
                ids: [@conversation.id],
                values: {}
              } } }
  @values = @params[:data][:attributes][:values]
end

describe 'POST #send_message' do
  context 'when send as is not selected' do
    before(:each) do
      # Set up parameters
      @values['Message'] = 'A message'
      # Call controller route
      post :send_message, params: @params
    end
    it 'returns status 400' do
      # Check for correct status code
      expect(response.status).to matcher eq(400)
    end

    it 'returns the correct error message' do
      # Check for correct JSON response
      expect(JSON.parse(response.body)['error']).to matcher eq('Select who to send this message as')
    end
  end
```

Figure 7.2: An example of an Rspec controller test

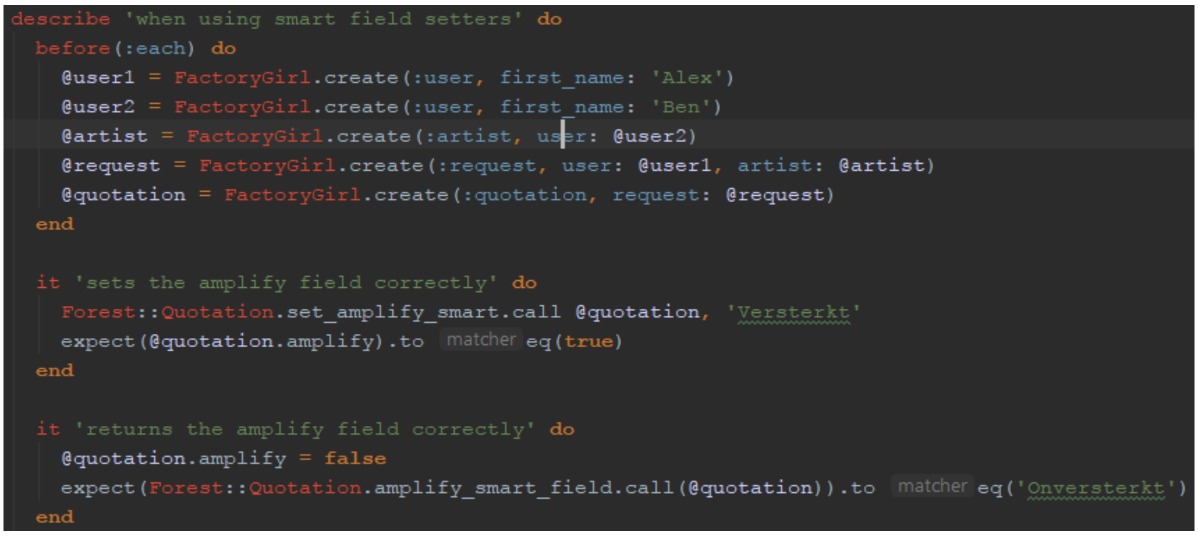
7.1.3. Forest collection testing

The Forest collection classes are classes in which smart fields, actions and segments are defined. Testing these with Rspec was a bit of a hassle, as forest initializes its smart fields, actions and segments on server start. Next to that, setters for smart fields are lambdas that are not easily called from a test class. To fix this, we have extracted the functionality to regular methods, which we then call both from the forest initialization methods, and from the test methods.

A test for a smart field generally has the following format:

- Getter testing
  1. Set up a model object with certain predefined data.
  2. Call the smart field getter method on the object.
  3. Check for a correct response.
- Setter testing
  1. Set up a model object with certain predefined data.
  2. Call the setter with the object, and the data that should be set.
  3. Check the data has indeed changed.

An example of a Forest collection test can be found in figure 7.3



```ruby
describe 'when using smart field setters' do
  before(:each) do
    @user1 = FactoryGirl.create(:user, first_name: 'Alex')
    @user2 = FactoryGirl.create(:user, first_name: 'Ben')
    @artist = FactoryGirl.create(:artist, user: @user2)
    @request = FactoryGirl.create(:request, user: @user1, artist: @artist)
    @quotation = FactoryGirl.create(:quotation, request: @request)
  end

  it 'sets the amplify field correctly' do
    Forest::Quotation.set_amplify_smart.call @quotation, 'Versterkt'
    expect(@quotation.amplify).to matcher eq(true)
  end

  it 'returns the amplify field correctly' do
    @quotation.amplify = false
    expect(Forest::Quotation.amplify_smart_field.call(@quotation)).to matcher eq('Onversterkt')
  end
```

Figure 7.3: An example of an Rspec Forest collection test

7.2. Manual testing

As many features are hard to test with automated tests, we have also performed extensive manual tests in

1. The old Plugify Admin Panel, to ensure we did not break any existing functionality. This is also due to the fact that the old AP barely contained any test code.
2. The Plugify website, as front-end code is hard to test automatically.
3. Our new admin panel, to test specific ForestAdmin functionalities that were customized for our needs. We expect the ForestAdmin tool to be heavily tested on its own, however, our own custom scenarios are new and should therefore be tested. This cannot be done automatically, because we do not have access to all the ForestAdmin code, and a lot of it influences the front end.

