

Bau einer performanten und wartbaren Web Applikation

am Beispiel von wahlkabine.at

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Andreas Taranetz, BSc

Matrikelnummer 1126689

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Purgathofer

Wien, 11. Oktober 2017

Andreas Taranetz

Peter Purgathofer

Building a high-performance and long-term maintainable web application

on the example of wahlkabine.at

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Andreas Taranetz, BSc

Registration Number 1126689

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Purgathofer

Vienna, 11th October, 2017

Andreas Taranetz

Peter Purgathofer

Erklärung zur Verfassung der Arbeit

Andreas Taranetz, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11. Oktober 2017

Andreas Taranetz

Danksagung

Ein besonderer Dank gilt dem Institut für Neue Kulturtechnologien/t0, die es mir mit dieser Arbeit ermöglicht haben, einen Beitrag zu einem hoffentlich noch lange laufenden Projekt für politische Bildung zu leisten.

Ein weiterer Dank gilt Alois Taranetz, der diese Arbeit durch das Bereitstellen von Serverressourcen für eine Staging Umgebung von Anfang an unterstützt hat.

Acknowledgements

I would like to offer my special thanks to the Institute for New Culture Technologies/t0, which enabled me to contribute to a hopefully long-lasting project for political education.

I would also like to thank Alois Taranetz, who has supported this work by providing server resources for a staging environment right from the start.

Kurzfassung

Wird man in sozialen oder öffentlichen Medien zum richtigen Zeitpunkt erwähnt, kann das zu Lastspitzen führen, die Web Applikationen an ihre Grenzen treiben. Diese Diplomarbeit zeigt wie man eine hoch performante Webseite entwickelt, die sich skalieren und dadurch rasch an wechselnde Lastverhältnisse anpassen kann. Die Neuentwicklung von wahlkabine.at wird dabei als Beispiel dienen. Diese Seite hat die zusätzliche Anforderung für lange Zeit wartbar zu bleiben.

Für ein performantes, skalierbares und wartbares Ergebnis müssen diese Eigenschaften im gesamten Entwicklungsprozess berücksichtigt werden. Diese Arbeit hebt vor allem die architektonischen Entscheidungen hervor und wie sie von spezifischen Anforderungen beeinflusst werden. Die Implementierung als Single-Page-Anwendung wird im Detail erläutert. Dabei wird ein Schwerpunkt auf wahrgenommene und tatsächliche Ladezeiten gelegt. Weiters wird das Hosting der resultierenden Anwendung betrachtet und welche Möglichkeiten existieren die zu übertragende Datenmenge vom Server zum Client reduziert werden kann.

Diese Arbeit richtet sich an Entwickler die Wege suchen, um die Performanz ihre Webseiten zu steigern, sowie an jene, die Softwareprojekte mit den gleichen Eigenschaften planen. Für wahlkabine.at konnte eine Leistungssteigerung erzielt werden, die jegliches Auslagern auf externe Ressourcen überflüssig macht, selbst wenn sie im Hauptabendprogramm des ORF erwähnt wird.

Abstract

Being mentioned on social or public media at the right time can lead to short load peaks that could bring web applications to their limits. This master thesis shows the design and implementation of a high performance system that is able to quickly scale to adapt to changing loads. As a running example, the rebuilding of wahlkabine.at is explained in detail, which has the additional requirement of remaining long-term maintainable.

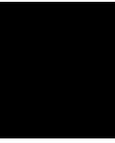
To achieve performance, scalability and maintainability, these qualities have to be considered in every step of development. This work elaborates on the architectonic choices and the influences of specific requirements to them. The implementation as a Single Page Application is explained with emphasis on actual and perceived loading times. Another focus is placed on hosting the application and corresponding ways of reducing the amount of transmitted data a client needs to load from the server.

This work addresses developers seeking for ways to increase their websites performance or to those in the midst of planning a new project with the same qualities. In the case of wahlkabine.at a performance improvement can be achieved which eliminates the need of external servers altogether, even when being mentioned at primetime on national television.

Contents

Kurzfassung	xi
Abstract	xiii
Contents	xv
1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement	2
1.3 Aim of the Work	3
1.4 Approach	4
1.5 Structure of the Work	5
2 Feature Analysis	7
2.1 Existing Features	7
2.2 New Features	10
2.3 Data Model	10
3 Choosing the Technology Stack	13
3.1 Existing Solutions for Similar Problems	13
3.2 Architectural Questions	15
3.3 Architecture of wahlkabine.at	17
4 Implementation	23
4.1 Angular and Tooling	23
4.2 End-user Interface	29
4.3 Administrator Interface	32
5 Hosting	39
5.1 Node Applications	39
5.2 Reverse Proxy	40
5.3 Performance Improvements	42
5.4 Scaling Out	45
5.5 Migration	47
	xv

6 Evaluation	49
6.1 Performance	49
6.2 Scalability	50
6.3 Maintainability	52
7 Conclusion	53
7.1 Achievements	53
7.2 Problems	54
7.3 Outlook	54
List of Figures	57
Acronyms	59
Bibliography	61



Introduction

1.1 Motivation

In the advent of major elections, political parties push hard to establish a positive image of their leading candidates through all kinds of media. What is sometimes eclipsed by this are their respective positions on topics, which matter to their target audience. This is where Voting Advice Applications (VAAs) come into play. Their intention is to help voters with their decision by providing a purely objective perspective on political parties.

To achieve this goal, VAAs collect a number of topics from the manifestos of all competing parties and phrase them in form of concise yes-no questions. Prior to an election these questions are answered by the parties or journalists on their behalf. The users of the application can then answer the same questions and compare their opinions and beliefs with the ones of the running parties.

The tendency of election campaigns becoming more and more concerned with leading candidates instead of topics is, among others, observed in Garzia's work in [1]. He further shows that VAAs help their users to get an objective result, given that the parties opinions are not shown until the user has entered his own. This prevents the user from answering the questions in favour of his preferred party. It further supports a general goal of VAAs, which is to show their users alternatives to their accustomed voting behaviour.

The website wahlkabine.at [2] is the leading Voting Advice Application in Austria. It is operated by the Institute for New Culture Technologies/t0. The institute is maintaining the site and has covered every general election as well as most of the regional elections since 2002. A major concern for the providers always was and still is to keep the project as independent and trustworthy as possible. Advertisements or selling of user data are avoided by any means. Instead some money comes from the ministry of education for diverse projects done at schools. As the journalists workforce to source the questions and

check the parties' replies are the biggest cost factor little resources are left for hosting and reworking the site's outdated technology stack.

1.2 Problem Statement

Due to the popularity of wahlkabine.at there are up to 800,000 unique visitors in the last four to six weeks prior to any major election. The daily data traffic slowly ramps up until the election day with small increases whenever the page is mentioned in newspapers or television. Today's effects of social media can lead to massive load peaks in the matter of minutes, e.g., when influencers such news anchors share website's link. The new version of the website should therefore be able to cope with fast changing traffic rates.

Prior to this work, the server had to be shifted to external hosting providers in order to handle the data traffic in advance to general elections, which are usually held every five years in Austria. Aside from these times there is only very little load to be handled such that external hosting would be too expensive. The outdated technology stack, which has never been intended to be shared on multiple servers, requires the site to be moved in its entirety, which is a very cumbersome and error prone process. A technology stack has to be found, which enables the site to be scalable across multiple nodes and which is as platform independent as possible for easy porting.

