Leopold-Franzens-Universität Innsbruck

Institut für Informatik
Datenbanken und Informationssysteme

Development of an OpenSource Feedbackcommuncationplatform

Bachelor-Arbeit

Martin Karrer

betreut von
Wolfgang Gassler und Eva Zangerle

Innsbruck, May 6, 2016

Abstract

Within this work, the development process of an open-source feedback collecting tool with the ability to communicate with the feedback-giving-person and the maintainer has been documented. Most feedback applications only offer a one-way communication from the feedback-giving-person to the feedback-requester. The approach of how this system has been developed with the ability to communicate or discuss a feedback, will be described in detail later on. This system is extendable, scale-able and especially easy to maintain and install.
All techniques and patterns used in this project to implement the single-page-application will be described in the relevant chapter. This project tries to show the utilisation of a functional language in everyday life and all advantages and disadvantages which have occurred during the implementation of said project.
The aim of the presented application is to be a minimal but extendable and scale-able open-source platform used for requesting and discussing feedback.

Contents

Chapter 1

Introduction

Getting honest feedback is in itself a challenge and it is even harder if it is not possible to collect this feedback anonymously. To be able to gain anonymous feedback the system has to accept feedback with and without user information. This application collects feedback and then makes it possible to discuss the content separately afterwards.
An e-mail mailbox or some online discussion board is commonly used to collect feedback. Most of these online services do not scale for the user or the system. For example: We held a presentation in front of 400 people and now we want to know if the audience liked it? Sending an e-mail containing feedback does not scale. Even in the best case scenario, with a feedback-friendly audience, one will get over 400 messages in ones inbox. Managing feedback in a discussion board can be a huge hassle if one wants to find and manage the feedback without spending more than a thought on specifically discussing one of the feedback entries.
In this project the main goal was to create an online platform, where anyone is allowed to create a feedback-channel with a specific topic. In this channel the users are able to post feedback anonymously or provide their identity as they please. Each feedback in those channels can be up-voted or down-voted and every user can reply to a specific feedback to discuss or clarify misunderstandings. The administrator of the channel is not restricted by any rules and can also collect universal feedback through not being constricted by predetermined questions. Therefore Constructeev will not provide a predefined questionnaire but the user can write down his/her own view instead and make suggestions for improvement. This will result in better and more widespread feedback.

Chapter 2

Requirement Specification

A description of the software requirements for Constructeev will follow below. This chapter mainly describes the necessary requirements for the application while Chapter 5 wil give a deep insight into the development process.

2.1 General Requirments

The application needs to be accessible from all kinds of devices, especially mobile-phones, tablets, notebooks and desktop workstations. Web technologies, like Javascript, CSS and HTML, ensure that the user only needs a HTML5 compatible browser, which is already pre-installed on all the devices listed above. All relevant data needs to be stored permanently on a storage medium and has to be accessible at any given time. It is important to keep the data structured and it should be possible to change the database adapter and migrate data from one to another database system without any issues.

The application needs to be split into two segments, a client-application and a server-application. Both should run on as many platforms as possible without any constraints on productivity and scalability.

2.1.1 Server Application

The server application should be scalable and runnable on virtually any operating system and platform. Furthermore it has to provide a uniform interface for the client application. In this case it will be a JSON-based REST-like API. The data storage has to be a common SQL-Database like Mysql or PostgreSQL. It should also be possible to exchange the database during installation time.

2.1.2 Client Application

The server application as well as the client application need to be able to run on any platform, but on the same hand target end-user hardware with a display and a HTML5 compatible browser. AngularJS is used to build the single-page-application. Asynchrone API calls will communicate with the application server.

2.2 Server System Requirements

The server system has to be easy to set up and to maintain. The application resulting from this project also runs as a public service available to anyone, which makes scalability an important key point. A channel with over 16.000 feedback entries should be as responsive and usable as one with only 20 feedback entries. Not only the amount of data has to scale, but the number of requests has to be easy to handle too by simply adding more servers.

2.3 Channel Components

To create an easy to use application it is important to stay as minimalistic as possible. It is a channel's task to hold a specific quantity of feedback for a specific topic which is appointed by the channel admin. There always has to be one admin per channel. Users which need more than one channel will get separate login hashes for each of their channels. This makes sharing a company channel or community channel easier.

2.4 Feedback Components

The feedback component holds all information about a feedback which is mainly important for the user. Whereas information which is relevant for the administrator is added to the feedback additionally. Trough this design the application gets faster in delivering a set of feedback entries. Feedback always belongs to a channel and is never orphaned. If a channel gets deleted, all its feedback entries will be deleted too. So feedback can never exist without its corresponding channel.

2.5 Scalability

Constructeev will be available as a reference service and does not limit users by quotas. Thus it was necessary to build an application which would scale easily. The main requirement was set by handeling 16.000 feedback entries per channel without any performance loss or noticeable

delay. It should be easy to setup a cluster, without having to change or recode the session management. It is not set or limited by the project in which direction to scale. Here we have two options: Vertical scaling is when the node gets upgraded with a faster CPU and more memory while horizontal scaling will add fully fledged servers to the system.

2.6 Administrator View

The administrator view displays the same content as the user view. But the feedback and channels with information that is considered irrelevant to the user will be hidden from the user view. A small management panel will be added to the top for the administratior. Inside this panel the administrator can change some basic channel settings. For instance he/she can open or close the channel, hide or view all feedback entries and a button which can change the channels description is enclosed. Another feature which is only visible to the administrator is the so called "mark-a-feedback-read-or-unread-button". Through this button you can easily keep track which feedback entries you have already seen and worked on and which ones you have not yet read. A feedback can also be favoured by the admin and then will be automatically marked with a star and a label saying "Fav". Through flagging a feedback with said star and label you can mark it as one of your favourites.

Chapter 3

Related Work

This chapter will give an brief overview of some of the work which also specifies in coding feedback related programs as well a brief summary of how this bachelor thesis has been steered and influenced by others. There are plenty different feedback collecting tools. Most of them are for one single purpose, but all of them differ materially from this bachelor thesis.

3.1 Arsnova

Arsnova is an open source audiance-response-service and was developed by the "Technische Hochschule Mittelhessen". The aim of this service is to be a supportive application for lectures and seminars. It provides a live feedback channel for the pace of the lecture indicated by four emoticons. The happy emoticon represents "I can follow the lecture", the wink emoticon means "please be quicker", the surprised emoticon signals the teacher is progressing too fast and the sad smilie tells the lecturer that the students have lost the golden thread. With Arsnova students are able to ask questions anonymously. Those questions can be projected to the black board and thereby discussed with the lecturer. Teachers are able to create some questions beforehand which consist of multiple choice answers. The system then evaluates the student through the findings of those answers. The emerging results are aggregated by the system and then are accessible by the students for their own learing process and the lecturer can look at those results and thereby conclude how well his/her students have been answering the prepared questions.

3.2 Feedback Button

The feedbackbutton is a small widget button which can simply be added to any website. When this button is clicked, a pop-up window will be

opened and a form will be presented to the user, where he/she has to fill in his/her name, e-mail address, feedback category and comment. The sent feedback is stored in a database on the company server. This tool is a commercial closed source for which the user pays a certain fee every month based on the approximate amount of feedback which will be sent. For instance if one makes 50 feedback entries a month the person has to pay 6.00 $ per month, for 400 feedback entries that would make 12.00 $ per month, for 1200 feedback entries the fee would be 18 $ per month and finally it would cost 99 $ per month for unlimited feedback entries. But there is no way to customize the form or receive anonymous feedback.

3.3 Feedbackify

Feedbackify is a commercial company from the United Kingdom which offers a website feedback toolkit. Here it is not possible to create a feedback form without having to deploy a button on your own website. With this tool you can create and customized label and add your company logo to the online form. The feedback will be sent to the Feedbackify servers and the company promises prompt real-time feedback. A big drawback for the user is the fact that he/she has to expose his/her name or e-mail. On the other hand it is easy for the administrator to group feedback into sections or campains. It is also possible to to see where the feedback comes from. Another con is the listing of sensitive data like your customer's geographic location, browser history and information about the used operating system as well as screen size and the IP-address.