A perfect example of something that cannot be tested with automated tests is a smart view. Getting and setting data through a smart view is done via generated Forest controllers that are not accessible in test classes. Therefore, this is tested through manual testing, by

1. Checking the correct data is fetched
2. Changing some data
3. Checking that the new data is stored on database, and correctly re-fetched before, and after reload.

Next to this, it is also not possible to detect certain actions happening on the click of a button automatically. For example, a click on the ’Reject request’ button should prompt a window with some fields that should be filled in. However, this cannot be tested automatically. Therefore, we have tested the calling of smart actions manually, by

1. Clicking the button that should trigger the smart action
2. Checking the correct action window is opened
3. Filling in the necessary data, and executing the action
4. Checking the action had the expected effect on the data

7.3. Usability testing with RITE

To properly test the usability of our system, with concerns to the feeds and the components we have put in place for each feed, we perform usability testing with a small number of members of Plugify that use our product in their day-to-day work.

We choose to use a variation of the RITE (Rapid Iterative Testing and Evaluation) usability testing method [9], as its formal creators advocate its advantages well. In RITE testing, as with many other forms of usability testing, test participants usually engage with the testable product in a verbal (think aloud) manner. The major difference in RITE is that the emphasis lies on rapid changes and adjustments that are iteratively made to the product during the testing process, which can then immediately be verified by test participants.

7.3.1. Rules

For our testing process, we set some simple rules in advance, in compliance with the rules Medlock et al. [9] recommended for RITE testing. These rules defined below:

1. We agree beforehand with testers on the tasks they must perform. In this case, we agree to test general usage of the three feeds, to assess their usage and effectiveness for a CHO’s daily work. Examples of tasks CHOs perform daily are: approving messages, processing notifications, and monitoring users’ requests.
2. We agree to solve issues that have an obvious cause or add missing features that have an obvious solution, within at most two hours during test periods, or before the next period of testing. Issues and features with either a non-obvious cause or solution may be processed at a later time, after testing.
3. We agree to set time aside after each period of testing, to review documented results concerning changes and adjustments made to the product.
4. We agree to run a sufficient number of tests on the product, for at least two weeks or until obvious issues and requests are processed.

7.3.2. Participants

The sole audience of ForestAdmin and the three feeds consists of Plugify’s members. As Plugify is a small company, and our product has a limited user base, we attempt to include at least two CHOs and one administrator of Plugify in our testing process.

7.3.3. Measurements

The primary source of data are the testers' observations and encounters, which we divide into the following types:

- Errors. Errors encountered where the product itself fails critically, that result in failure to complete CHO tasks.
- Issues. Issues encountered with the design of the product, or failure by a tester to properly use the product, that result in failure to complete CHO tasks.
- Critiques. Critiques given of features or design choices of the product.
- Requests. Requests given for features or design choices to be included in the product.

7.3.4. Procedures

Prior to testing we discuss with testers the tasks they have to complete, namely general CHO tasks such as approving user messages, processing thrown notifications, or monitoring users' requests. We also discuss with testers the manner in which they have to communicate. Preferably, we use think-aloud. However, to maximally leverage the testing process, and to remain flexible towards testers, we agree with testers that they can also send us feedback while using the feeds in their daily work routine. We also discuss with testers what they can expect from us, i.e. constant iterations and changes being made to the product during testing periods. As both the group of testers and the group of developers is small, we ensure that at least one developer is constantly working on fixes and improvements for the product during testing periods, iteratively pushing out new builds.

7.3.5. Test results

Below, we list all errors, issues, requests and critiques that are encountered during testing, as well as the direct adjustments made or not made as a result. We show results per feed separately. Results are shown in the direct order in which they have been encountered.

Notification feed measured results

- Issue: tester has a hard time reading and comparing request data for FOMO notifications.
  Adjustment: added more detailed information pop-ups per request in FOMO notifications specifically.
- Request: tester wishes to have ability to assign or reassign an administrator to a notification.
  Adjustment: tester can now assign or reassign an administrator in a small form input in the view.
- Critique: tester does not clearly understand difference between request field and FOMO status field, and mentions the request field may be unnecessary information.
  Adjustment: The request field is no longer shown for FOMO notifications.
- Request: tester wishes to have ability to directly click on an e-mail address, and have an e-mail form show up.
  Adjustment: e-mail addresses can now be clicked on, after which an e-mail form will show up, and a user can directly send an e-mail.
- Request: tester requests ability to view relevant messages sent between users in the feed if a notification concerns a certain request.
  Adjustment: If a single request is provided, at all times a special *Conversation* field is added with the accompanying conversation.
- Request: tester wishes to be able to see the value of a request in the request's data, as well as specifically the average value of a group of requests for a FOMO notification.