For each election, wahlkabine.at is creating a questionnaire containing around 25 questions. The answers and comments from the parties are then collected to a spreadsheet and later imported into the database through scripts. This procedure makes it difficult to change even simple things such as typing errors in the imported data.

Many political questions are difficult to boil down to simple phrases. For this purpose, wahlkabine.at uses a glossary of political terms to further explain certain topics. The glossary entries are directly embedded into the respective questions. This is another type of content that should be easier editable by the editors themselves without the need of a web developer. Aside from the content specific to elections, news about the site itself or other related events are published every few months.

Therefore a way of management has to be implemented that does not require the editor to have a deeper understanding of Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) or other web technologies.

A means of counting the number of started and successfully finished questionnaires must be implemented, to give the providers a sense of the impact their project has in the general public. This, however, must be done without relying on external services. Furthermore, to ensure the users' privacy it is preferred to abstain from using any external services at all, be it content delivery networks (CDNs) or externally hosted caches, so no tracking of users by third parties is possible.

In order to validate the journalistic process and detect biases that may exist in the selected questions, not only the number of finished questionnaires, but also the given

answers should be logged. Again the privacy of the user must be protected, thus it must not be possible to link a stored answer back to the user who entered it.

The previous version of `wahlkabine.at` was lacking a certificate to enable a secure connection. Although no sensitive data was transmitted, this must be established in order for the website to become future-proof.

1.3 Aim of the Work

For the general election on the 15th of October 2017 `wahlkabine.at` should be rebuilt to cope with the increasing traffic and the now existing variety of devices from which the users access the site. In collaboration with Alexandra Geier [3], who designed a future-friendly user interface, this work covers the development and relaunch of `wahlkabine.at`.

The aim of this work is to show how the right choice of modern technologies can lead to a web application with eligible properties. The priority for `wahlkabine.at` was on the following qualities: performance, scalability, long-term maintainability and user experience. As these are very comprehensive concepts we will narrow them down for the purpose of this work.

The term performance is used in the sense of delivering the website to as many users with as little server resources as possible. Scalability in our context means horizontal scalability, or being able to share the application on multiple computational nodes. As indicated earlier, the Institute for New Culture Technologies/t0 does not have the financial resources to employ a web developer specifically for `wahlkabine.at`. Hence, after this work, the project will stay in roughly the same shape over the next one or more decades. The choice of technologies should therefore also be influenced by the requirement of being able to find developers that understand them in the next decade. Although the term user experience is usually associated and indeed largely affected by the optical design of a website, there are limitations and effects to it that stem from the choice of technologies. This work will focus only on those technological influences on user experience.

The selected tools, frameworks and choices for other software in this work can not be the perfect solution for every kind of project, but the details of the considerations behind each layer of the resulting stack can be transferred to a wide variety of projects.

Besides the comparison of different technological choices, this work will cover patterns or approaches that are necessary for building high-performance web applications. All those insights will be applied in the running example of rebuilding the website `wahlkabine.at`.

Additionally to the above mentioned goals, the hosting is an important pillar of performance. This work aims at optimizing hosting, specific to the example of `wahlkabine.at`, but most of the shown improvements can be conveyed to other projects as well.

Humans are not the only users of a web application. Therefore, when talking about user experience, we will also consider search engines and robots with other purposes. Before this work, `wahlkabine.at` was ranked on the first place of major search engines. After

switching to the new version, this rank should be maintained. It will be shown which actions can be done to retain the so-called pagerank when rebuilding a website from scratch.

A final goal of this work is the data migration of the old system into the new data structures. In the previous version of wahlkabine.at all the questions ever asked are still stored, but not all of their respective questionnaires can be accessed. Getting the legacy data into a new union format could enable the users to run through all questionnaires since the establishment of the website in 2002.

1.4 Approach

As Bennett and Rajlich [4] described, every software project undergoes cycles of evolution until the knowledge about its architecture is lost. At this point the project enters the servicing phase, in which only minor changes can be done. The previous version of wahlkabine.at has reached this servicing phase. This is why it will be rebuilt from the ground up in the course of this work. In addition, the old website will be phased-out and closed down.

The decision to reconstruct the website enables us to use a totally different technology stack, which is also necessary to achieve the aforementioned qualities. Building a whole web application raises a variety of questions to be solved and decisions to be made. This work will approach them in a top down manner.

First of all it will be analysed what type of application wahlkabine.at is, depending on its features and content. The specifics of required features will be examined in detail to discover technological barriers, which need to be overcome. A data model is created from the results of the feature analysis to persist the dynamic contents in a structured way.

It will be considered if it is possible to build a complete solution with a single tool as the requirements seem rather simple. Possibilities for the client and server side stacks and their respective layers will be presented. Each of these stacks layers will be evaluated separately according to parameters suitable to determine which option meets our qualities of performance, scalability, long-term maintainability and user experience best.

Furthermore, the specifics of the chosen tools and frameworks will be examined and how they can contribute to achieving our desired qualities. Also the problems involved with the chosen technologies as well as possible ways to mitigate them will be investigated.

The first weeks of productive usage will be monitored to discern if the developed solution is sound and fit for possible future demands. It is of especial interest if the resources of the site's providers suffice or if scaling out by using external servers is necessary.

1.5 Structure of the Work

The work is structured based the software development cycle. The next chapter will cover a detailed feature analysis of wahlkabine.at. The result of this will be used to classify the type of application wahlkabine.at falls into, as well as to help with all latter technology choices. Another outcome is the data structure used to represent every piece of dynamic content.

Chapter 3 builds up on these results and presents various technology stack options and arguments why to use or not to use them in certain scenarios. For each layer of the stack alternatives will be given and compared to each other. The selected choices for wahlkabine.at will be explained in more detail and it is reflected why they meet the given requirements.

Chapter 4 will present in depth the abilities and problems of Single Page Applications (SPAs) and how they can be conquered with the chosen front-end framework Angular. Aside from those, solutions to other well known problems such as internationalization and state management are offered. The tools for developing and building the productive site will be mentioned as well. Both end-user and administrator applications will be illustrated and explained.

In Chapter 5 details about hosting the site will be shown. Many small performance improvements presented in this section, mostly implemented with the reverse proxy Nginx, can be applied to other web applications with different tooling as well. Furthermore, it demonstrates how the layers of the application could be scaled out across multiple machines independently from each other. Last but not least the migration of the old website's content and the preservation of the pagerank is explained.

An evaluation of the effectiveness of all efforts undertaken to achieve the desired qualities is done in Chapter 6. External testing tools are used to compare the performance of the old version of wahlkabine.at to the new application. The usage diagram of the first weeks of production shows what loads the application is required to withstand.

Chapter 7 concludes by pointing out what was achieved in the course of this work. It presents some of the problems that occurred during the project life cycle. Finally, possible improvements for a future version of wahlkabine.at will be shown.

Feature Analysis

Under the premise of staying long-term maintainable it is of uttermost importance to do a thorough feature analysis upfront to the implementation or even the technology selection. This can prevent the need for larger changes in the near future, thus saving costs for developers or in the best case getting by without having to change the application at all.

In the case of wahlkabine.at the task of covering all possible features is facilitated by the fact that the website has already been running for 15 years. Thus many possible use cases for the application have occurred in its lifetime. As a result many features could be extracted by searching through the existing version of wahlkabine.at. To stay sure that nothing was omitted and the details were done right, regular meetings with the Institute for New Culture Technologies were held every second week on average. As soon as the first prototypes were finished and especially towards the release date those meetings were held more frequently.

