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, April 25, 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 to develop this system, with the ability to communicate or discuss a feedback, will be described in detail. The 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 later on. It tries to show the utilisation of a functional language in everyday life and all advantages and disadvantages which occurred during the implementation of this 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 achieve this, the system has to accept feedback with and without user information. This application collects feedback and makes it possible to discuss the content separately afterwards.
Commonly, an e-mail mailbox or some online discussion board is used to collect feedback. Most of these online services do not scale for the user or the system. For example: We are holding a presentation in front of 400 people and now want to know, how the audience liked it? Sending an e-Mail containing feedback does not really scale, because in a best case scenario with an feedback friendly audience you will probably have over 400 messages in your inbox. Managing feedback in a discussion board can be a huge hassle if you want to find and manage the feedback even without spending more than a thought on the ability to discuss a specific one.
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. Each feedback in those channels can be up- or down-voted and every user can reply to a specific feedback to discuss or clarify misunderstandings. The administrator of the channel is not limited by any rules and can collect general purpose feedback. Constructeev will not provide a predefined questionnaire by the administrator instead the user can write down his/her own view 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. This chapter mainly describes the necessary requirements for the application while Chapter 5 gives a deep insight intothe development process.

2.1 General Requirments

The application needs to be accessible from all kind 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 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 issue.

The application needs to be spitted into two segments, a client-application and a server-application. Both of them should run on as many platforms as possible without any constraints to 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 at 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 an 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 pure application resulting from this project also runs as a public service available to anyone, which makes scalability an important key point. A channel with more than 16.000 feedbacks should be as responsive and usable as one with only 20 feedback. Not only the amount of data has to scale, but the number of requests should also be easy to handle by simply adding more servers.

2.3 Channel Components

To create a easy to use application it is important to be minimalistic. It is a channel task to hold a specific quantity of feedback for a specific topic which is set by the channel admin. There always is one admin per channel. Users which need more than one channel will get seperate login hashes for each channel. This makes sharing a company channel or community channel easier.

2.4 Feedback Components

The feedback component holds all information about a feedback which mainly is important for the user. Administator relevant informations are added to the feedback. Trough this design, the application gets faster in delivering a set of feedback. Feedback always belongs to a channel and is never orphaned. Even if a channel gets deleted, all its children will be deleted too. So a feedback can never exist without its channel.

2.5 Scalability

Constructeev will be available as a reference service and does not limit user by quotas. So there was also the Idea to build the appliaction to be easy to scale. The main requirement was set by handeling 16.000 Feedbacks per channel without any performance loss or noticeable delay. It should be easy to setup a cluster, without changing or recoding the

session management. It is not set and also not limited by the project in which direction to scale. Vertical scaling where the node is upgraded with a faster CPU and more memory or scale horizontal with adding fully fledged servers to the system.

2.6 Administrator View

The administrator view displays the same content as the user view. Aside from the feedback and channels with more information, which will be hidden from the users view, a small management panel will be added to the top. Inside this panel the administrator can change some basic channel settings. Like opening or closing the channel, hiding or viewing all feedback and a button which can change the channels description. Another feature visible only in the administrator view is "mark a feedback read or unread". Thereby you can easily track which ones you have already seen and worked with and which ones you have not yet read. A feedback can also be a favorite to the admin and will be marked with a star and a label. Through flagging a feedback with a star and a label you can earmark it as one of your favourites.

Chapter 3

Related Work

This chapter gives an brief overview of some related work and how this work has been steered and influenced by others. There are plenty feedback collection tools, most of them are for special purpose, but all of them differ materially from these work. In this chapter it is also explained why some features are not implemented or which of them led to imitation.

3.1 Arsnova

Arsnova is an open source audiance-response-service and developed by the Technische Hochschnule Mittelhessen. It aims to be a supportive application for lectures and seminars. Especially for this case it provides a live feedback for the pace of the lecture indicated by four emoticons. The happy emoticon represents "I can follow the lecture" the wink emoticon is "please be quicker", the surprised means the teacher is too fast and the sad smilie tells the speaker the student lost the golden thread. With Arsnova students are also able to als questions for the whole lecture hall. Those can be projected to the black board and discussed with the speaker. Teachers also can create some preparation questions with multiple choice answers, where the system evaluates the student. Statistics are aggregated by the system and accessible by the student for his own learing process and the speaker/teacher can see those statistics and how well students were answering the preperation questions.