  Adjustment: a request's value is now shown together with the request, and for a group of requests the average value is shown.

- Error: tester encounters notifications for which no conversation field is shown, even though it should be.
  Adjustment: the error has been tracked down, and conversation fields are now always shown at the right moment.

- Issue: tester does not understand the difference between *Snooze for* and *Save changes* button.
  Adjustment: The *Snooze for* button is removed, and the *Save changes* button now both saves any changes made, and snoozes the notification if data for this is entered.

- Issue: tester has trouble finding a user's (or an artist's) relevant contact information.
  Adjustment: the user and artist contact information fields have been moved to the very top, and are now always the first visible items for a selected record.

- Request: tester wishes to see all of a user's previous requests if a user field is shown.
  Adjustment: none made, as this was determined to be an issue with a non-obvious or easy to implement solution that might impact the system's performance drastically.

- Error: tester notices that archived notifications, once their archival date has passed, appear in the feed in the exact position they disappeared from, instead of appearing at the top.
  Adjustment: sorting for notifications was incorrect, and has been adjusted to correctly show unarchived notifications in position at the date and time on which they were unarchived.

Message feed measured results

- Error: not all messages in a conversation are shown.
  Adjustment: this is an error with pagination. Now all messages are fetched successfully.

- Critique: tester has comments on the way marking a request as rejection is currently handled. They imply that it would be easier if, in the list view, artists' messages could be marked as rejections with a single button press, and users' messages could be marked as withdrawals with a single button press. Currently, they have to select a notification, select *rejection*, and enter a form where they select whether the relevant artist rejected or the user withdrew. This process, according to the tester, is cumbersome, and could be simplified.
  Adjustment: Messages can now be marked as a rejection/withdrawal with a single button press and form field. Required button pressed have hence been reduced.

- Request: tester wishes to have ability to *white-list* an artist, so some artists' messages never appear, as these can be considered *safe*.
  Adjustment: feature implemented. User can now white-list an artist, after which messages that concern them are never displayed.

- Critique: tester notes that they always have to select (and thus load) a message, to check if there are other messages in the relevant conversation which need approving, for each and every message. They recommend that a count is added to every message that shows whether there are other unapproved messages in the relevant conversation, so they can skip selecting most messages and just approve them directly.
  Adjustment: feature implemented. User now sees a count of other unapproved messages in a conversation, per message.

- Issue: tester cannot find user/artist data in messages feed.
  Adjustment: while the tester could click on an artist's id (and similar for a user), we added two fields showing artist/user contact information to the top of the feed.

- Issue: tester cannot set attribution data for a user in the messages feed.
  Adjustment: tester assumed this feature already existed, while it did not. Feature added.

Request feed measured results

- Critique: tester notes that request are grouped by creation date on the same day, while they should be grouped by performance date.
  Adjustment: requests are now grouped by performance date.
- Request: tester requires ability to flag requests as *completed* so they can be hidden. Otherwise, requests that are fine will remain visible in the feed for no reason.
  Adjustment: at time of writing, none are made. Will be adjusted after report deadline.
- Issue: tester cannot see type of artist per request.
  Adjustment: this has been added, and can now be viewed at a glance.

7.4. Usability testing through measurements

As can be seen in subsection 2.4.2, we have previously gathered performance and usability metrics for the old system, with regards to how actions are performed, the number of clicks required per action, as well as the general time taken for each action. To objectively measure the usability of our system, we will now revisit these same actions, and perform the same measurements in the new system, with regards to the three feeds.

We will analyze the speed and ease of performing the following actions in our system, with specific regards to the three feeds. These are adapted slightly from their previous specifications, because of design changes to the system. For example, a 'new message' notification no longer exists, but the 'message' feed fulfills the same and more purposes. The actions are:

- Checking if a new request is unique, and otherwise deciding what other unique request it belongs to.
- Handling a new message. ('new message' notifications no longer exist, and are replaced by the message feed)
- Handling a 'FOMO' notification. ('1 open or all rejected' notifications no longer exist, and are replaced by 'FOMO' notifications)

We will not measure (subjective) intuitiveness of actions to perform, as we have gathered enough information on this in section section 7.3. We will only measure the following:

- The number of clicks required to perform the action.
- The total (average) waiting time for pages to load.

Checking if a group of new requests is unique, and otherwise deciding what other group some requests belongs to

1. Click to select request group.
2. Read contents (date, time, type) of all requests, and decide whether there should be subgroups for different requests.
3. Group requests if necessary, by clicking on the 'Select' button for all groupable requests, and then clicking on the 'Group requests' button.

This action requires at least one click, and at most the number of requests that must be separated. The total average time spent waiting for the feed to load data is 3.8 seconds. Selecting requests does not influence load times, as data has been fetched already.

Handling a new message
The sequence of actions to do this in the new message feed is as follows:

1. Read message content.

2. If message is ok, click ’approve’, ’mark as request’, ’mark as withdrawal’, or ’mark as rejection’ button. Otherwise, continue.