This chapter will now cover all previously existing features in detail as well as all new features that were proposed to facilitate the workflow of the editorial team. To conclude this chapter a data structure will be presented that involves all dynamic content of the website.

2.1 Existing Features

2.1.1 Start Page

The start page of wahlkabine.at should show the current internal news articles and latest elections. If there is an impending election the user should have the possibility to start its questionnaire from the start page. Much of the start page's other content is static. There is general information about the aim of wahlkabine.at and a disclaimer about the interpretation of the questionnaire results. Furthermore, links to most of the static

subpages are embedded. With the exception of the articles, the only other dynamic content is a list of sponsors and media partners that are embedded with their logo and a link to their respective sites.

2.1.2 Questionnaire

The questionnaire component is of course the most prominent and important feature of wahlkabine.at. For each election a list of around 25 yes-no questions is created by journalists covering important political topics. The question text is a short sentence, but it can contain terms that are further explained in a glossary. If such an explanation exists, the respective term in the text should be highlighted and an explanation should be shown if the users interacts with the highlighted section. For every question the user can answer with "yes", "no" or "no answer" as well as rank the question from unimportant to important on a 9 piece scale. When using the "no answer" option too often, the user should not be allowed to answer any further questions with "no answer". To prevent any unwanted bias due to the order of the questions, they are randomized for every user.

In the previous version of the website this questionnaire was built as a separate application starting in a new browser window. One of the reasons for this is the requirement that during the questionnaire the users' attention should not be distracted by other components of the website. Another reason would be to prevent the user from accidentally exiting the questionnaire by mistakenly pressing on a button. For a better user experience it should now be integrated into the rest of the website while incorporating the same benefits. Also the loading times between each question should be as short as possible. A progress bar shows the users how many questions they have already answered and should motivate them to finish the questionnaire.

2.1.3 Result Page

After finishing all questions the result page names the party with the highest score and shows a sorted bar graph with all participating parties and their score. This result page allows the user to share his result on major social media networks. It also prompts the user to restart the questionnaire, compare his answers with the ones given by the parties or go back to the overview page of an election. Although the number of questions and participating parties were roughly the same over all past elections covered by wahlkabine.at, the questionnaire and especially the result page should function for an arbitrary number of parties or questions to be future-proof.

2.1.4 Election Page

Each election has its custom subpage containing all relevant information, such as the involved parties and the editors responsible for the questions. A text block with custom format can contain information and external links about the election that do not fit into any other content block. The list of sponsors and partners is included in the same manner as on the start page. In fact, the list on the start page should be the same as those on

the latest election page. A further subpage should list all questions and the respective answers of the parties as well as comments given by the parties to explain their answer. In the same way all rejected questions should be included in a separate page. Those are questions that were answered by the parties, but did not provide enough distinction between them to be useful for the questionnaire. Another dynamic content block are the downloads on the elections' subpages. For most of the elections custom documents are created explaining the used methodology – although it stays the same for all elections, – a compact overview of the parties' statements and information for the media. As almost all of those contents are optional, the site must be able to handle missing information.

2.1.5 Social Media Sharing

The sharing feature of the result page is an important tool to extend the publicity of the website. When users share their results a page has to be created that contains the results in the form of a description text inside the meta tags. Instead of the real result page this custom page is contained inside the sharing uniform resource locator (URL) leading social networks to showing the result in an orderly fashion. Other benefits of this method are that users cannot manipulate their results easily and possible character limits cannot be exceeded by the description text.

2.1.6 Further Subpages

There are several other subpages containing the information of the election pages in other forms. The glossary page contains all explanations used inside the question texts in alphabetical order. The editor page catalogues all editors of each election sorted by the election date. An archive page indexes all elections ever covered by wahlkabine.at as well as links to the election results. Last but not least the news page lists all articles and elections sorted by publishing date. In the previous version of wahlkabine.at all those sites were maintained with a simple content management system (CMS) and had to be changed manually for each new election.

2.1.7 Search

To compare the statements and answers of parties across different elections a search page lets the user find all questions ever asked in any questionnaire. For this purpose every question is associated with a list of topics which can be used as a filter criterion. Other criteria are the type of election, a date range or a specific string that must be contained inside the question text.

All other pages contain only static content that is not about to change in the near future, like the idea and goals of wahlkabine.at, the history of the project, related literature or information about the general methodology used to calculate the matching on the result pages.

2.2 New Features

The most important requirement for the new version of wahlkabine.at, aside from the qualities mentioned in the previous chapter, is to be usable on a wide variety of devices. This applies to the server side application as well, as it is unknown on which system it will be deployed on in years to come.

2.2.1 Internationalization

Although the website was and will be primarily used for Austria the new version should be prepared to get translated. An important detail concerning internationalization is that it would be sufficient for the providers to have different websites accessible under language specific domains instead of one single website with an integrated language switch option.

2.2.2 Logging

The previous version of the website already recorded how many people finished the questionnaire and what answers they have given to each question, but the extraction of this data from the object database was a tedious task and therefore seldom performed. The logging itself was done after every question generating a lot of requests to be handled by the server. The new version should provide an easier way to generate the same statistics.

2.2.3 Administrator Interface

An interface must be installed to enable the editing of articles, election pages and their associated data. The information about parties as well as the glossary entries should be reusable because they are only rarely changed. If they do, however, the changes should only be visible in future elections and must not be propagated backwards. This requirement is necessary to address the parties participated in a questionnaire in the same manner as they were on the ballots of the respective election. For glossary terms the problem is similar. If existing occurrences would be changed, their explanations could be deceptive for the context they were used in.

The interface to perform these changes must be adequately secured to prevent an unauthorized person from defacing the application. All the above mentioned content should be editable without requiring special knowledge about HTML, CSS or similar technologies. Nevertheless, it should be possible to place custom HTML into articles to be future-proof and give the editors maximum freedom in creating content.

2.3 Data Model

As a result of the feature analysis, all discovered dynamic contents were structured in a way, that allows for easier querying of the data.

The following class diagram 2.1 shows the final data structure used in the new version of wahlkabine.at. Internal news are stored as *articles* with an HTML content. By setting the publication date to a day in the future, an editor can create content and review it before it is visible to the users. A custom URL segment allows for better search engine optimization (SEO) of new articles and is necessary to take over the URLs of the previous version of wahlkabine.at which did not follow any specific pattern. If the content is very long, the option exists to cut it off on the start page.

As the basic information for *elections*, is very similar to *articles* the latter one just gets extended with all necessary fields. The downloadable contents themselves are not stored in the database but only their file names. Every *election* has at least two parties involved. Every *question* must be answered by each *party* although there is the option to set the consent to the equivalent of "no answer" for a party as well. In the final solution *questions* do not store a relation to *glossary entries* but embed them inside the question text. To make them reusable, they are stored as their own entities as well, just like parties.

This data model has to be kept in mind when selecting an appropriate database for the application. In Section 3.2.3 different types of databases are compared, which are reasonable to store this structure and facilitate the required queries.