3.2 Feedbackbutton

Feedbackbutton is a small widget button which can be added to any website. When this button is clicked, a pop-up window is opened and a form is presented to the user, where you have to fill in your 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 example if you give 50 feedbacks a month you have to pay 6.00 $, for 400 feedback 12.00 $, for 1200 feedback 18 $ and for unlimited feedback 99 $ per month. There is no way to customize the form or receive anonymous feedback.

3.3 Feedbackify

Feedbackify is a commercial company from United Kingdom which offers a website feedback toolkit. It is not possible to create a feedback form without deploing a button on your own website. With this tool you can create and customized tab and also add your company logo to the online form. The feedback will be sent to the Feedbackify servers and is promised to be real-time feedback. The user is foreced expose his name or e-mail. For the administrator it is easy to group feedback into sections or campains. It is also possible to to see where the feedback comes from. Sensitive data like your customer's geographic location, browser, operating system, screen size and IP-Address alre listed.

3.4 GetFeedback.com

Getfeedback is here a representation of the most popular feeback survey tool. Those tools are exellent to get answers to a specific set of question, but those does not cover maladministration outside of the scope of those questions. Sometimes there are not as many selection options available as needed or the survey does take too long. Those statsitic which will be generated by the GetFeedback servers is nice to present and get a visual point, but it has also to be pointed out that this can not cover every This is defintily not the goal of Constructeev to be a survey tool.

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/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.

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 a nice tool for creating and packaging applications.
Another impressive feature offered by Elixir is Metaprogramming, whereby it is easy to extend and define the language itself and one can basically write code which then writes 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 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 through providing Elixir and the ErlangVM. This results in a simple and elegant code as well as a high performance real-time application running over multiple CPU's or even Nodes.
Let us look how the Phoenix pipeline works and which layers of those pipeline a single request will passes 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 url's according to the controller. A pipeline can be specified in the controller, which makes encapsulation of different scopes and API's 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 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 pre-compiled 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 and a low memory usage. It uses Ranch for managing connection. It is pure Erlang and is easy to integrate on every platform without extra dependencies. Cowboy provides a complete HTTP Stack with support for HTTP1.0, HTTP1.1, SPDY and Websockets. The Cowboy project has a small and clean codebase, is well tested and provides a rich documentation.

4.1.3 ETS

ETS is an abbreviation for **E**rlang **T**erm **S**torage. Erlang term storage was built to hold an huge amount of data inside the Erlang runtime but one still has constant access to the data. 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 another. Message passing also works between nodes which concludes in all data being available throughout the whole cluster.

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 too 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 procedures.

4.2.1 Schema

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

4.3 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. So any changes in the view are visible inside the controller and vice-versa.

4.4 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 customary extensions.

4.5 Server-Platform

This sections describes other software and hardware used for developing and testing the 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.

4.5.1 Tested Hardware

In this section some hardware combinations will be shown, in what ways the software has been tested and developed. 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 system can be installed via the included shell script without problems.

Scaleway ARM Server

Scaleway is a Online.Net Company. 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 banwith. 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 CPU's. Those multi processor systems work under one single operating system. Therefore you only have to start a single instance of the BEAM/ErlangVM. They 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 autodiscovery 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, a unixoid operating was installed underneath the ErlangVM.

Chapter 5

Implementation

In this chapter, 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 on how it is possible to scale with over multiple nodes will follow. 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 to evaluate those external dependencies costs time and is important if a distributed scaleable system is built. To confirm that non of them add a bottleneck to your application, is highly important. There will be about this topic 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 be loaded lazy 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 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). 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 those pipelines. All routes with the '/api' prefix are processed by the so called API-Pipeline which is optimized for sending JSON responses. The other line has access to the file system, caches as well as recently used blobs, like images, fonts, HTML templates, precompiled cascading style sheets and JavaScript files. This other line then delivers them to the client.

5.2 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 such 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.
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 handels 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 we