3. Click to select message.
4. Read other messages in conversation.
5. If message is ok, click ’approve’, ’mark as request’, ’mark as withdrawal’, or ’mark as rejection’ button.
6. If a form appears, fill in required data and complete form.

This action requires between one and four clicks, depending on the message. The total average time spent waiting for the feed to load data is 4.0 seconds. Selecting messages does not influence load times, as data has been fetched already.

Handling a ’FOMO’ notification
The sequence of actions to do this in the new notifications feed is as follows:

1. Click to select ’FOMO’ notification in notifications feed.
2. Read through available user data, and read through available request data for each associated request. No clicks are necessary, as all data is presented instantly.
3. If the situation is okay, click the ’Close’ button.
4. Otherwise, fill in text field, choose snooze date if necessary, and click ’Save changes’.

This action requires between 2 and 5 clicks, depending on the situation. The total average time spent waiting for the feed to load data is 4.5 seconds. Selecting notifications does not influence load times, as data has been fetched already.

8 Final Product Evaluation

At the end of the development process, after we have designed, implemented, tested and iterated the product, we now step back to assess and evaluate what has become the final product. First, in section 8.1, we analyze which functional requirements that we set in section 2.5 have been modified or missed. Then, in section 8.2, we take another look at the goals we set in section 2.3. Then, in section 8.3, we assess the ethical implications of the final product.

8.1. Functional requirements assessment

In this section, we identify and discuss functional project requirements that we have either not completed, or were decided not to be part of the actual product during development due to requirements and design changes.

8.1.1. Dropped or modified functional requirements

We first discuss functional requirements that, due to changes in how the design would turn out, have been dropped or modified.

- *In the message feed, it should be possible to snooze the conversation for a certain amount of time, so that the user will be reminded again at that time. This snooze may be interrupted by certain triggers, such as the creation of a booking.*
  This requirement was dropped, because it was deemed inconsistent with the feed's final design. Messages are dealt with directly in the message feed.

- *In the request feed, the feed should contain all requests that are possibly unique.*
  This requirement was dropped, because it was deemed inconsistent with the feed's final design.

- *A new artist notification should trigger when an artist receives a request and it has been three months since someone from Plugify last spoke to the artist.*
  This requirement was dropped, as it was considered harmful to a CHOs productivity if they had to mark when they spoke to an artist every time they did.

- *A FOMO notification should trigger when nothing has been done with automatically sent suggestions.*
  This was dropped as, in the end, automatic suggestions fell outside the scope of the project.

- *For dashboard graphs, it should be possible to indicate start and end points.*
  This was dropped, as the dashboard graphs function differently from how we envisioned them.

- *For dashboard graphs, it should be possible to adjust the intervals of measurements.*
  This was dropped, as the dashboard graphs function differently from how we envisioned them.
- *For dashboard graphs, it should be possible to filter results.*
  This was dropped, as the dashboard graphs function differently from how we envisioned them.

8.1.2. Missed functional requirements

We now discuss those requirements that we have missed due to factors such as loss of scope, time constraints, and shifting priorities.

- *For a potential loss notification, it should be possible to indicate why this loss occurred.*
- *For a potential loss notification, it should be possible to indicate which other artist was booked, on what platform, if possible.*
- *For a FOMO notification, it should be possible to mark what actions have been taken.*

8.2. Project goals assessment

We return to the project goals that were defined in section 2.3. For thes goals, we used the Goal-Question-Metric approach [1]. Hence, can assess our progress towards achieving the goals we set beforehand by answering the questions defined for these goals, through use of the metrics for which we have gathered data.

The goals we defined concern the following areas: feature-completeness, usability, maintainability, reliability, and performance. Per goal, whose original definitions are repeated below, we add an assessment concerning that goal's completion.

- Feature-completeness
  - Goal: Ensure the system is feature-complete, serving Plugify's needs.
  - Questions: Are all features Plugify's employees need present in the system? Are any features Plugify's employees need missing?
  - Metrics: Number of missing features indicated by Plugify's employees.
  - Assessment: as evident from RITE testing in section 7.3, which identified missing features in the product, as well as the fulfilled functional requirements discussed in section 8.1, we can confidently state that, while not all requirements have been fulfilled, we have brought the system to a sufficiently usable state where it is regarded as *complete*. It not only encapsulates all functionality of the old system, but contains all vital requested functionality, as evident from testing. The goal is hence considered achieved.
- Usability
  - Goal: Improve the usability of the system for its users.
  - Questions: Is the performance and efficiency of CHOs and other users improving? Is the satisfiability of CHOs and other users with the systems improving?
  - Metrics: Time taken per task user has to complete, number of clicks taken per task user has to complete, and (subjectively) general satisfiability of users with the system.
  - Assessment: as evident from RITE testing in section 7.3 and other gathered metrics in section 7.4, we see a definitive improvement in usability.