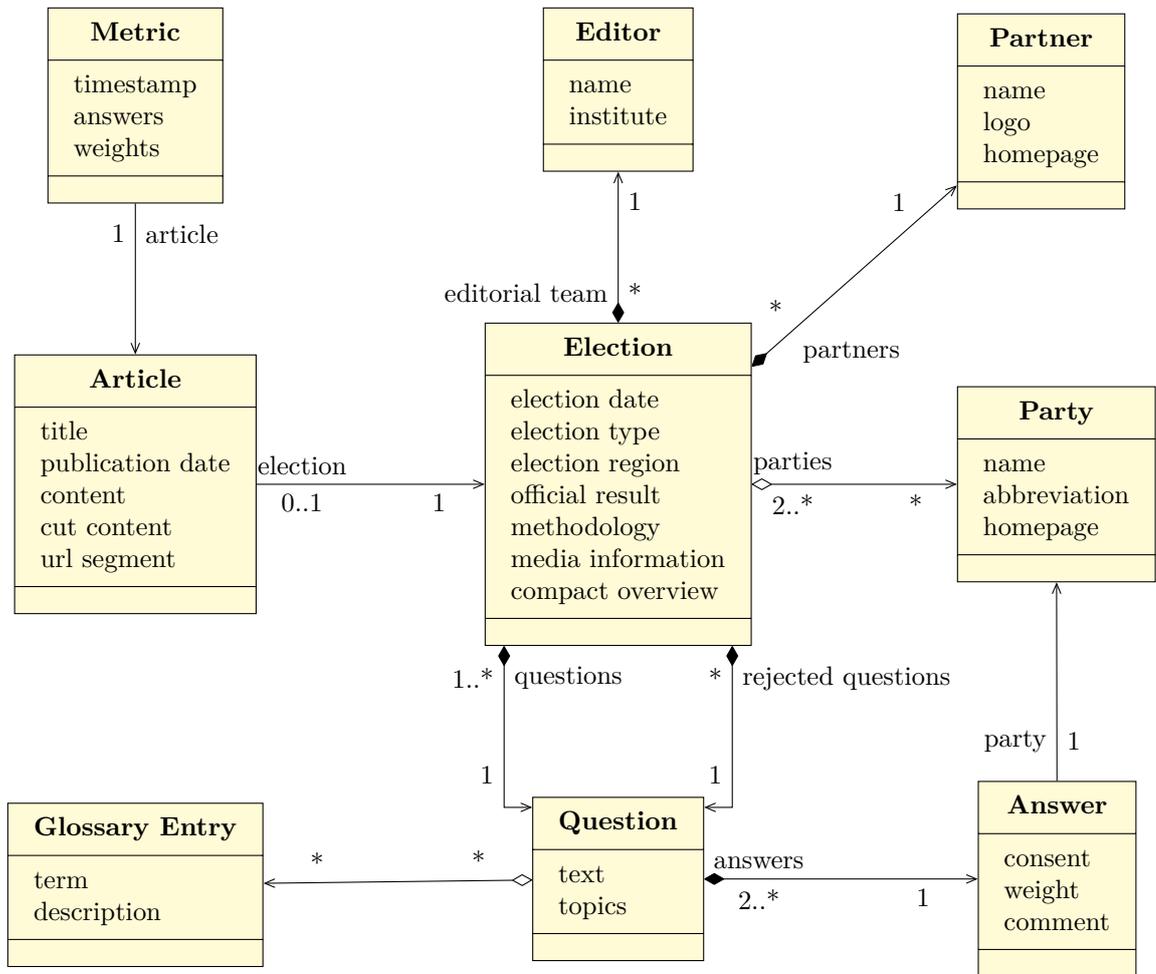


Figure 2.1: Class diagram

Choosing the Technology Stack

The previous chapter provided a thorough feature analysis on top of which this chapter will build upon. After elaborating which kind of application wahlkabine.at is, this chapter explains the technological choices that were made. Furthermore, it is shown why the resulting stack is more than suitable to achieve the desired qualities proposed in Chapter 1.

3.1 Existing Solutions for Similar Problems

A good starting point for finding the right technologies to build a certain product, or in our case host a web application, is comparing it to existing products with similar characteristics and take survey their inner workings. Although the Internet brought forth many diverse websites and web applications, there are some common systems that appear frequently throughout the web, such as online shops, forums, social networks or wikis.

wahlkabine.at is most similar to a journal or blog system. It has articles that are published and sorted in reverse chronological order. The questionnaires can be seen, and are indeed modelled, as extension of normal articles. Blogs usually give the users some kind of possibility to comment on the published content. The lack of this feature is compensated by the possibility to easily share the site's content or questionnaire results on social networks and lead the discussion there. The intention hereby is also to further enlarge the media reach of wahlkabine.at.

As blog systems are fairly common, there already exist a variety of tools to build them. A few of them will be listed and reviewed if they are suitable for the problem at hand.

3.1.1 Blog Publishing Service

There are services that offer editors in a what-you-see-is-what-you-get (WYSIWYG) style for building customized blogs like [5]. Such sites are perfect if the need for customization

is little and the creation and editing of content should be done by non-developers. As not only the editor but also the final blog is hosted on servers of the respective service provider, the blog owner has a permanent dependency of the provider which could easily cease to exist in the next decade. For performance this implies that the blog owner has no means of influencing how the site is delivered to his users. Also the user experience is heavily restricted by the possibilities of the particular service. On the other hand, coping with the traffic and providing the needed resources to cover the demand is a problem left to the service provider. Keeping the website up-to-date and secure is another problem that is shifted towards the service provider. For `wahlkabine.at` no such site could provide the necessary level of customization needed to cover the required features.

3.1.2 Static Site Generator

Another option are so called static site generators. These are tools that generate a structured website out of input files that are easier to create than HTML directly. Markdown is used as a common input language, often with some proprietary extensions. As the name suggests, the output of those generators are static HTML, CSS, JavaScript (JS) and media files that only need a simple web server to be delivered to the client. Whenever an editor changes some contents of the site, parts or sometimes the whole website is regenerated and replaces the old version. More about their inner workings and some notable examples can be found in the work of Dhillon [6].

This modern approach goes against a general trend seen in the Internet a decade ago, namely the switch from static to dynamic web applications as described by Jazayeri [7]. As no rendering or other processing happens on the server side, the only limitation for the performance is the upload speed of the server. Fewer moving parts also mean fewer attack vectors and thereby better security. Scaling out is also fairly easy, as the content can be hosted on arbitrary many nodes. As long as the generator itself is maintained the maintainability of the site is given. As there are site generators that accept very similar input files, even the migration from one to the another is possible.

For `wahlkabine.at`, however, this solution is not applicable as it is mainly intended for rarely changing, completely static sites. A dynamic feature like the questionnaire would require some kind of client side rendering framework to be used embedded in the static pages. Also the search feature relies on a database where the content of the otherwise static pages would be partially duplicated.

3.1.3 Content Management System

A third option would be to use a classic CMS. These systems are often used for corporate websites or to present projects. They can be heavily customized and extended with plug-ins ranging from simple photo galleries to whole web-shops. This flexibility has made them very popular but their reputation also made them a wanted target for hackers. Patel et. al. compared the biggest open source CMSs in terms of their strength against common

attacks in [8]. As Patel found, the security issues often stem from misconfiguration or wrongly used third party plug-ins.

A general purpose system, like those compared by Patel et. al. [8], is harder to be trimmed for performance than a specific solution. Contents, styles and other metadata are stored inside a database, so aside from the rendering engine the database could become another performance bottleneck. Server side rendering and the resulting necessity of sessions makes horizontal scaling another problematic aspect for CMSs. There are services that are specialized on hosting wordpress or other common CMSs, but they mostly prohibit the usage of custom plug-ins which would be necessary to implement the questionnaire or other required features.