3.4 GetFeedback.com

GetFeedback.com is a representation of the most popular feeback survey tool. Those tools are exellent to get answers to a specific set of questions, but they do not not cover maladministration outside of the scope of their questions. Sometimes there are not as many selectable options available as needed or the survey takes too long altogether. The statsitic which will be generated by the GetFeedback servers is nice to present and to have a visualisation, but it has to be pointed out that this can not cover every users opintion, and the result can steered by the asked questions.

Chapter 4

Used Technologies

In this chapter a brief overview of the technologies used in this project will be given since most of them are not widely used or known. Firstly, there will be an overview of the functional language called Elixir and some open source frameworks and modules which are used to build the application based on this fairly new language. The remaining chapter will describe the more commonly used front-end technologies as well as the server platform in use.

4.1 Elixir

Elixir is an alternative language for the Erlang Virtual Machine (ErlangVM) which allows the programmer to write code in a more compact and cleaner way than through using Erlang itself. You write your code in Elixir and compile it to a BEAM-Bytecode and run it like any bytecode from Erlang in the BEAM symmetric-multi-processor environment (beam.smp). BEAM stands for Bogdan/Björn's Erlang Abstract Machine, which has tried to compile the Erlang Code in an early Erlang version to C and thereby gain a better performance. Nowadays BEAM is a performant and a complete "process virtual machine" and does not compile code to C anymore, because the effort is too high for the result of a minimal performance gain on modern CPU systems.
The language Elixir is an open source project started by José Valim and now is maintainedand developed by over 373 contributors. The source code of elixir can be found on the Github Repository at `https://github.com/elixir-lang/elixir`.
One major advantage of Elixir, it being a young programming language, is that the well tested, proven and hardened BEAM Virtual Machine is used for bytecode execution. A second advantage is that Elixir produces the same byte-code as Erlang, hence you can use Erlang libraries in Elixir and vice versa. Everything programmed in Erlang can also be

programmed in Elixir and usually the Elixir code is as fast and performant as its Erlang counterpart.
Elixir provides a huge set of boilerplates, reduces duplication in code and also simplifies some important parts of the standard-library. It provides some syntactic sugar and is a nice tool for creating and packaging applications.
Another impressive feature offered by Elixir is Metaprogramming. This process makes it is easy to extend and define the language itself and one can basically write code which then continues to write the code for you. With macros you can abstract and uniform Elixir code parts to restructure and abstract more complex code away from the programmer.

4.1.1 Phoenix

Phoenix is a web framework written in Elixir and it implements a server-side MVC pattern.
Phoenix is very similar to other modern web-frameworks like Ruby on Rails or Python's Django. The core developer team of Phoenix tries to provide the best of both web-frameworks through providing Elixir and the ErlangVM. This results in a simple and elegant code as well as a high performance application running over multiple CPU's or even Nodes in real time.
Let us look on how the Phoenix pipeline functions and which layers of this pipeline a single request will have to pass through:
After a request was accepted by the cowboy webserver it will be handed over to the **endpoint**, which will apply all necessary functions, like those from the plug-system, to the request before it will be passed into the **router**. The router will parse the incoming request, discover the assigned controller and provide some helper functions for routes, paths and urls according to the controller. A pipeline can be specified in the controller, which makes encapsulation of different scopes and APIs simple. The router will pass the request on to the assigned **controller**. In this segment the functions, also known as actions, are called to provide the main functionality. The controller's main task is to prepair data and pass it into views, invoke the view rendering and/or perform http redirects. After the **view** got all the data it then renders a template and provides a self-defined helper function which can be used in the template. The view acts like an presentation layer in Phoenix. **Templates** are precompiled and are the only string concatenations in the background, thus blazing fast.
Phoenix depends on some other projects in the Elixir and Erlang Ecosystem:

- **cowboy:** lightweight super fast Erlang web server

- **ecto:** a query and database wrapper
- **PhoenixHTML:** working with HTML and HTML safe strings
- **Plug:** a specification and conveniences for composable modules in-between web applications
- **poison:** a pure Elixir JSON library
- **Poolboy:** a worker pool factory
- **Ranch:** a socket acceptor pool

4.1.2 Cowboy

Cowboy is written in Erlang and is optimized to have a low latency period and a low memory usage. It uses Ranch for managing HTTP and socket connection. Cowboy is pure Erlang code and is easy to integrate on every platform without extra dependencies. This server provides a complete HTTP-stack with support for HTTP1.0, HTTP1.1, SPDY and web-sockets. The Cowboy project has a small and clean codebase, is well tested and provides a rich documentation.
In this project Cowboy is used to accept and forward the request to the Phoenix framework. Cowboy is also able to keep socket connections which can be used in Phoenix channels for real-time communication between the client in JavaScript and its Elixir counterpart.

4.1.3 ETS

ETS is an abbreviation for **E**rlang **T**erm **S**torage. ETS was built to hold a huge amount of data inside the Erlang runtime but constant access to the data has to be still possible. The ETS data is stored inside dynamic tables and each table is occupied by only one process. Those tables merely have a lifetime of one single process. If the process is terminated the data will be destroyed. Via message passing it is possible to request data from one process and give it to an other one. Message passing also works between nodes which concludes in all data being available throughout the whole cluster.
The ETS is used like a Redis database. It does run inside the ErlangVM and so not another dependency has to be installed and maintained in the production system. It is possible to run the whole app as a Unikernel [1] on Xen [2] or bare-metal.

[1] Minimal Operating System with the Application Code compiled into the Kernel
[2] Xen: x86 & ARM Hypervisor `xenproject.org`

4.1.4 Ecto

Ecto is not only a database wrapper it also provides macros for a nice query DSL inside Elixir, which means you do not have to escape queries. An adapter for the database you want to use is necessary. Ecto repositories are wrappers around the desired database. With Ecto repository, it is possible to create, update, destroy and drop custom queries. Changesets are brought by Ecto and provide filters and casting functionality for external parameters. Through this we get another security layer in front of the database.

4.2 PostgreSQL

PostgreSQL is a widely known open source relational database system. It runs on all major platforms like Windows, Unix deviates and Linux. It has full ACID (Atomicity, Consistency, Isolation, Durability) and supports all SQL 2011 standard statements like foreign keys, joins, views, triggers and varying other procedures. PostgreSQL was used in this project, because with the Postgres adapter for Ecto it is super easy to deploy Constructeev on Amazon web-services. There the Amazon RedShift Database can be used without a single line of code having to be changed.

4.3 Schema

In the schema shown in this section, the database normalization rules have been violated only for one single reason. This schema can be automatically transformed into a Riak DB format and then can be used inside the Erlang ecosystem. Through changing the Database from Mysql/PostgreSQL to Riak this process is no problem and can be easily achieved.

4.4 AngularJS

AngularJS is a JavaScript framework for building client web applications with dynamic views. This framework is used to build HTML single-page-applications and supports a full model-view-controller-pattern and has its own router-module for managing views. Data inside the controller and model are usually two-way binded. Any changes in the view are visible inside the controller and vice-versa. AngularJS is used for the whole front-end funcitonality. Routing and views are done by an Angular-router replacement called ui-router. Ui-router allows to route

and manage nested-views and and nested-states, which are not completely performant and easy with the standard router that is shipped with Angular-JS.

4.5 Semantic-UI

Semantic-UI is an open source development framework for a responsive design. Natural language is used to describe the class names of HTML containers and it provides a nice and clean codebase for theming and adding customary extensions.

4.6 Server-Platform

This section describes other software and hardware used for developing and testing the Constructeev system. Basically the system should be runnable if a network stack is available and the minimum set of both external dependencies (ErlangVM and Postgres) is fullfilled. It is also possible to use the RiakDB and a small kernel with ErlangVM. The complete code of Constructeev will be compiled into the kernel.
The application was developed on a Macbook Pro 15,6" running with Macintosh OS 10.11 (El Capitan) with an Intel Core i7 (r) 4 Core Processor with Hyperthreading up to 8 "virtual" Cores. SSD powered, where Postgres mySQL and Riak had their persistent data.