    Objectively, we see the required number of mouse clicks per task decrease drastically. For example, when dealing with a complex 'FOMO' notification, relevant data is available at a glance and mouse clicks have dropped from 11 to at least two and at most five clicks.

Subjectively, feedback from users during RITE testing has been positive, with very little critique being given, and all feature requests and encountered issues having been responded to.
The goal is hence considered achieved.

- Maintainability
  - Goal: Improve the maintainability of the system for developers.
  - Questions: Is the system maintainable? Is it well structured? Does it follow common software design principles?
  - Metrics: Percentage of code duplication, average cyclomatic complexity, maximum cyclomatic complexity, test coverage, and (subjectively) number of other code smells.
  - Assessment: following the same tools as described in 2.4.2, we analyzed the final code base of our project. We only considered the parts of the code base that we touched or wrote to ensure the analysis is accurate. We reduced the number of duplicated lines according to Simian to 1190 over 71 cases of code duplication, compared to 1714 lines over 150 cases in the old AP. Only 26 of our duplicated lines are in the crucial Ruby on Rails back end code. We drastically improved the test coverage, from 13% in the old Rails AP code, to 87% in the new Rails AP code. We also used Rubycritics to analyze the Rails code, which concluded with an average complexity of 28 per file, 13 duplications per file and 5 smells per file. This is a major improvement compared to the average complexity of 106 per file, 42 duplication per file and 13 smells per file in the old AP. As these metrics show a major improvement to the maintainability of the AP codebase, this goal is considered achieved.
- Reliability
  - Goal: Improve the reliability of the system for its users.
  - Questions: Is the system reliable? Are bugs present in the system? Is the system well tested?
  - Metrics: Test coverage, number of encountered errors during general usage.
  - Assessment: As discussed in the analysis of the maintainability goal, we improved the test coverage from a bare 13% in the old AP code, to 87% in the new AP code. This increase in coverage provides a major improvement to the reliability of the system, as confidence in well-tested code is generally heightened. Furthermore, all errors that were encountered in usability testing, as few as they were, have been resolved. Hence, the system is considered reliable. The goal is hence considered achieved.
- Performance
  - Goal: Improve the runtime performance of the system.
  - Questions: How fast is the system? Does it display data quickly? Does it perform well under load?
  - Metrics: loading times of system, loading times of views that require large amounts of data.
  - Assessment: we see a definitive improvement in performance of the system. For example, in section 7.4, we gathered load time metrics of specific pages that proved to be bottlenecks in the old system. For handling a 'FOMO' notification or similar, we see a dramatic decrease from 23.11 seconds loading pages to 4.5 seconds.
    The goal is hence considered achieved.

As discussed above, we consider all the major project goals achieved. We attribute this to the GQM method, which provided us not only with a clear goal-set from the beginning, but also allowed us to constantly revisit the project goals during development.

8.3. Ethical implications

To assess the ethical implications of our product, we consider the following questions:

- What influences does the product or use of the product have on people?
- What consequences does the product or use of the product imply for people?

While most parts of the product influence people on a minimal scale - the product is, after all, contained within Plugify and for use by its members only - there is one small part of the product that is exempt from this rule. This exception is the message feed, which both influences and bears consequence for artists and users.

The message feed allows a CHO or an admin to directly read all the conversations between Plugify's artists and users. Furthermore, it has the capability for modifying - and even sending - messages. This serves a two-fold purpose. CHOs can monitor for inappropriate messages to safeguard users and artists, and they can better monitor requests, to ensure that these convert to a booking. This bears consequence for users that violate Plugify's policies.

These features were and always have been present, as the previous administration software shared these capabilities. Furthermore, artists and users should be aware of these capabilities, as Plugify has ensured to clearly mention their rights to these, paired with a thorough explanation, in their terms of service [11].

9 Conclusion

When we started this project 10 weeks ago, the people of Plugify were managing their data model and funnel with the existing administration panel. Although the functionality of this panel worked as expected, there were some serious usability issues, as well as some outdated or missing features identified by Plugify over the last year.

We have managed to build a new administration panel that perfoms well and is intuitive to use. The new notification system ensures the notification feed is no longer overloaded with message related notifications, but instead allows users to focus on converting as many requests to a booking as possible, while not forgetting to welcome new users and artists to the platform.

The three new feeds allow for rapid processing of all relevant data that comes through the platform, something that took a lot of time in the old administration panel. This allows the CHOs to work more efficiently, something that is key to a fast-growing business like Plugify.

The new dashboard functionality provides an easy way for everyone inside the company to see how the company is doing at any time. Instead of manually fetching data from different places, as was previously the case, this now happens automatically, making sure the most recent data is always shown.

The people of Plugify are enthusiastic about the new system we developed. Although the CHOs still need to get used to the new system, during usability testing it became clear that some trivial tasks that took a lot of time in the old administration panel, are now performed fast and with ease.

We are thus content with the end results of the project, and look forward to seeing Plugify put the project to good use in the future.