3.2 Architectural Questions

Having seen these tools from the previous section and why they do not work for wahlkabin.e.at we will now take a step back from the problem and analyse qualities of applications that have a direct impact on the architecture. The feature analysis will enable us to answer the questions and the answers will guide us towards an architecture, fitting for our specific problem.

To be able to talk about architecture in an abstract way we will only focus on three logical layers existing in almost every application in one way or another. The presentation layer, responsible for displaying the content and handling the user's interaction with it. An application layer managing the internal business logic and a data layer handling the various data sources. As we are talking about web applications there is always at least one server and arbitrary many clients involved.

3.2.1 Rate of Change in Content

Under the aspects of high-performance and scalability it is of utter most importance to know how fast an application's content is changing and how fast this changes must be displayed to users. This influences the way how the content needs to be delivered to the user and also where and how the rendering of the content into views, or in our case web pages, should happen. Together with the question of how much content is personalized for specific users and which content is shown to all users in the same manner, this shows how much impact caching could have on the overall performance of the application.

Depending on how complex it is to render the content and what resources are available on the server or client, the presentation layer described above should be moved to the server or client respectively. The bounds of the logical layers do not have to coincide with the bounds of the computational nodes, meaning that it could be beneficial to move parts or sometimes the whole application layer to the client as well. This decision, however, is affected by the following questions as well.

3.2.2 Different Types of Users

What types of users are interacting with the application and how are they interacting with the application? Also, non-human users such as external services or bots and their interactions with the site have to be considered. The answers to these questions should determine what interface should be used between the application and the presentation layer. Furthermore these considerations lead to meaningful vertical cuts through layers. For example in classic CMSs there are content authors and end users. Both have different application requirements and therefore have separate presentation layers providing different means of interacting with the content.

With long-term maintainability in mind, the design of the interface between application and presentation layer should receive special attention. Technologies used on the client side are changing rather rapidly and the layer getting outdated first will be the presentation layer. Having a robust interface the switch to a different presentation layer can be done more easily.

Last but not least different types of users mostly have different permissions and will require some kind of authentication mechanisms. For security to be effective it must be considered during all stages of development, therefore of course influencing the architecture as well.

3.2.3 Data Structure and Usage

The choice of the database system should heavily depend on the structure of the stored data and the types of operations performed on the data. Relational databases are very popular and widespread because of their versatility and the existence of well established and high performance products. Also the modelling of entities as tables fits well with the object oriented paradigm, which is the basis for most programming languages used in enterprise applications.

In terms of performance and scalability Not only SQL (NoSQL) database systems can outperform relational databases in some specific use cases which is why they should be considered as an alternative if these qualities are to be achieved. Han et. al. [9] identified major types of NoSQL databases such as key-value, column-oriented and document databases and their respective downsides.

Especially when it is hard to transform the data model into a null value free normal form, a good alternative could be document databases as they can store objects with varying schemas in the same collection. Single documents can be written and read very fast even if the database grows to enormous sizes. A drawback, however, is that these databases are not built to efficiently join different entities in queries. The next section will mention how this downside can be circumvented.

Some NoSQL databases achieve their performance and scalability by neglecting consistency. Although this may seem undesirable, there are many applications that do not require transactions and their consistency and should not strive for it in the prospect of other more important qualities.

3.3 Architecture of wahlkabine.at

We will now try to answer the question from the previous section to derive the right architectonic choices for wahlkabine.at. In the next sections different possible choices for each layer will be compared according to our qualities.

The rate of change in content is exceptionally slow on wahlkabine.at as elections only happen every other year and not that many internal articles are published. There is no private section on the site and therefore no personalized content. Because of these facts and the limited computational power of the server, owned by providers of wahlkabine.at, it would be enough to render the pages e.g. every hour and cache the rendering results.

Unfortunately, there is the questionnaire feature where the view of each question can not be rendered as a static site, as they must be shuffled for each user. For every election there are several thousand possible result pages, depending on the number of questions and the number of participating parties. To be able to cope with these dynamic features and still stay able to render them with the limited server resources we will use a presentation layer on the server and on the client. Details about this will be explained in latter sections.

There are only two types of users of wahlkabine.at. On the one hand the administrators creating new articles and inserting data of elections, parties, questions and glossary items. On the other hand the end-users conducting the questionnaires. The only instance where an end-user manipulates data happens indirectly by generating log entries. Except from that they only access content in various forms of read operations. The administrators only require a simple and functional interface to insert data in a structured way. This interface must prevent the insertion of invalid data and should check for missing information when adding new entries.

The modelled data structure depicted in figure 2.1 is already partitioned in entities that could be stored in separate tables if a relational database would be used. Consistency is not an issue for wahlkabine.at as new articles or content changes must not be visible immediately and it does not matter if some log entries are left out for the statistics calculation. From the feature analysis we can already deduct the operations that will be executed on this data. The start page, the news page and the articles page all require a list of articles joined with the election data, sorted and projected one way or another. For the article pages or the questionnaire a single article with all its associated data is needed. These observations lead to the conclusion that a document database could be an option in our use case. To facilitate the above mentioned queries, all entities related to an article could be persisted as one joined document. Thus, only projection and no joins operations must be done when querying the data.

3.3.1 Database Layer

The previous section already showed how a document database could be used as an alternative to classic relational database systems in this special case. Those are often

3. CHOOSING THE TECHNOLOGY STACK

specialized for specific operations, for wahlkabine.at we will select the one with the best performance for features presented in Chapter 2.

Li and Manoharan compared several different NoSQL databases to a Microsoft SQL Express database [10]. Most interestingly for our use case are the times for reading and writing single entries as well as fetching all keys of a collection or table. Of the compared database systems only RavenDB, CouchDB and MongoDB [11] are document databases so except the Microsoft SQL Express the other result do not matter for our use case. The following results are all taken from [10] and the given values are milliseconds used to perform the operations.

Database	Number of operations					
	10	50	100	1000	10000	100000
MongoDB	8	14	23	138	1085	10201
RavenDB	140	351	539	4730	47459	426505
CouchDB	23	101	196	1819	19508	176098
Cassandra	115	230	354	2385	19758	228096
Hypertable	60	83	103	420	3427	63036
Couchbase	15	22	23	86	811	7244
MS SQL Express	13	23	46	277	1968	17214

Figure 3.1: Performance of read operations

Database	Number of operations					
	10	50	100	1000	10000	100000
MongoDB	61	75	84	387	2693	23354
RavenDB	570	898	1213	6939	71343	740450
CouchDB	90	374	616	6211	67216	932038
Cassandra	117	160	212	1200	9801	88197
Hypertable	55	90	184	1035	10938	114872
Couchbase	60	76	63	142	936	8492
MS SQL Express	30	94	129	1790	15588	216479

Figure 3.2: Performance of write operations

Database	Number of keys to fetch					
	10	50	100	1000	10000	100000
MongoDB	4	4	5	19	98	702
RavenDB	101	113	115	116	136	591
CouchDB	67	196	19	173	1063	9512
Cassandra	47	50	55	76	237	709
Hypertable	3	3	3	5	25	159
MS SQL Express	4	4	4	4	11	76

Figure 3.3: Performance of fetching all keys

When performing simple read operations, MongoDB and CouchDB outperform the SQL database. For writing, this only holds for MongoDB and here only for higher quantities of operations. Write operations to our database will happen mostly because of the logging of metrics data so the write performance is a very relevant factor as well at peak loads. The third important factor when considering the way our application uses the database is the performance of fetching all keys of a collection. Here, MongoDB is beaten by the

performance of the relational database, but as the collection of e.g. articles and glossary items will not grow in the thousands this will not make any difference.