4.6.1 Tested Hardware

In this section some hardware combinations will be shown. Further an explanation in what ways the software has been tested and developed will follow. Benchmarks will be described in chapter 7. Some of them are rather specific like for instance the ARM-Server platform.

University of Innsbruck

The Database and Information System Group from the University of Innsbruck created and granted access to a virtual CentOS7 machine with a single core virtual x86/amd64 CPU core with a clockrate of 2666MHz and 30GB SSD based storage. The application can be installed via the included shell script without problems.

Scaleway ARM Server

Scaleway ARM Server hosting platfrom. It offers some interesting services. Bare Metal ARM Servers, which are a custom build system on a chip with 4 ARM Cores (1.6GHz Clock) 4GB RAM and 50GB SSD disk,

which offers only one IPv4 adress per server and 200Mbit/s external and 1Gbit/s internal bandwidth. The 1Gbit/s internal link is perfect for the communication between the cluster nodes.

Custom AMD 16 Core two CPU Server

This custom AMD Server has two CPU Sockets (both filled with the same Opteron Server CPU) and is the test system for the **s**ymmetric **m**ulti**p**rocessing (smp) functionality in Constructeev. SMP Systems share the main memory with all CPU's, hence all devices and common resources are available to all CPUs. Those multi-processor-systems work under one single operating system. Therefore you only have to start a single instance of the BEAM/ErlangVM. Elixir will discover the available cores on the installed CPU and spawn a single OS-process on every CPU and one OS-thread per core. Message passing and auto-discovery of the ErlangVM will glue those CPUs together to a single 16core ErlangVM node. The hardware specification of this server are two 8-core Opteron 6320, 32GB main memory, 12TB ZFS Raidz1 storage with a 256GB SSD cache, 10Gbit/s network interface card with 1Gbit/s internet connection. FreeBSD 10.1, an unixoid operating system was installed underneath the ErlangVM

Chapter 5

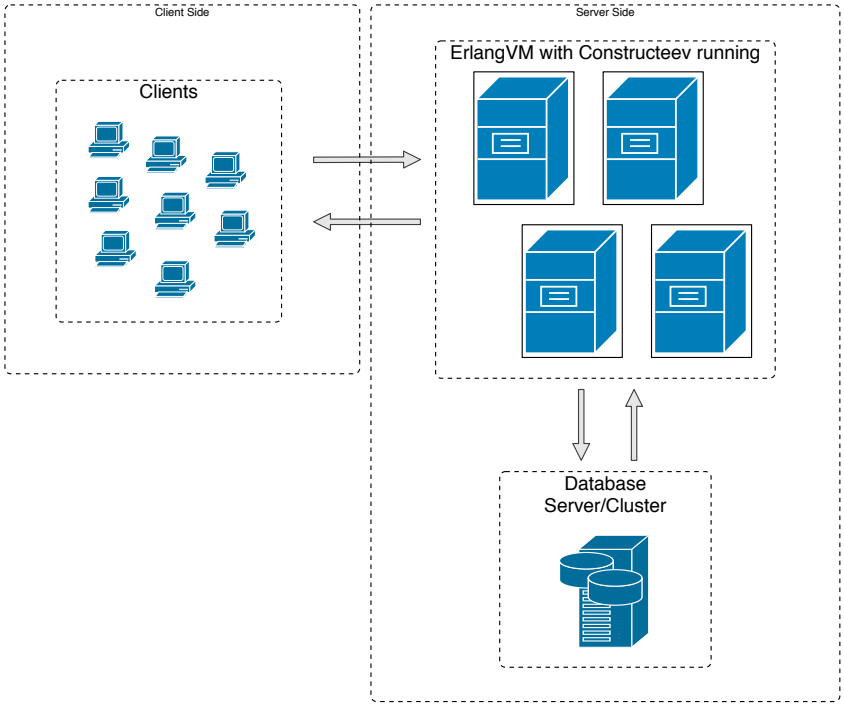
Implementation

In this chapter, the explanation of the implementation of the Constructeev application will be given. First of all there will be an overview of the whole system, divided into the back-end written in Elixir and the front-end where Javascript and CSS have been used. Further on a deep dive into how it is possible to scale with multiple nodes and why Elixir has been chosen as programming language will follow. As already described in chapter 4 the application has not been developed from ground up, therefore some dependencies to other open-source projects from the Elixir and Erlang community exist. Those modules are used especially for accepting and routing HTTP calls, escaping springs in HTML-safe responses and accessing the databases and file systems. In this project an external API is also requested for the channel and user avatars. All used modules are available on the Elixir package management system called "mix" and will be installed automatically with 'mix get.deps'. The process of evaluating those external dependencies costs time, but is important for building a distributed and scaleable system. It is highly important to confirm that non of those dependencies add a bottleneck to your application. There will be more about the topic of scaling in chapter 7

A graph-diagram will show the schematic setup of all microservices and how they work together.

This single-page-app typically loads at the base of the first request, which includes the base structure of the application. Everything else will lazy loaded for more responsiveness. The web-app communicates, in a specific way for modern web applications, via a single HTTP/1.1 opened connection with a REST-API with JSON being the payload format. JSON (JavaScript Object Notation) is preferred over XML, because it is natively supported in JavaScript and you can work without having to extra parse the servers response. JSON is also less redundant, hence it does not open a field with '¡field name¿' and closes again with

Figure 5.1: System Schema



almost the same string '¡/field name¿'.

5.1 Back-end

The back-end is written in Elixir, runs inside an ErlangVM (see section 4.1) and uses cowboy for accepting HTTP requests. The Phoenix framework is used to route and apply functions to the request and respond. Data is stored in a database which supports the SQL 2011 standard (like mysql, mariadb, PostgreSQL, Amazon Redshift etc.) or the RiakDB (recommend in distributed mode). For more Information see chapter 7).

5.1.1 Selection of the proper programming language

The first prototype was built in Ruby and used the Ruby-on-Rails web framework. Many modern websites are built with Ruby-on-Rails, Github being a popular example. [ROR] [Doe]. While working on a company project, Elixir was discovered and Chris McCord, the creator of the phoenix framework presented the new version of this framework on Eixir-Conf2015. The language concept of Elixir and the combination of the

Phoenix framework seemed perfect for this project. Also the ability to keep a state for something, like a chat over multiple nodes, is perfect for some future feature implementation. [1] After rebuilding the prototype within a few hours, the first impact of the thin web framework and the fast web-server was revealed. Requests which took 3-6ms in Ruby-on-Rails now took 700μs to be on the server. The developing process was a pleasure through Elixir and the ErlangVM is known to be a stable platform. The decision to switch the programming language fell right then.

5.1.2 Choosing of the Database and Schema

While the back-end was under development, a set of databases was being tested. After making a few performance tests of different databases as well testing varying data-structural designs it was decided to use mysql since it showed the best results. The first implementation was using Nested-Sets which did not scale, because inserting into the Nested-Set was terribly slow. Inserting into a 10.000 entity channel took about 1 - 5 Seconds. After finishing the back-end with a very simple schema, Constructeev was able to accept 2.000 feedback entries per second on a single 4 core ARM server. Using the Nested-Set the database would have a long queue to work on.
In this version of Constructeev the database schema had an one-to-many relation between channels and feedback entries. Each feedback can also has many feedback entities underneath with will represent replies. Channel and feedback will be related to their properties with a one-to-one relation. A feedback property does only store additional information like the read or unread state or if the feedback has been favorited.
The Nested-Set had one huge advantage over the regular parent-child schema. Counting the children inside a Nested-Set is really easy and can be done with only one calculation [2]. Within this schema, which has more performance range, we would have to count every child with the SQL count() function. This call has a linear time complexity of $\mathcal{O}(n)$. Linear time complexity is good, but with an extreme large number of n this takes too much time.
This problem can be bypassed with a write, memory and time trade-off. Every time a feedback is inserted, the channel row will be updated in relation to the feedback. In this case we have one additional row updated, as a result we get a correct count of feedback entries instantaneously with every channel select, Same with replies on feedback entries. The parent feedback will be updated. The server allows a depth of four

[1] more in conclusion

[2] Subtract the left index of the right index is the number of children if the Nested-Set has not optimized indices.

replies, which means, that in a **worst case scenario** the maximum complexity for the correct count is a $\mathcal{O}(\log_2 n)$ plus row update time, which is less than $\mathcal{O}(\log 2n)$. Furthermore we will receive much more reads than writes, especially within the hierarchy.