10 Discussion and recommendations

As the project is finished, we step back and offer some recommendations regarding improvements and additions that can be made to the system in the future. Furthermore, we take some time to look back at our choice to use the ForestAdmin administration software for a base administration panel. We discuss ForestAdmin specifically, as it has had a defining impact on the project as a whole.

10.1. Improvements and additions

While the system we implemented is wholly usable by Plugify as is, there are areas to the system where improvements can be made or features can be added that Plugify may need in the future, that we did not cover. For example, one area that fell outside the scope of our project is the automatic generation and sending of artist suggestions to Plugify's users. This provides benefits which Plugify can leverage to better funnel users towards a booking.

Another area we did not touch, mostly due to time constraints, is the storage and analysis of data surrounding a CHO's work and interaction with clients. For example, data regarding call time with users, or even just data regarding what tasks take up a CHO's time the most, can be invaluable to Plugify.

10.2. Regarding ForestAdmin

About ForestAdmin, we can say the following: any and all requirements of basic administration software, such as data creation and updating, are included in ForestAdmin by default. As such, the use of ForestAdmin has bootstrapped our project, saved us weeks of work, and allowed us to focus entirely on the specific features that Plugify needs, such as a smarter notification system, accessible data metrics, and more usable software for CHOs.

We do have to mention, however, that projects should only use ForestAdmin if they do not have very specific requirements of the software. As ForestAdmin is not completely open source software, it may be difficult to tailor it to specific needs. Furthermore, ForestAdmin is under active development by a small team, and as such may suffer from a number of issues such projects often have. Still, considering these drawbacks, we can recommend ForestAdmin as it is a capable piece of administration software.

A

UML diagrams

Figure A.1: Class diagram of the existing administration panel

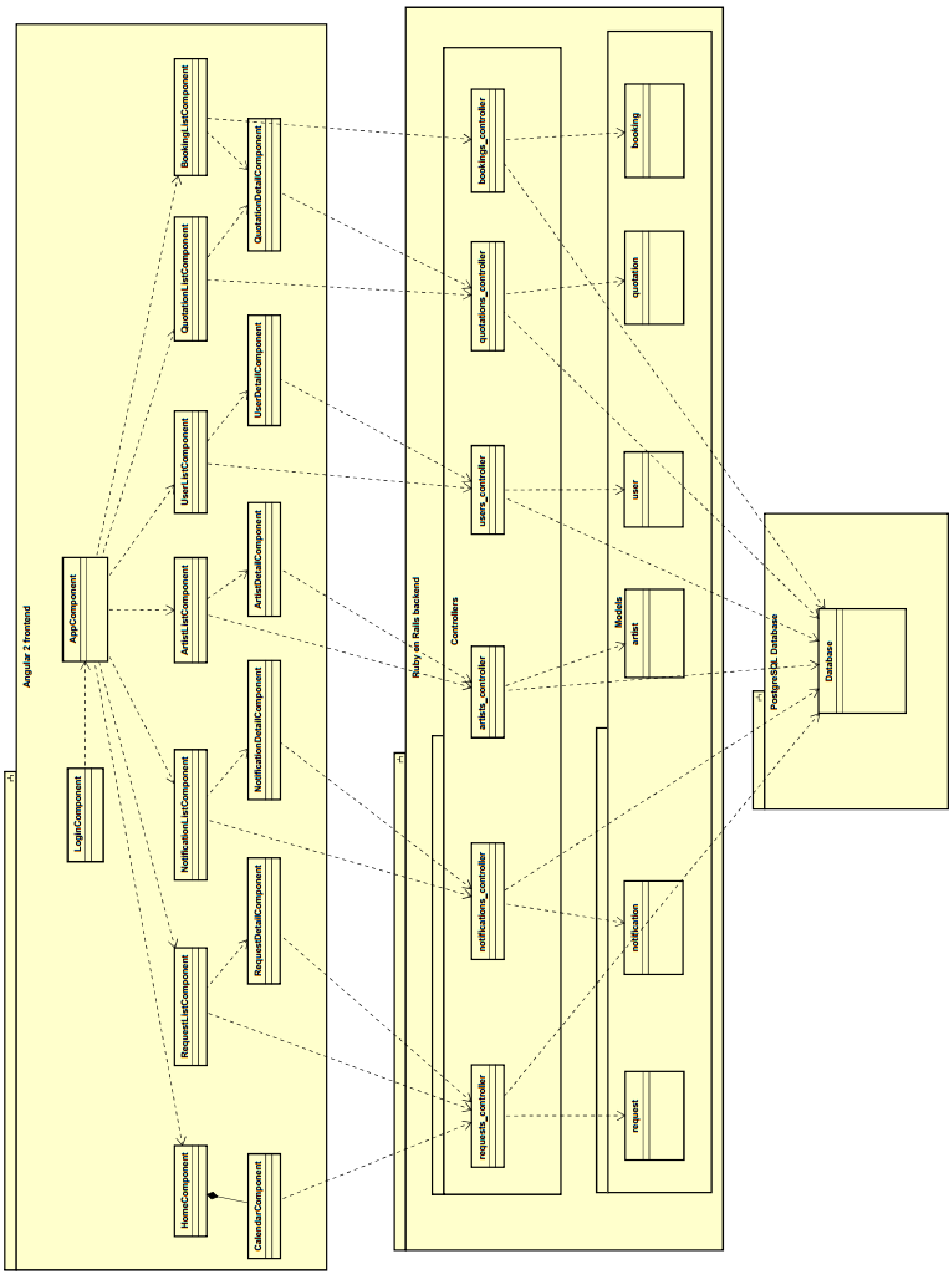
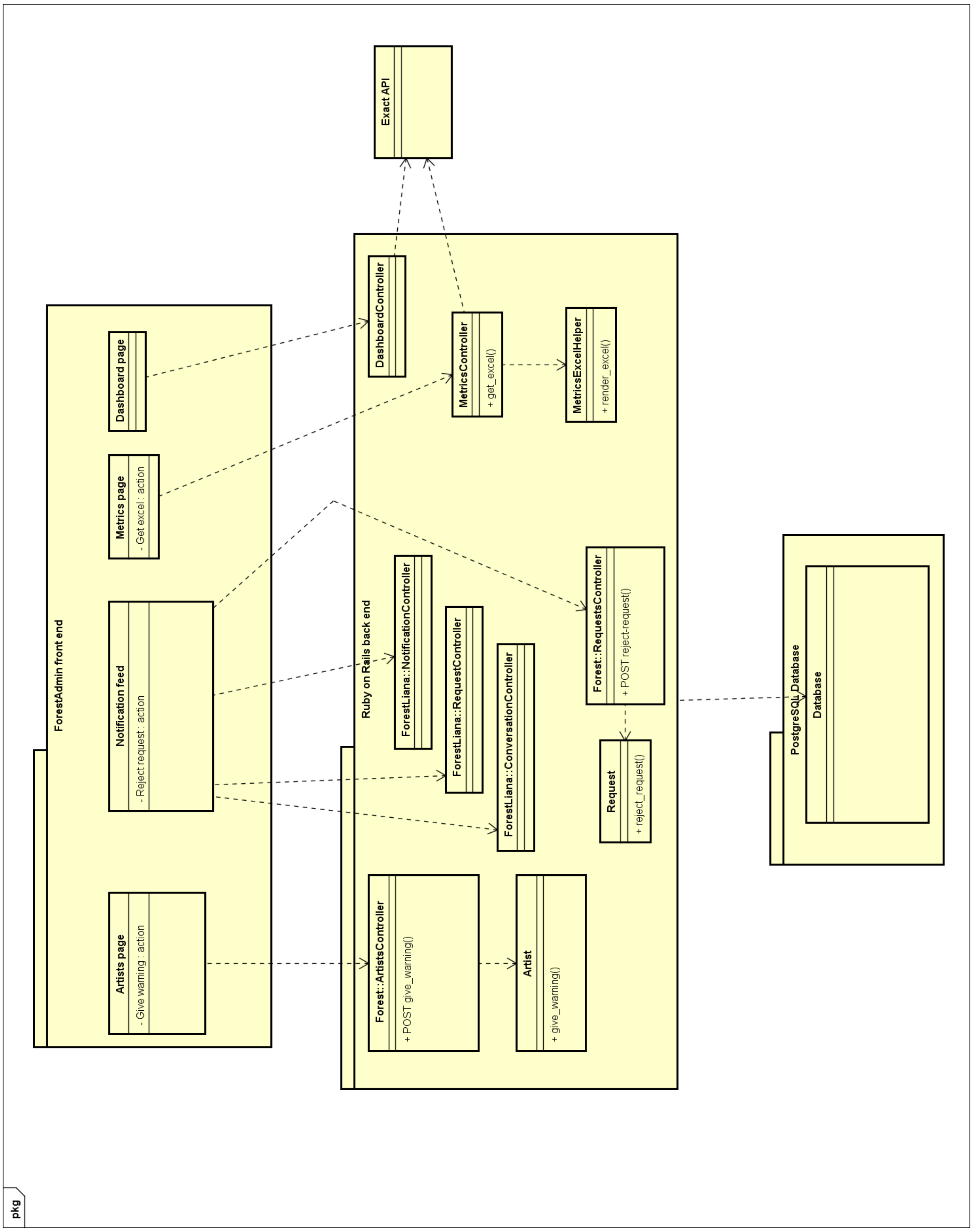


Figure A.2: Class diagram of the new administration panel



B
Interview questions

The following topics and covering questions are used for a semi-structured interview.

- Topic: regarding your work:
  - Can you describe, in short, what your work entails?
  - How much of this work is done using the AP?
  - Are there things you do manually outside the AP, that you feel can/must be incorporated in the AP?