Parker et. al. compared MongoDB to the Microsoft SQL Server more thoroughly in [12]. They arrived at the conclusion that the relational database outperformed the document oriented MongoDB only when more complex queries should be executed. Fortunately, such operations only occur in wahlkabine.at when the administrator queries for statistics about metrics data.

With those statistics in mind the decision fell on MongoDB as the single underlying database system for wahlkabine.at. Other factors that were considered are the possibility to scale the database across multiple nodes and the quality of documentation. Scaling is a feature inherently built into MongoDB, because it is meant to handle quantities of data that single database systems often cannot handle. According to Stackshare [13], a site that lists the usage of software development tools of all kinds, MongoDB is one of the most popular NoSQL database systems, which hopefully means that it will continue to be supported for years to come. It is open source and aside from being ported for various operating systems also provides drivers for almost all major programming languages and frameworks.

It could be beneficial for the performance of the application to use a separate database for the logging function as there are more specialized databases for this task. The additional maintenance work required and the already good performance of MongoDB lead to the decision to use only a single database.

3.3.2 Application Layer

In Chapter 2 it has been shown that the interaction with the end-users is rather simple and the dynamic aspects of the application are very limited. This reasons, together with the rather limited server resources, lead to the resolution of putting most of the application logic on the client side. This leaves the server with the task of simple content delivery and a secure database access thereby effectively reducing server load as the whole rendering process happens on the client. Another desirable side effect of this is that, after an initial load time, the navigation to other pages is faster to almost instant. Instead of requesting the rendering of a new page only the raw data required to construct the page is requested from the server or if no new data is required the new page can be shown almost instantly.

This kind of cut between server and client may not be the optimal solution for the administration interface where e.g. the validation of inputs should happen on the server side. This solution will be used for the administrator interface nevertheless. Firstly, it is more important to optimize for the several hundred thousand users instead of the few administrators. Secondly, regarding maintenance it is easier to have two almost identical applications instead of a totally different stack for the administrators.

The resulting requirements for the application's layer are therefore as following: It must handle as much concurrent access as possible and provide a simple way to access the

3. CHOOSING THE TECHNOLOGY STACK

database. Furthermore, to communicate with the client a representational state transfer (REST) interface must be provided by the server.

The choice of MongoDB as database also poses some restriction to the choice of the application layer, as it would only be sensible to choose a language and framework with a supported driver to connect to the MongoDB. This, however, does not narrow the search all that much. With long-term maintainability in mind it should again be a language that is popular now, to hopefully be supported in decades to come as well. According to [13] the top three tools used for web applications are JavaScript running on a Node interpreter [14], PHP [15] and Django [16] which is written in the language Python.

Lei et. al. [17] compared these three with more or less computational intensive tasks all running on the same hardware, behind the same web server and on top of the same database. It has to be mentioned that instead of Django, WebPy is used as representative Python web framework. The following figure taken from [17] shows the results of comparing the throughput (requests per second) of each tool when calculating the tenth Fibonacci number.

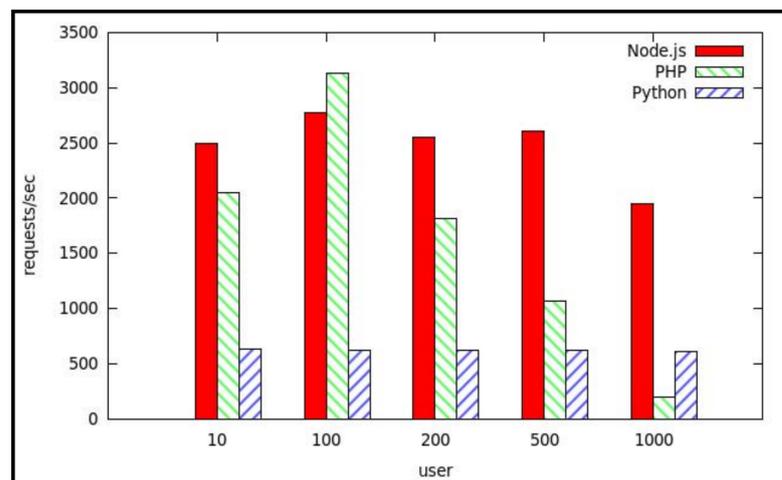


Figure 3.4: Throughput of different frameworks

Another simpler application was tested as well by Lei et. al. [17], with Node having even more advantage over the other two. Aside from the performance aspect, other reasons speak for Node as well. There are tools available to manage clusters of node applications making scalability easier. The data format for communicating with MongoDB is JavaScript object notation (JSON). As Node already uses JavaScript there is no need for additional object-relational mapping (ORM) in the chosen configuration. This reduces moving parts and effectively removes another possible performance bottleneck. Another synergistic effect with the presentation layer will be explained in the next section. Similar to MongoDB, Node is available for a wide range of operating systems and has a large and active community, due to its popularity.

3.3.3 Presentation Layer

Web development is driven by existing standards, protocols and browser technologies as Jazayeri pointed out [7]. Especially these advanced browsers' capabilities allow us to fully use the resources of the client's machine through the browser. Instead of simply displaying pre-rendered HTML files, so called front-end application frameworks emerged that are capable of manipulating or building the document object model (DOM) themselves. Furthermore they are able to react to user inputs and communicate to the servers application programming interface (API) without having the browser to load a new page after every interaction.

This allows us to build what is called a Single Page Application (SPA). These types of websites only need to load a single page from the server that contains the whole application. The DOM changes depending on user inputs and data loaded through the API from the server. To come back to the use case of wahlkabine.at, a user completing a questionnaire only calls the server once per questionnaire instead of once per question which should further reduce the load and thus increase the overall performance of the website. This, and further specifics of SPAs, will be explained in detail in Chapter 4.

There is a variety of front-end frameworks to choose from, e.g. VUE, React, AngularJS, EmberJS, BackboneJS, KnockoutJS. Again we will only look at those with the biggest community, with the prospect of them being still supported in years to come.

Molin did an in-depth comparison of the popular AngularJS, its successor Angular2 and React [18]. For a practical comparison he built a sample application. Figure 3.5 and 3.6, taken from his work, show the time the respective framework needs to initially load and to update the items of this example application.

In both cases Angular2 outperformed the other two, but there are two other important metrics especially for the user experience. Figure 3.7, also measured in [18], shows the time the framework takes to initialize on the client. The other factor is the framework size, because this could result in a longer initial page load if the client's connection is slow. Here, Angular2 is far bigger than its competitor React, but it will be shown how to cope with this in Chapter 4 and 5. The smaller size of the React framework has to be taken with a grain of salt, as it relies heavily on third party plug-ins which would increase the bundle size again and make it harder to maintain. For wahlkabine.at the choice was therefore made to use Angular2.

In general, all technological choices were made with given resources in mind and the need for other developers to take over the product resulting from this work.