In one of the last month of the developing process, the database had to be changed again. MySQL was replaced with PostgreSQL, hence we strictly used the SQL 2011 standard and only had to change the Ecto adapter. The only reason for this change of plans was for Constructeev being able to integrate into the Amazon web-services data-centers with ease. Amazon's highly scaleable SQL database, called Amazon Redshift which is a modified PostgreSQL, can be used with any standard PostgreSQL database driver/adapter

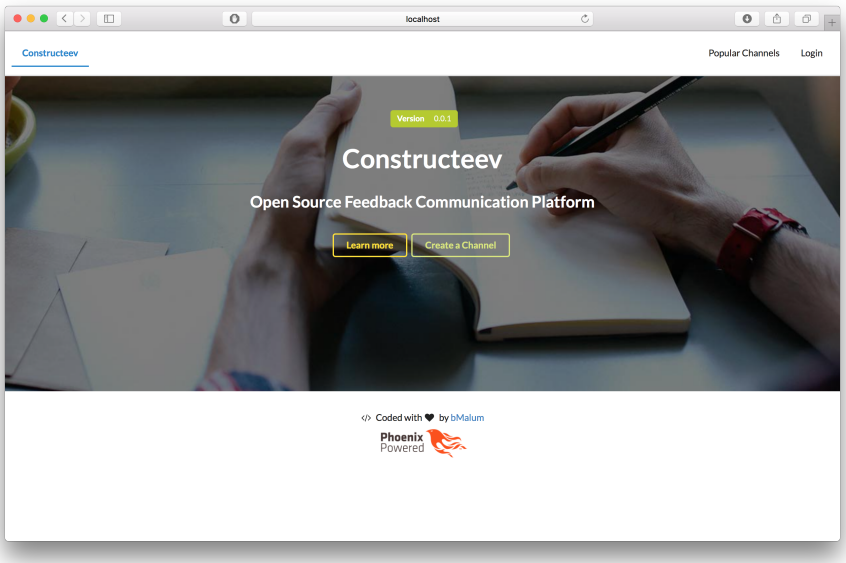
5.2 Phoenix Pipelines

The back-end provides two highly optimized pipelines, because a single ErlangVM is providing binary large objects (blobs) and has, in relation to the blobs, tiny JSON-Object responses or may even only result in a HTTP header with a status code. A single pipline is not optimized for both large and small responses. Scopes are built around all pipelines. All routes with the '/api' prefix are processed by the so called API-pipeline which is optimized for sending JSON responses. The other pipleline has access to the file system, caches last recently used blobs, like images, fonts, HTML templates, precompiled cascading style sheets and JavaScript files. This line then delivers them to the client.

5.3 Front-end

The front-end is written in JavaScript and uses AngularJS, an open-source web application framework used for developing single-page-applications. By providing a framework for client-side-model–view–controller it aims to simplify both the development and the testing of web applications. With AngularJS it is possible to build feature rich internet applications. The single-page-app is bootstrapped by a very small HTML template, which contains a, by the server pre-processed, cascading style sheet, the minified AngularJS Framework and those extension/dependencies.

Figure 5.2: Startpage Constructeev



Constructeevs client side is divided into several main parts:

- MainView
- ChannelList
- ChannelDetail
- AdminView

Every view can have multiple models and each of them has a own factory which handles the API calls and prepairs the data for the controller. All of them communicate with the server via a REST like HTTP-API using JSON as dataformat. The API is not a full REST API because sometimes a concurrent state is represented. In this project's case this will be the likes from every channel and the up-voting and down-voting of a feedback. REST APIs solves concurrent problems were the data arriving last is the valid one. This does not work in concurrent problems like for instance when one tries to count a click. It is easy to reproduce those errors:

```
bash$: curl http://localhost:4000/api/channels/1
{
  "data": {
    "updated_at": "2016-04-22T08:41:03Z",
    "slug": "luntpnbs",
    "name": "Zaam-Dox",
```

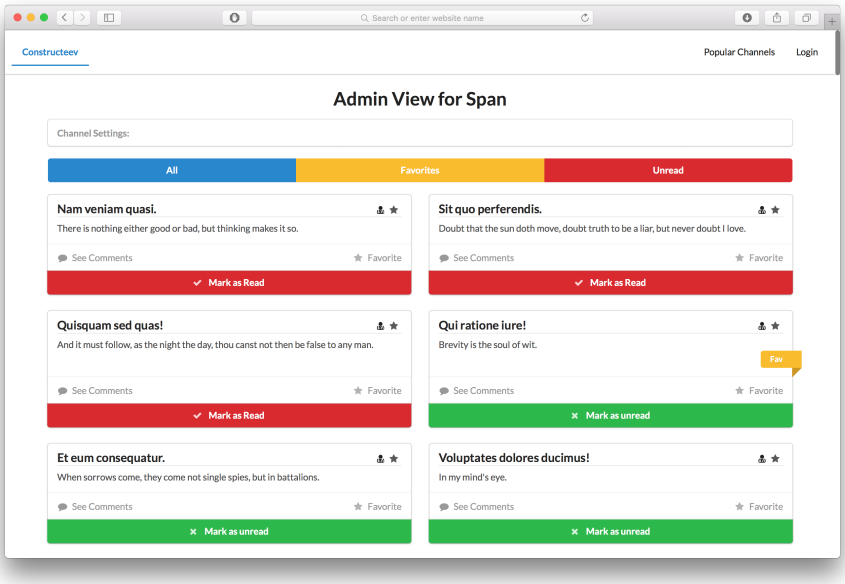
```
    "likes": 198,
    "id": 1,
    "feedback_counter": 101,
    "email": "ewald.lehner@little.net",
    "description": "Brevity is the soul of wit.",
    "like_url": "/api/channel/1/_like",
    "channel_hash": "8b83774b8932d5ef99c6af660b3f8461"
  }
}
```


The JSON represented above is the state of the data in the browser from the user as it is at that exact moment. Now let us call an other client the "like_url" 50 times. The JSON inside the user's browser is the same state as shown above meanwhile the client wants to like the channel and produce a PATCH-request with following data:

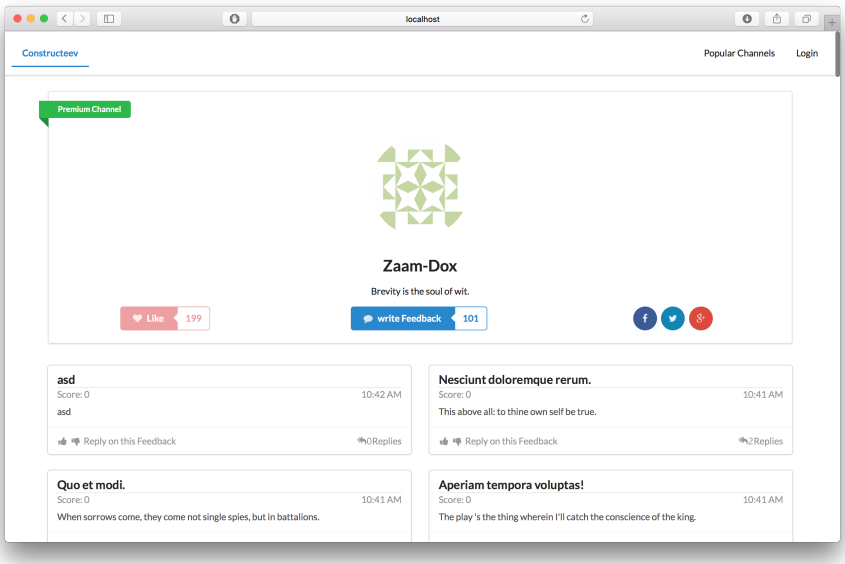

```
{
  "data": {
    "likes": "199",
  }
}
```


The client pushes the value of 199 (198+1) likes to the server, which will accept and write down 199 likes in the database, instead of the 249 likes which would be the accurate value. 50 of the 249 likes are dropped in a so called "race conditon". REST does not fit here, but instead a RPC (Remote Procedure Call) has been added. This action-URL will be delivered with the API, therefore the RPC-URL can be changed without the client having to be notified or changed because the client does only call the like_url which is being provided inside the JSON-Object.