- Topic: regarding notifications in the AP:
  - Can you describe, in short, how you process a notification?
    - ◊ Are there steps during this process that you feel are unnecessary or too complicated?
    - ◊ Are there steps or data that are similar, for all or most notifications?
    - ◊ Are there steps or data that you feel are missing, for all or most notifications?
  - What notifications take the most time to process, and why?
  - Do you feel there are notifications missing, and why?
  - Do you feel there are unnecessary notifications, and why?
- Topic: regarding the AP, excluding notifications:
  - Can you describe, in short, how you use the AP outside notifications?
    - ◊ Are there parts of the AP that you feel are unnecessary or too complicated?
    - ◊ Are there parts of the AP that you feel are buggy, or broken?
    - ◊ Are there parts not in the AP, that you feel should be in the AP?
  - What part of your work in the AP, excluding notifications, takes up most of your time?

C Project description

C.1. What do we do?

Plugify is the online marketplace for the viewing, listening and booking of live music. From bands to dj's to ensembles to solists, something for every occasion. Plugify was founded because booking a performance can be made so much easier. As an artist you are able to create an artist page in an approachable manner, after which all information that is relevant for customers is presented in an attractive form. As a customer you get in touch with your chosen artist, who you can then book online, including the necessary technology. Plugify is the new way to book live music.

C.2. What will you do?

Because Plugify is a web platform, the entire application depends on the database which stores all data regarding users, artists, requests, bookings, etc. To maintain this database and easily edit user information we use an AP(Admin Panel). Our CHO (Customer Happines Office) team also uses the notification system that is built into this AP to keep track of the actions of users and artist, so they can assist when necessary.

Our current AP is getting out-of-date, and could use some bug-fixes, updates and new features, as this would improve the efficiency of our CHO team. Some things you could think about:

- new (smarter and faster) information system for our CHO team
- improvements on loading times and reliability
- automation of many trivial tasks
- intelligent processing of data
- giving insights into company financials and making forecasts based on historical data

Whether you will update the existing AP, or create a new one is up to you.

C.3. Which technologies do we use

When I did my own BEP, my topmost priority was to know which technologies I would be working with. If you get to work with a startup, you get the chance to work with the newest web technologies. At Plugify those are:

- AngularJS and Angular2
- Ruby on Rails 5.0
- Amazon Web Services

And of course you are free to introduce something new.

C.4. Company description

Plugify is a young and ambitious startup with an office opposite Amsterdam Central Station. Since the launch of the platform in March 2016 and a funding round in September 2016 we have been growing rapidly. With a 12-man team we are working on modernizing the nontransparent world of live music every day.

Also check our website: https://www.plugify.nl

Bibliography

[1] Victor R. Basili, Gianluigi Caldiera, and Dieter H. Rombach. *The Goal Question Metric Approach*, volume I. John Wiley & Sons, 1994.

[2] Greg Bell. The administration framework for business critical ruby on rails applications. 2011. `https://activeadmin.info/` (visited: 04-05-2017).

[3] Exact Software Nederland B.V. Exact website. 2017. `www.exact.nl` (visted: 27-06-2017.

[4] Plugify b.v. Plugify website. 2017. `https://www.plugify.nl` (visited: 26-06-2017).

[5] Elmas. Elmas exact api wrapper gem. 2017. `https://github.com/exactonline/exactonline-api-ruby-client` (vistited: 27-06-2017).

[6] factory girl. factory-girl gem for setting up ruby objects as test data. 2017. `https://github.com/thoughtbot/factory_girl` (visited: 27-06-2017).

[7] ForestAdmin. Developers guide. 2017. `http://doc.forestadmin.com/developers-guide/` (visited: 04-05-2017).

[8] Tilde Inc. Ember.js - ember.component documentation. 2017. `https://www.emberjs.com/api/classes/Ember.Component.html` (visited: 21-06-2017).

[9] Michael C. Medlock, Dennis Wixon, Mark Terrano, R. Romero, and Bill Fulton. Using the rite method to improve products: A definition and a case study. *Usability Professionals Assocation*, 51, 2002.

[10] Michael D. Myers and Michael Newman. The qualitative interview in {IS} research: Examining the craft. *Information and Organization*, 17(1):2 – 26, 2007. ISSN 1471-7727. doi: https://doi.org/10.1016/j.infoandorg.2006.11.001.

[11] Plugify. Terms of service. 2017. `https://www.plugify.nl/ondersteuning/gebruiksvoorwaarden/` (visited: 27-06-2017).

[12] rails-settings cached. Rails settings gem. 2017. `https://github.com/huacnlee/rails-settings-cached` (visited: 27-06-2017).

[13] Rspec. Rspec testing framework. 2017. `http://rspec.info/` (visited: 27-06-2017).

[14] SIG. Software improvement group. 2017. `https://www.sig.eu/` (visited: 27-06-2017).