3. CHOOSING THE TECHNOLOGY STACK

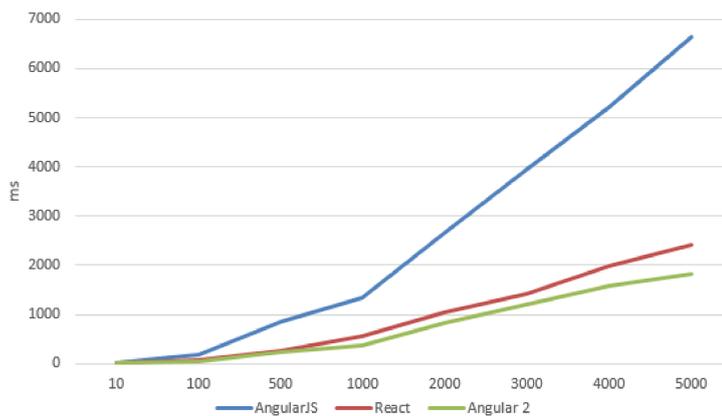


Figure 3.5: Front-end frameworks item load time

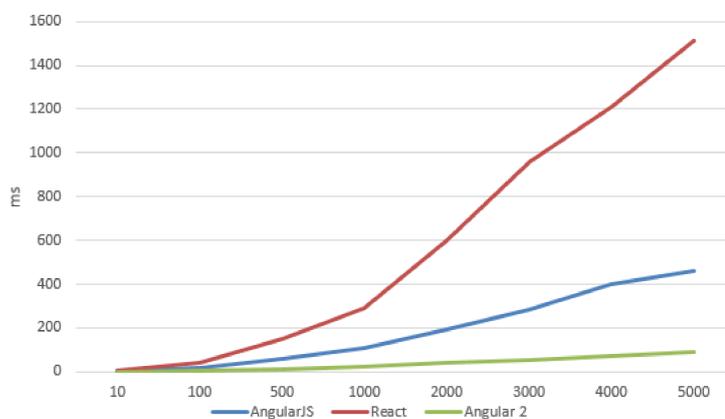


Figure 3.6: Front-end frameworks item update time

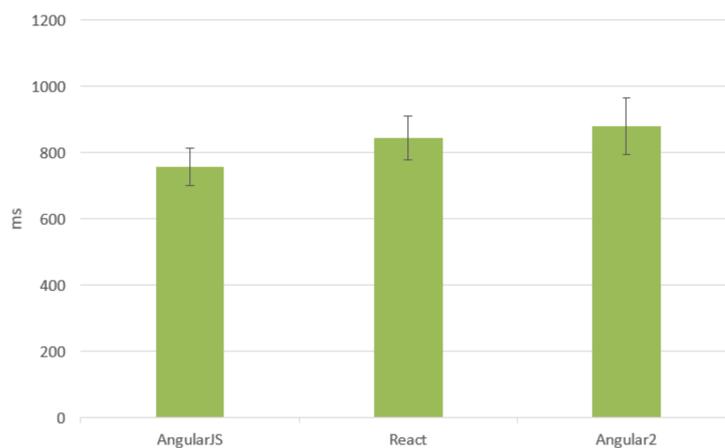


Figure 3.7: Front-end frameworks initial load time

Implementation

In this chapter the details of the implementation of `wahlkabine.at` will be explained in depth. After explaining SPAs and Angular 2 in particular, some problems of SPAs are presented and how they can be solved with Angular 2. Furthermore the development tools used to build the productive site are presented. In the following we will reference Angular 2 simply as Angular, which should not be confused with its predecessor AngularJS.

4.1 Angular and Tooling

Single Page Applications (SPAs) allow us to execute the task of rendering the HTML and process a lot of the application logic on the client's machine. For the user experience this means a responsiveness close to that of a native application. Figure 4.1, taken from [18], shows the communication between server and client when using a SPA.

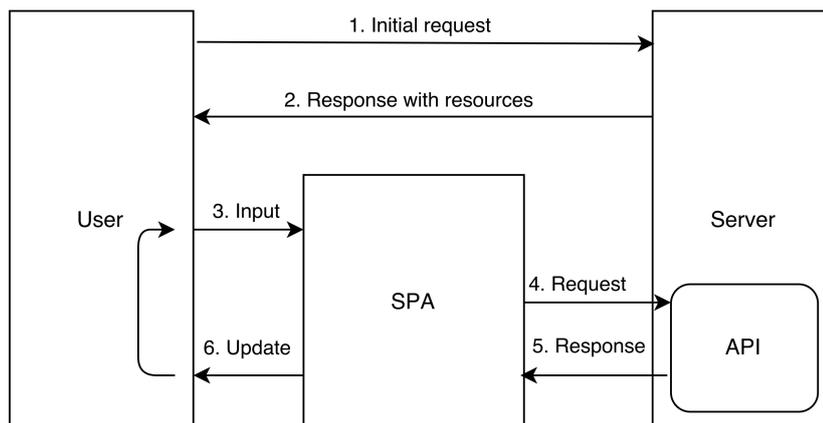


Figure 4.1: SPA server communication

The first request (1) from the client to the server is executed exactly the same as for every other website. The server responds (2) with the a file, e.g. the `index.html`, which contains only references to JavaScript and CSS files. The front-end application framework is contained in its entirety in these script files and initializes itself on the client after the first load has finished. Now the DOM is rendered by the front-end framework and the browser can draw the page according to the CSS. This is where the initial load time from 3.7 and the framework size come into play, as they significantly affect the time required until a first meaningful page is rendered.

It is at this point in the life cycle, that the application becomes interactive and responds to user input (3). Most SPAs will show a user some kind of loading or splash screen to cover the time it takes for the application to become reactive. We will see a better solution in Section 4.2. Depending on the application logic, some user inputs will trigger requests to the server (4). Until a response from the server (5) arrives, the site stays fully responsive and continues to react to user inputs. From these inputs and the new data coming from the server a new view can be rendered by the SPA (6).

Like most front-end frameworks, Angular aims for a clear separation of concerns by using the model view controller (MVC) pattern. Angular applications are built from multiple so-called components that are encapsulated into each other. These contain code that acts as the controller and store the model in form of the respective component's variables. They also contain an HTML template that is extended with custom syntax, which defines the connection with the model and controller methods.

In his book [19] Takada pointed out how important modularity is for maintainability. Angular components are grouped together into modules which make it fairly easy to separate concerns. Component's responsibilities can range from whole sub pages to simple banners or pop-overs. A component can have a selector which acts as a custom HTML-tag name. Together with custom parameters that act as input and events that can return data from a component, custom HTML elements can be built and reused.

Building the application from small components makes it easier to test and refactor and therefore maintain the application easier. A requirement for this statement to be true is to have as little dependencies as possible between components and modules. This can be especially tricky when considering states and state changes. There are programming patterns which allow for a comprehensible state management even in complex applications. One of the most widely used such pattern, namely Redux, is shown in Section 4.2.

Although there are more projects using AngularJS than Angular according to [13] at the time of writing this work, one should consider to use the newer version of Angular as it is finally production ready and made some important progress over its predecessor. The performance improvements shown in Figure 3.5 and 3.6 stem from better change detection which is the algorithm that checks if and what parts of the model changed and what parts of the DOM have to be changed to represent these changes.

Another big change is the programming language used. Although there are other possibilities too, Angular is primarily meant to be written with TypeScript. Most

browsers currently only fully support the ECMAScript-5 standard. TypeScript is an extension of the newer ECMAScript-6 developed by Microsoft that introduces typing into JavaScript. As it is simply a superset of normal JavaScript every code written in JavaScript is also valid TypeScript, which makes the transition to the new language very easy for JavaScript developers. To be executable on current browsers the TypeScript code has to be transcompiled into ECMAScript-5 code. Although this requires another step in the build process and an additional tool, the advantages outweigh the disadvantages.