5.3.1 Admin View



5.3.2 Feedback View



5.4 SDLP - Simple Double-Like Prevention

Each Channel can be liked by anyone. No registration or log-in is required. With a normal implementation a single user could push the "Like-Button" as often as he/she wants to. This would result in the feedback-giving-person being able to boost his/her own feedback. Also the admin could simply like all positive feedback so there then seems to be more positive feedback altogether. To prevent this from happening a simple double-like prevention has been added to this project. At the beginning it is only client-sided and could be outsmarted with a bit of know-how. The single other rather annoying solution would be to add captchas to every single like-request. HTML5 compatible browsers do have a client side storage module called "local storage". It is a key-value store inside the browser of the accessing device. This local storage was designed to hold a large number of keys without affecting the performance of the website. Also the five megabyte limit of a cookie can be over skipped with this technology. The not sending the local storage to the origin is another important feature. The storage offers a new database-bucket for each origin (domain or protocol). Only web-pages with the same origin can work in this unique bucket. SDLP uses the local storage feature and keeps track on which feedback or channel the client has liked. The channel is the key and true or false equals the value. If the client receives a click on the like button, the ID of the feedback-channel will be added to the local storage. On every channel-page load, the client has to look up inside the local storage, wether the channel has been liked or not. If the channel has already been liked, the like button is disabled and can not be clicked again. This technique is also used with feedback up-votes and down-votes. In this case the browser also checks if the channel has been liked but instead of disabling the buttons, it will only offer the reverse action. For example: An already upvoted feedback can only be down-voted by the same user.

5.5 Distributed Mode

The distributed mode is in use when a single Phoenix application is running on multiple servers. Here it is not important for the server hardware to be identical, but to be in many differing distributed frameworks. You can use different hardware types since every process is abstracted from the BEAM virtual machine. The main goal is to set up a distributed Phoenix application to get more physical CPU-cores thus the application can work on more cores parallelly.
In distributed systems there are some fallacies which would allow a full-blown horizontal scaling.

1. The network is reliable.
2. The latency is zero.
3. The bandwidth is infinite.
4. The network is secure.
5. The topology does not change.
6. There is only one administrator.
7. There is zero transport cost.
8. The network is homogeneous.

[Deu] In this project the network topology is hidden by the ErlangVM so we assume it is homogeneous and secure. Per definition there is only one administrator and transport costs are almost zero nowadays. Only points one to three can be the bottlenecks for horizontal scaling. Network can not be reliable. Sometimes network interface cards fail or some more trivial routing of the internet service provider fails, in some cases the network driver can crash.[Her07]
Sending messages from one node to another takes time, even if they reside in the same rack and are connected directly to each other. This will slow down the system. If this happens it is important to keep latency as low as possible.
In bench marking a 1Gbit/s connection has been used (for bechmarks see chapter 7) and thereby we were strictly limited by connection-speed. When an open connection exists, passing messages, updating code, passing requests to other servers and dealing with database requests must be possible under all circumstances. With increasing the server's performance more messages from the ErlangVM are sent to each node for synchronization. Also ETS will be more distributed for the session's information and more messages are sent across the network to find the necessary information.

5.6 Sessions

Sessions information is a kind of state which should be kept within the period of a session. Cookies are often used to store session information, because these cookiesWe were sent to the server with every request . The server needs match the collected information to the session or state he connects to a specific user. Most frameworks keeps session-information

inside the main memory of the application. This has one simple reason: Access-time inside the main memory is to slow.
As described in section 5.5 through adding some servers the application tries to scale as horizontal as possible. When adding a machine, the main memory is not accessible to the other servers. With the standard session management we lose state as well as the session while the clients are redirected to the new server. A common way to solve this is writing session information into Redis, which is a distributed in-memory data structure storage.
Constructeev could also use Redis as a distributed in-memory-store for collecting session data, but adding another dependency is unnecessary when the ErlangVM offers almost the same feature. In this project ETS (Erlang Term Storage) is used for storing session data. The only feature, which ETS is missing, is persistent information storage on hard drives. Persistence is not required for storing session information. If the hole system went offline (eg. power outage) the ErlangVM will be restarted and a new secret key will be generated. All cookies generated before the power loss are now invalid, because the secret key has changed. Therefore we do not need persistence, because the security function will invalidate all of the persistent data. In this specific case Constucteev is more performant in recovering form a major issue, because we do not recover useless data.

5.7 Testing

The server side code has been tested with ExUnit, a Unit test framework for Elixir. The API has been tested with the emacs restclient and the client side framework has been tested with Protractor. Protractor is a test framework that is running on a real browser, which interacts with the user world. Travis-CI was used for the API and front-end testing as a continuous integration service. This service allows the running of all tests from different commits which are then pushed to github.com. The Travic-CI did not support Elixir or Phoenix applications therefore a custom configuration file has been added to the repository. Now Elixir could be supported and the configuration file was updated to a cleaner and smaller file:

```
before_install:
  - git submodule update --init --recursive
language: elixir
after_script:
  - MIX_ENV=test mix deps.get
  - MIX_ENV=test mix inch.report
```

Chapter 6

Usability and Responsive Design

Usability is a very important aspect of an application. Therefore it is crucial to think about how the functionality and features of the application can be displayed and presented to the user. A lot of effort went into preparing and displaying information in a clear and well-structured way. Sometimes features have to be limited to provide a better usability. This concept occurred in the project and will be described in the next sections.
Furthermore the application has to be useable with all devices, which obviously do not all have the same screen size. The application has to fit on small screens like mobile-phones, medium sized screens like tablets as well as small notebooks and large screens mostly used as a desktop workstation. All features and informations have to fit on any screen-size, here a responsive design framework helped to develop this function of the application. Via example the next chapter will describe how useability can influence an application's design. At the end of this chapter a overview of the Semantic-UI framework and how it has been used in this work will be given.

6.1 Usability of the Feedback Communication Feature

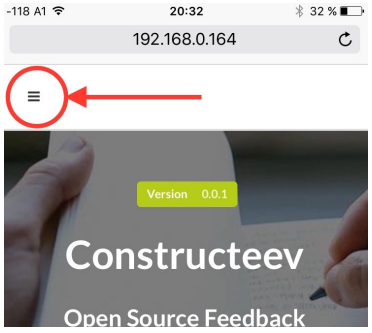
In this section the usability of the feedback communication feature will be shown and it will be described how it influenced or limited the features of the application. Usability is more important for a web-application than the range of a feature. If the user interface is not easy to use, those features will not be needed at all.
In Constructeev it is possible to answer a feedback discussion message. This allows the building of a recursive order of replies to replies. The API

also supports the answering of said message. Regrettably the usability of this dicussion forced the application to show the argumentation to a feedback in a flat chat. A flat chat is a chat where no hirarchy exists and the messages are ordered by time, like in an IRC chat protokoll. [JO93] The following images will illustrate through examples how hirarchial messages limits the usability on mobile devices.