represent sometime a State which is concurrent. In this project case this will be the likes from every channel and the up and downvoting of a feedback. REST APIs solve concurrent problems with the last arrived data is the valid one. This does not work in concurrent problems like counting a click. It is easy to reproduce those error:

```
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 at this moment. Now let call an other client the "like_url" 50 times. The JSON on inside the users browser is the same state as shown above, now meanwhile the client want to like the channel, produces a PATCH request with following data:

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

The client now pushes the value of 199 (198+1) likes to the server, those will accept and write 199 likes in the database, instead of the 249 likes, which would be the actual value. 50 likes are dropped in a so called race conditon. So REST does not fit here, so an RPC (Remote Procedure Call) has been added. Those action url will be delivered with the API, so the RPC-URL can be changed without notifying or changing the client, because those only call the like_url which is provided inside the JSON-Object.

5.3 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 want to. This would result that the feedback-giving-person could boost his feedback or also the admin can like all positive feedback more. For this scenario a simple double-like prevention has been added to the project. For now it is only client-side and could be tricked out with some know how, but the only other (annoying) solution would be to add captchas to every like request. HTML5 compatible browsers do have a client side storage module called "Local Storage". This local storage is a key-value store inside the browser of the accessing device. It was designed to hold a large number of keys without affecting the websites performance. Also the five mega byte limit of an cookie can be passed over with this technology. Another important feature is, that the local storage is not sent to the server with every to the same Domain. The storage offers a new bucket for each origin (domain and protocol), only web-pages with the same origin can work in this unique bucket. SDLP uses this feature and stores on the client which feedback or channel has been liked. The channel is the key and true or false is 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, if the channel has been liked. If the channel has already been liked, to like button is disabled. This technique is also used with feedback up and downvotes. In this case the browser also checks it the channel has been like, but instead of disabling the buttons, it will only offer the reverse action. In one example: The used up-votes a feedback it will only be able undo the action he did before. Another protection for the future would be only to vote one IP-Adress for one Channel/Feedback. But a test inside a larger companie all traffic where mapped to the same IP-Adress because of the IP Network Address Translator (NAT). So only one of 50 was able to like this channel, all others were not. [PS99]

5.4 Distributed Mode

The distributed mode is in use when a single Phoenix application is running on multiple servers. In this case it is not important for the server hardware to be identical, as it has to be in many other 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, so the application can work on more cores parallelly.

In distributed systems there are some fallacies which would allow a fully horizontal scaling.

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
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 cost nowadays almost zero. Only points one to three are now the bottlenecks for horizontal scaling. Network can't be reliable, sometimes network interface cards fail or some more trivial routing of the internet service provider fails, in some cases also the network driver crashes.[Her07]
Sending messages from one node to another takes time, even if they are in the same rack and connected directly to each other. This will slowdown the system, in this case it is important to keep latency as low as possible.
In bench marking a 1Gbit/s connection was used (for bechmarks see 7) so we were strictly limited by those. Message passing, updating code, passing requests to other servers and database requests have to be done through this connection. With adding more server, also more messages from the ErlangVM are sent to each node for synchronization. Also ETS for the session information will be more distributed and more messages are sent across the network to find those information.

— nur wenn zu wenig text — If the network is not reliable, each node should be able to work on the given task and answer if he can reach the client, otherwise it has to wait and sync with the other nodes when the network is available again. With Phoenix only the pipeline is distributed not sending or receiving a request. When a node loses network connection, it will wait for 5000ms to reconnect. If no connection is reestablished the lost node will discard the work and wait for network. The node which handles the incoming and outgoing requests will give

this request to another node to work with. Because the standard timeout of browsers it between 60 and 120 seconds, the application has time to recover. — ende —-

5.5 Sessions

Sessions are a kind of state which should be kept within the period of a session. Cookies are often used to keep session information, because those were sent with every request to the server. The server needs to map those information to the session or state he knows for this 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 low. As described in section 5.4 through adding some servers the application tries to scale as horizontal as possible. When adding a machine, those main memory is not accessible to the other servers. With the standard session management, we lose state and the session, when the clients are redirected to the new server. A common way to solve this is writing session information into Redis, a distributed in-memory data structure store. Constructeev could also use Redis as a distributed in memory store for session data, but why adding another dependency when the ErlangVM offers almost the same feature. For storing session data in this project, ETS (Erlang Term Storage) is used. The only feature which ETS is missing is persistent information store on hard drives. For storing session information we do not need persistence. Just image if the hole system goes offline (eg. power outage) the ErlangVM will be also restarted and a new secret key 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.

5.6 Testing

Chapter 6

Usability and Responsive Design

Chapter 7

Performance and Scalability

In this chapter the benchmark process is described, which also allows to test the scalability of this application to a certain degree. The test system is can not represent real traffic whether conditions which could be in production. Altogehter this benchmarks let us see if the application is even able to handle a large ammount of requests per second and to gain a speedup while adding more nodes to the ErlangVM. All benchmarks in this chapter are done with a open source tool called siege, supports basic authentication, cookies, HTTP, HTTPS and FTP protocols and different than others you can mix GET with PUT/POST and PATCH requests and is also gentle to 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.
Four Benchmarks will be described in the following sections, one of then only tests the read performance, so this test does not add entries to the application only requesting existing ones. Write performance benchmark does only create or edit existing data in the application. The mixed performance does contain PUT and GET Requests with a specified distribution.

7.1 Read Performance

The test system was a Ubuntu 14.04 ARM Server with 4 Core 1.6GHz CPU and 2GB main memory and 1Gbit/s network interface card. A custom Marvell Quad-Core ARMADA XP Series SoC is used in this case [Mar12]. After installing the dependencies database has been filled

with 1000 Channels and in each channel holds about 1600 feedbacks. After the first and recurring benchmarks the server will be restarted so every cache (database, ErlangVM and operating system caches) are cold. The benchmark software was started on another server which is connected with 1Gbit/s connection to the application server.