6.2 Responsive Design

Responsive design is a technique used for building websites or web applications. It provides an optimal experience through covering a wide range of display sizes. A framework, which already has the ability to be responsive, is being used. Semantic-UI is easy to use and has many pre-designed collections and modules which support the developer in building a responsive design. The resulting frameworks need to be used with care, because it does not mean every idea is possible in reality and every feature is exactly as usable in small-sized view as it is in the full-sized view. Some parts have been customized for Constructeev, especially features about the mobile navigation menu. The mobile navigation menu, had to be redesigned for small screen views. In this special case a horizontal menu does not work anymore, or simply consumes too much space on top. If the menu is on the top of the side, the user has to scroll to see the content. This would happen every time a new view was being used. Since the menu is not used constantly, it is acceptable to hide it and provide a commonly known icon, which when activated shows the menu again. Many web-designers still avoid using the so called "Hamburger Menu", because it is inefficient in displaying menus. The user has to open the menu before he/she can use it. Bad discoverability and operating system design conflicts are the main hindrances why designers choose to use other design patterns.[1]

Figure 6.4: Hamburger Menu

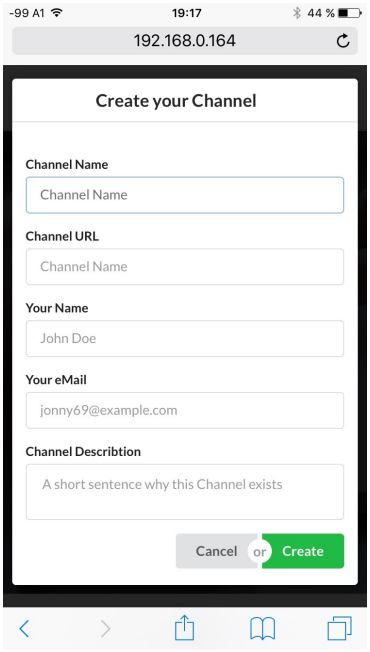


Usability tests with different occupational groups [2] have clearly shown, that it depends on how the application workflow has been designed. All of the volunteering application testers were happy with the "Hamburger Menu." The placement of the menu-button is consistent in the same left corner on every page. The user dose not have to search for the correct button to click, since there exists only this single menu-button. Every workflow inside the front-end between the different actions and views has been designed thoroughly. The concept was to provide a fluid flow and not to force the user to move the cursor/finger too much. The user also does not necessarily need the menu to complete different actions.

[1] Source: https://lmjabreu.com/post/why-and-how-to-avoid-hamburger-menus/

[2] Subjects were: medical doctors, nurses, welding engineers, documentation assistants, law assistants

Figure 6.5: Constructeev Modal



The same concept has been applied on the forms used for user input. The user input is not opened on a new page as in many other web-applications. Instead of redirecting the user to a new page, a modal window is opened. The advantage of this design is, that the user does not loose the orientation inside the app on a small screen.
Another important matter for designing a user friendly, responsive application is providing a function to undo a unintended action. This function should not be hidden, it has to be highly visible to give the user the feeling of safety and cancellation.
Using colored buttons makes it easier to navigate on the screen. However "cancel-buttons" are never colored red in Constructeev. The red buttons are reserved for delete/destroy actions. The "cancel-buttons" are intentionally colored gray, to emphasize that they are not irreversible.

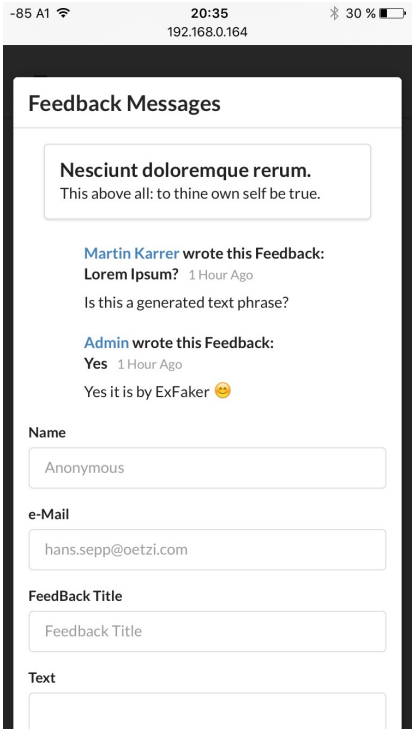


Figure 6.1: Constructeev Flat Chat

Figure 6.2: Reddit Conversation

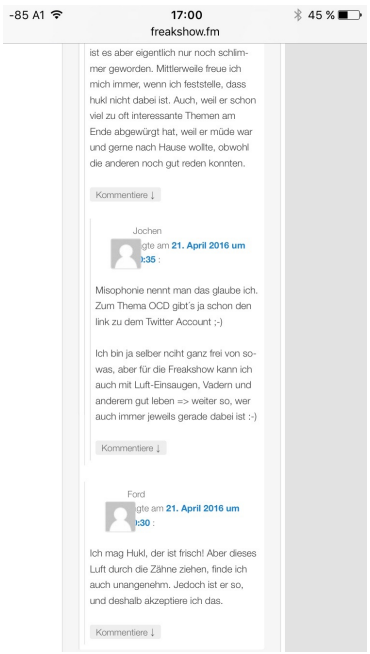


Figure 6.3: Wordpress Comments

Chapter 7

Performance and Scalability

In this chapter the benchmark-process will be described. This process allows the testing of the scalability of the application, but only to a certain degree. The test system can not represent real-time traffic. Altogether these benchmarks let us see if the application is even able to handle a large amount of requests per second and if it gains a speedup while adding more nodes to the ErlangVM.
Performance in general is a amount of work accomplished by a computer system. [1] Work can be split into the following three main sections:

- Response time (less is better)
- Throughput (more is better)
- Utilization of resources (less is better)
- Processing speed/requests per second (more is better)

Benchmarking an application can be difficult. There are quite a lot side-effects which can affect the result of the performance measurement. You will always have to share resources with the operating system and other processes. Resources like disk input/output or high CPU-consumption of other processes can affect the result negatively.
To minimize the side effects a Linux Distribution has been used to run the benchmarks. Every deamon or process which is not essential has been stopped before the test and the latest Erlang [2] environment and PostgreSQL [3] version has been installed.

[1] Definition: https://en.wikipedia.org/wiki/Computer_performance
[2] OTP 18.3 - `https://www.erlang.org/news/101`
[3] 9.4 - `http://apt.postgresql.org/pub/repos/apt/pool/9.4/`

Separately to the server a client has been set up with the benchmark tools. The clients are located in the same datacenter, are connected to the same server-rack via one router which functions as a hub with a 1Gbit/s connection to the server. A direct connection was not possible and a router/switch was necessary to be able to test with more than one client or server. The clients also need some preparation, because to save money a single client has been optimized to be able to make as much connections per second as possible during a request. The benchmark software has not been coded by myself, thus I had to optimize the system.
To simulate as many concurrent users a possible the benchmark software has to open hundreds of connections to the test server. Therefore some operating system threads are forked and used to open connections. This procedure uses a few file descriptors. The operating system has a hardcoded limit, which means how many file descriptors can be used by the whole system to prevent misuse. We hit this limit very soon so we had to set the limit higher.

```
root@bMalum:~# ulimit -n
10000
root@bMalum:~# ulimit -n 80000 # Setting the new limit
80000
```

All benchmarks in this chapter are done with a open source tool called "siege". Siege supports basic authentication, cookies, HTTP, HTTPS and FTP protocols and other than with certain tools you can mix GET with PUT/POST and PATCH requests while being gentle with the CPU.

```
siege -c 200 -b -i  http://127.0.0.1:4000/api/channels/$arg
```

Where -c is the number of concurrent requests and -b flag does not allow any delays between the requests. The last argument which is passed over to the program is the test URL.
Previous tests during the development process have also been done with foreign tools.These tools had some disadvantages over the chosen "siege" stress test tool. Except for the ApacheBench all of them did use a lot more CPU-cycles and were only able to do less than half of the requests per second. Another really interesting discovery made during development mode was, that the logger application, which is used to log and debug the application, crashed and then did not recover. A bug-report has been sent to the corresponding developer. "WRK" is one of the benchmark tools which used up to 10 times more CPU-power than the ErlangVM used to process and answer the requests. This would result in the necessity of renting more clients to exhaust the server.

7.1 Read Performance

The test system was an Ubuntu based 14.04 ARM server with a 4 core 1.6GHz CPU and 2GB main memory as well as a 1Gbit/s network interface card. A custom Marvell Quad-Core ARMADA XP Series SoC was used too.[Mar12]. After installing the dependencies, the database has been filled with 1000 channels. Each channel holds about 1600 feedback requests. After the first and recurring benchmarks the server will be restarted so every cache (database, ErlangVM and operating system caches) are cold. The benchmarking software was started on another server. It was linked to the application-server with a 1Gbit/s connection.

Listing 7.1: read benchmark by siege

```
1  Lifting the server siege...
2  Transactions:                  32706 hits
3  Availability:                 100.00 %
4  Elapsed time:                   6.37 secs
5  Data transferred:               8.56 MB
6  Response time:                  0.04 secs
7  Transaction rate:            4234.19 trans/sec
8  Throughput:                     2.34 MB/sec
9  Concurrency:                   91.76
10 Successful transactions:       32706
11 Failed transactions:               0
12 Longest transaction:            0.10
13 Shortest transaction:           0.00
```

Siege sent 32,706 requests to the server, all of them were answered by the server. On line 7 of Listing 7.1 we can see that previously 4,234 transactions per second were processed by the server.
To ensure the correct results the same test has been done with the ApacheBenchmark. Before testing the system again with ApacheBench the server was rebooted to clear all operating system caches and database caches. It showed almost the same request-per-second-result but a increased concurrency level. Thereby it can be assumed that both benchmarks are equivalent to each other and as correct as benchmarks can get.

```
1  Server Software:
2  Server Hostname:        127.0.0.1
3  Server Port:            4000
4
5  Document Path:          /api/channels/{arg}
6  Document Length:        292 bytes
7
8  Concurrency Level:      100
9  Time taken for tests:   232.579 seconds
10 Complete requests:      1000000
11 Failed requests:        0
12 Keep-Alive requests:    990049
13 Total transferred:      555761176 bytes
14 HTML transferred:       292000000 bytes
15 Requests per second:    4299.62 [#/sec] (mean)
16 Time per request:       23.258 [ms] (mean)
17 Time per request:       0.233 [ms] (mean, across
      all concurrent requests)
18 Transfer rate:          2333.56 [Kbytes/sec]
      received
19
20 Connection Times (ms)
21                min  mean[+/-sd] median   max
22 Connect:         0    0   0.1      0       6
23 Processing:      4   23   3.1     23      86
24 Waiting:         4   23   3.1     23      86
25 Total:           4   23   3.1     23      86
26
27 Percentage of the requests served within a
      certain time (ms)
28   50%     23
29   66%     24
30   75%     25
31   80%     25
32   90%     27
33   95%     29
34   98%     31
35   99%     34
36  100%     86 (longest request)
```

Figure 7.1: ErlangVM Observer

Erlang has a built in module for observing the state of a single ErlangVM node. This module is called "Observer" and can be started inside the ErlangVM through the following command:

```
1 :observer.start()
```

Observer is a graphical tool for observing the health and state of the chosen ErlangVM. In the screen-shot shown in figure 7.2 the real time scheduler utilization as well as the memory and input/output usage is displayed. We can see that none of the schedulers were at 100% usage. That lets us come to the assumption, that the server would be able to handle even more requests per second. The test had to be run again with additional clients.

Coordinating Clients

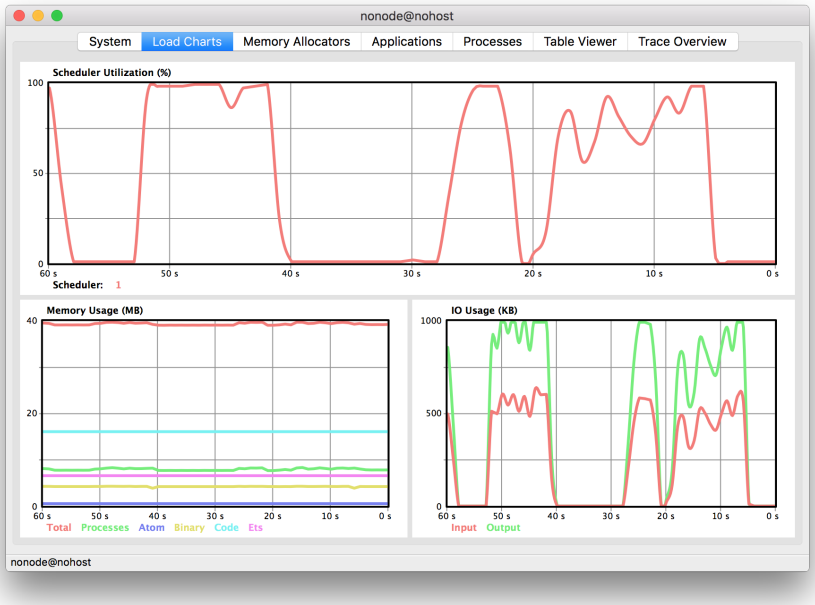
The problem in solving this, was that we needed to run the same benchmark on multiple client-servers at the same time to loot out the limits of the server. In this section the standard reference value is how many request a server can handle in one second. So we can start multiple independent benchmarks at exactly the same time on different clients. The weak point in this case is that we can not synchronize the time on all servers, instead we must force the server to align its system clock to a NTP-server system clock, which leaves plenty of room for errors. The

Table 7.1: Read Performance

Concurrency	Single Client	Multiple Clients		
	-	Client #1	Client #2	Sum
100	4180 req/s	2875 req/s	2887 req/s	5.762 req/s
200	4269 req/s	2828 req/s	2904 req/s	5.732 req/s
250	4462 req/s	2908 req/s	3053 req/s	5.961 req/s

PTP (Precison Time Protokoll) can guarantee correctness under 100μs inside the local network [Pro]. Before every timed benchmark the system clock gets synchronized and then the tool starts the stress test on a given time for a specific period of time. Hence the tool is started at (almost) exactly the same time and runs for the same amount of time. We have more concurrent users and can simply sum up the request per second of every client.

Figure 7.2: ErlangVM Observer



The Erlang Observer shows us, that the server using while beeing under siege 100% scheduler utilization, which means we hit the maximum.

7.2 Write Performance

Table 7.2: Write Performance

Concurrency	Single Client	Multiple Clients		
	-	Client #1	Client #2	Sum
100	2006 req/s	1351 req/s	908 req/s	2259 req/s
200	2049 req/s	1385 req/s	1129 req/s	2514 req/s
250	2141 req/s	1424 req/s	1252 req/s	2676 req/s

To test the write performance, the benchmark tool has to be able to send PUT or POST requests. For the write performance test the same experiment setup as shown in section 7.1 was used. The database has been filled with the same data too. The system has been rebooted to clear all caches of the operating system (file-system cache) and from the database. Another preparation which had to been done for this test was to create a text-file with JSON-data which will be passed with the POST requests to the database. This file was generated by a small elixir shell script and used Elixir Faker[4] to generate the test data. The benchmark can be called with this command:

Listing 7.2: Siege Post Data

```
1 siege -H 'Content-Type: application/json' 'http
    ://localhost:4000/api/channels/{$1}/feedbacks
    /' PUT < /home/bMalum/postdata.txt'
```

In Listing 7.2 the siege is started with some new parameters. First the content type has been set to "application/json" to signal the server that the payload is JSON data. The second argument is the tested URL with one parameter to insert data into different channels and the next argument sets the HTTP request type to 'PUT'. The data file is piped into the program and will send the generated data to the server. The clients have been prepared and coordinated with the same like in section 7.1 in this test.