```
Lifting the server siege...
Transactions:                  32706 hits
Availability:                 100.00 %
Elapsed time:                   6.37 secs
Data transferred:               8.56 MB
Response time:                  0.04 secs
Transaction rate:            4234.19 trans/sec
Throughput:                     2.34 MB/sec
Concurrency:                   91.76
Successful transactions:       32706
Failed transactions:               0
Longest transaction:            0.10
Shortest transaction:           0.00
```

To be sure, the same test has been done with the ApacheBenchmark and resulted in almost the same request per second result with a higher concurrency level. So it can be assumed that both benchmarks a equivalent and as correct as benchmarks can be.

```
Server Software:
Server Hostname:        127.0.0.1
Server Port:            4000

Document Path:          /api/channels/{arg}
Document Length:        292 bytes

Concurrency Level:      100
Time taken for tests:   232.579 seconds
Complete requests:      1000000
Failed requests:        0
Keep-Alive requests:    990049
Total transferred:      555761176 bytes
HTML transferred:       292000000 bytes
Requests per second:    4299.62 [#/sec] (mean)
Time per request:       23.258 [ms] (mean)
Time per request:       0.233 [ms] (mean, across
   all concurrent requests)
Transfer rate:          2333.56 [Kbytes/sec]
   received
```

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       6
Processing:     4   23   3.1     23      86
Waiting:        4   23   3.1     23      86
Total:          4   23   3.1     23      86

Percentage of the requests served within a certain
   time (ms)
  50%     23
  66%     24
  75%     25
  80%     25
  90%     27
  95%     29
  98%     31
  99%     34
 100%     86 (longest request)
```

7.2 Write Performance

```
Lifting the server siege...
Transactions:                  23.297 hits
Availability:                 100.00 %
Elapsed time:                  10.37 secs
Data transferred:               8.56 MB
Response time:                  0.04 secs
Transaction rate:            2236.29 trans/sec
Throughput:                     4.08 MB/sec
Concurrency:                   71.42
Successful transactions:       23.297
Failed transactions:               0
Longest transaction:            0.10
Shortest transaction:           0.00
```

7.3 Mixed Performance

65% only read - 35% some post action - from those some post actions are 25% Answers or Comments

```
Lifting the server siege...
```

```
Transactions:                  23.297 hits
Availability:                 100.00 %
Elapsed time:                  10.37 secs
Data transferred:               8.56 MB
Response time:                  0.04 secs
Transaction rate:            2236.29 trans/sec
Throughput:                     4.08 MB/sec
Concurrency:                   71.42
Successful transactions:       23.297
Failed transactions:               0
Longest transaction:            0.10
Shortest transaction:           0.00
```

7.4 Distributed Performance

```
Lifting the server siege...
Transactions:                  23.297 hits
Availability:                 100.00 %
Elapsed time:                  10.37 secs
Data transferred:               8.56 MB
Response time:                  0.04 secs
Transaction rate:            2236.29 trans/sec
Throughput:                     4.08 MB/sec
Concurrency:                   71.42
Successful transactions:       23.297
Failed transactions:               0
Longest transaction:            0.10
Shortest transaction:           0.00
```

Chapter 8

Conclusion

The developed application as part of this bachelor thesis shows that it is possible to build with a functional language a scalebale web service for requiring and storing feedback. With the combination of well-known front-end technologies a full stack application can be built with ease. Everyone can create on the reference system a channel and use it without limitation. The creator of the channel, also called channel-administrator or adminstrator will recieve from the system a login-key. This key can be easily shared through work groupes or from one person to another. The simple benchmarks has shown that the application is scalable in a certain extent by adding more servers or CPU cores to the ErlangVM.

Appendix

A.1 Subsection Appendix

Bibliography

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

[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.

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

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

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

[Red16] 2016, URL `http://redis.io`.

[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.