7.3 Distributed Performance

To measure the distributed performance the same client setup as in section 7.1 and 7.2 has been used. The server setup does differ in this case and there has been more preparation work. For this performance measurement two different setups have been built. The first setup was for using two servers provisioned with a Linux operating system (Ubuntu 14.04 LTS) on a Scaleway C1 unit (4 times 1.6GHz ARM Core, 4GB main memory and 50GB SSD and 1Gbit/s network interface). A third

[4] https://github.com/igas/faker

server has been installed with the same operating system, but only a database with fake-test data. This server is simply used as a database server. On the remaining two servers Erlang 18.3 and Elixir 1.2 has been installed. The test servers have not yet been set up and had to be configured to run in a cluster. This has been done by adding a system configuration to the ErlangVM environment.

Listing 7.3: sys.config

```
1 [{kernel,
2   [{sync_nodes_optional, ['n1@10.0.0.143', 'n2@10
    .0.0.144']},
3    {sync_nodes_timeout, 1000}
4   ]}
5 ]
```

In the sys.config above two nodes are configured to sync and run as a distributed cluster. A timeout has been set to one second. If the node's clock will drift off more than one second and can't be re-adjusted or the network is not available for longer than one second the second node (n2@10.0.0.144) will stop trying to complete its task an try to reconnect to the cluster in 60 second cycles.
Each node can be started with the following command:

```
1 root@titanium$ elixir --name n1@10.0.0.143 --erl
    "-config␣sys.config" -S MIX_ENV=prod PORT=80
    mix phoenix.server
```

This has to be done with the other server too. The parameters have to be modified though.

```
1 root@titanium$ elixir --name n2@10.0.0.144 --erl
    "-config␣sys.config" -S MIX_ENV=prod PORT=80
    mix phoenix.server
```

The single application is running on both machines using all the resources from both bare metal servers. Both servers are listening to port 80 (standard port for HTTP connections). We can send requests to `http://10.0.0.144:80` but we cannot guarantee which node will process the pipeline, only which node did accept the request and send the result. Therefore it would be okay to publish one IP-address from a single node of the cluster.This tactic is not to be recommended. The best way is to publish all IP-addresses of all nodes. This can be done by creating a so called "Resource Record Set" within the DNS and set the order of the addresses to cyclic. When the domain is resolved, one of the node IP-addresses will be randomly used and all traffic will be distributed to all nodes. A Resource Record Set of this setup would look like this:

Table 7.3: Distributed Performance

Concurrency	Single Client	Multiple Clients			
	-	Client #1	Client #2	Client #3	Sum
100	4096 req/s	3189 req/s	2994 req/s	3218 req/s	9401 req/s
200	4207 req/s	3212 req/s	3189 req/s	3094 req/s	9495 req/s
250	4356 req/s	3298 req/s	3247 req/s	3129 req/s	9674 req/s

Listing 7.4: Resource Record Set

```
1 appname.domain.tld.    180   IN   A   10.0.0.144
2 appname.domain.tld.    180   IN   A   10.0.0.143
```

With the data from table 7.3 the distributed mode can achieve in processing "request per second" by a speed up factor from 1.62 to 1.68. Which is a great result for scaling a distributed system [97810]. Of course a higher speed up factor could be achieved with multiple single systems like NodeJS. Projects where stateful connections do play a role can't be carried out so easily with NodeJS. [Gui]

7.4 Chapter Summary

In this chapter performance measuring has been described in detail. From the gathered data we can conclude with certainty that Elixir can be used to build a state of the art web service. Most community components written by the Erlang association are built to scale and do not result in smaller bottlenecks as if a similar software written by associations using a different programming language is employed. It is possible to build a stable an reliable server, which is not only fast in single mode but even performs well when distributed.

Chapter 8

Conclusion and Future Work

The developed application shows that it is possible to build a scalebale web service for rallying and storing feedback with a functional language. Through combination of well-known front-end technologies, a full stack application can be built with ease. Everyone can create a channel on the reference system and use it without limitations. The creator of the channel, also called "channel-administrator" or just administrator will receive a login-key from the system. This key can be easily shared betwixt work groups or the old way, from one person to another. Feedback can be sent anonymously, without attaching personal information. Single feedback can be discussed in a chat-like form and up-voted or down-voted. The simple benchmark has shown the scalability of the application, to a certain extent, by adding more servers or CPU cores to the ErlangVM. Consturcteev can be downloaded from the public Github Repo and installed on nearly any system. It is integrable into the Amazon Ecosystem for dynamical scaling with using the Amazon Redshift database out of the box With the plug system of the Phoenix framework the application and the pipelines can be extended at will. The front-end is in an separately git repository, witch is included as a submodule into the Constructeev repository. This makes changing or customizing the front-end easier and allows the development of the front-end to happen independently from the back-end.

The aim for the future is distribute more and more data-communication over the "Phoenix Channel" system. This would allow us to distribute information, such as new feedback entries or comments, to the clients in real-time. The response time is even faster on a "Phoenix Channel" than on a normal channel. Abstractly speaking it would be possible to send messages from an ErlangVM process to the client and then receive the reply directly back on the channel over a socket, which in turn

reduces network traffic and unneccesary polling from the client side. In future projects the managing and simplifying of diverse channels will be tackled.
Furthermore the development of a Constructeev-Widget will be pushed until it is possible to integrate the widget into other websites as well as export the option-feedback-data. Assigning different colours to different channels will be a new gadget too. Last but not least it will be possible for the user to work on pictures offside Gravatar for a better user experience.

Appendix

A.1 Subsection Appendix

Bibliography

[97810] *Principles of Distributed Systems: 13th International Conference, OPODIS 2009, Nîmes, France, December 15-18, 2009. Proceedings (Lecture Notes in Computer Science)*, Springer, 2010, URL `http://www.amazon.com/Principles-Distributed-Systems-International-Proceedings/dp/3642108768%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D3642108768`.

[Deu] P. Deutsch: *The Eight Fallacies of Distributed Computing*, URL `https://blogs.oracle.com/jag/resource/Fallacies.html`.

[Doe] DoesWhat: *INTERVIEW WITH CHRIS WANSTRATH (GITHUB)*.

[GHVD03] B. N. Grosof, I. Horrocks, R. Volz and S. Decker: *Description logic programs: combining logic programs with description logic*, *WWW '03: Proceedings of the 12th international conference on World Wide Web*, ACM Press, New York, NY, USA, pages 48–57.

[Gui] G. Guimaraes: *Statelessvs Stateful Web Apps*.

[Her07] J. N. Herder: *Failure Resilience for Device Drivers Failure Resilience for Device Drivers*, Technical report, IEEE, 2007.

[JO93] D. R. J. Oikarinen: *Internet Relay Chat Protocol*, Technical report, IETF, https://tools.ietf.org/pdf/rfc1459.pdf, 1993.

[Mar12] Marvell: *Marvell ARM SoC*, Marvell, https://origin-www.marvell.com/embedded-processors/armada-xp/assets/Marvell-ArmadaXP-SoC-product1 edition, 2012.

[Pro] T. L. P. Project: *The Linux PTP Project*, GNU Software Foundation, http://linuxptp.sourceforge.net.

[PS99] L. T. P. Srisuresh, M. Holdrege: *IP Network Address Translator (NAT) Terminology and Considerations*, 1999, URL `https://tools.ietf.org/html/rfc2663`.

[Red16] *Open Source In-Memory Data Structure Store*, 2016, URL `http://redis.io`.

[ROR] *Ruby on Rails Usage Statistics*, URL `http://trends.builtwith.com/framework/Ruby-on-Rails`.

[Tid01] D. Tidwell: *XSLT*, O'Reilly & Associates, Inc., New York, 2001.

[Tür03] C. Türker: *SQL: 1999 & SQL: 2003*, dpunkt.verlag GmbH, Heidelberg, 2003.

[Vol04] R. Volz: *Web Ontology Reasoning with Logic Databases*, Ph.D. thesis, Universität Karlsruhe (TH), Universität Karlsruhe (TH), Institut AIFB, D-76128 Karlsruhe, 2004.