With its type inference mechanism the TypeScript Transpiler makes assumptions about the types of variables, even if they are not declared by the programmer. This allows for static code analysis and better tooling, which both can prevent errors early on. Using explicit types is also a form of documentation that leads to better maintainability. To enforce explicit types we used a linter for `wahlkabine.at` with a rule set specific for Angular applications. Again this is an additional tool that has to be maintained, but the consistent code style and better code quality justify this effort.

Another major advantage of Angular over AngularJS is the use of Observables. Programming in an asynchronous manner was previously done with use of Promise Objects. The Observable and its related classes work similar to the design pattern of the same name. Where a promise was used to handle a single event, an observable is used to handle a stream of events. This allows for easier asynchronous handling of API calls with the ability to e.g. retry failed calls, cancel running calls or handle errors in an orderly fashion.

As mentioned in Section 3.3.3 the size of the JavaScript bundle and initialization effort affect the time until the user can see a first meaningful page and start interacting with the site. When initializing, Angular needs to parse the metadata included in annotations on component classes. The component parser then creates factories to instantiate the respective classes. At the application entry point a bootstrap function is called on the root module. This module specifies a component that gets bootstrapped. In `wahlkabine.at` this component has the selector `body` resulting in a replacement of the body element in the `index.html` file and thereby granting full control over the sites displayed DOM. The component parser is included in the bundle and does the above described *compiling* of Angular components on the client.

As all possibly used components are known at build time and defined in their respective modules, the parsing could happen at build time as well. Angular supports this approach with its so-called ahead of time (AOT) compilation. For `wahlkabine.at` different build processes are defined with Webpack [20]. Webpack is a tool to create bundles of resources. With the use of plug-ins and loaders for different kinds of file types it controls how bundles are created. To use AOT compilation a different loader for TypeScript files is used that bundles the compiled factories of each component. In contrast to the normally used just-in-time (JIT) compilation, bootstrapping the application is done by calling the factory method of the root module. While this work arose, Angular introduced a command-line interface (CLI) for creating and building Angular applications. When using this CLI to build an application for production, AOT compilation is used by default.

Reducing the initialization phase to just calling already compiled factories reduces the startup time at the client drastically. Figure 4.2 demonstrates the time used at the client for the JIT compiled version of wahlkabine.at. As the graphic shows, the majority of the time is used for executing JavaScript. Although Figure 4.3 reveals a significant improvement of AOT over JIT compilation, these graphics disguise the second important factor for the start-up time, namely the bundle size as they are measured in a locally hosted environment.

Range: 0 – 3.66 s

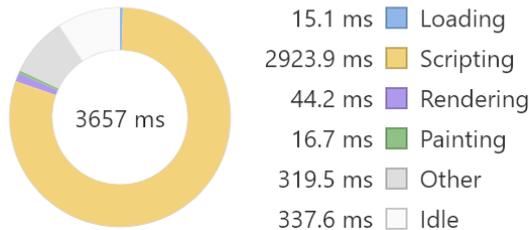


Figure 4.2: JIT performance

Range: 0 – 1.66 s

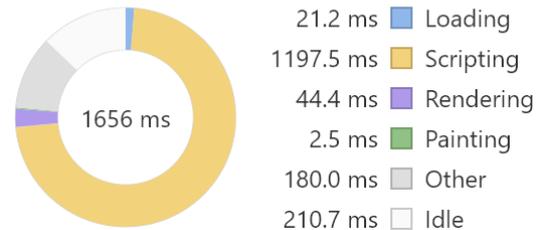


Figure 4.3: AOT performance

AOT compilation already helps reducing the bundle size for small applications like wahlkabine.at. Though the size of the compiled factories is bigger than that of the raw components, the code of the Angular compiler is a chunk bigger than the entirety of the application code.

Using Webpack wahlkabine.at is split up into three files. The app.js file contains all code specific to the application. A vendor.js file combines the framework code and other third party libraries. To stay compatible to as many browsers as possible a polyfill.js file includes scripts that compensate missing features of some browsers that are used by Angular. Each of those files name is extended by a hash dependant of its content. The reason for this approach lies in the behaviour of caches. Their usage will be explained in detail in Chapter 5. The clients are requested to cache the JavaScript files for a long time to prevent them from loading those big resources on every visit. Whenever a new version of the application is deployed, not necessary all of the JavaScript must be loaded again from the server, as the vendor bundle and the polyfills will most likely stay the same.

Figures 4.4 and 4.5 show these three files and every contained chunk depicted in relation to their file size. As mentioned earlier the Angular compiler poses the biggest chunk and although the app.js file is bigger when using AOT compilation, the combined size of all three bundles decreases from 2.72 Megabyte to 2.24 Megabyte because of the omission of the compiler.

For production builds this JavaScript bundle and the CSS is minimized with special Webpack plug-ins. Otherwise they would still be far too big for users with slower connection speeds. Further techniques to reduce the amount of data necessary to be transmitted from the server to client will be shown in Chapter 5.

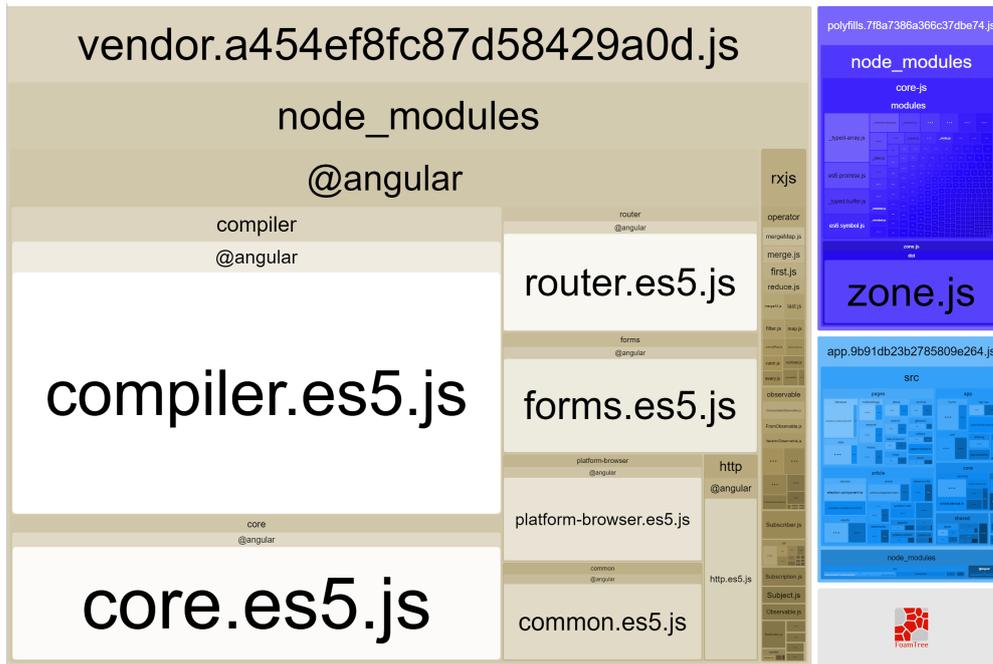


Figure 4.4: Bundle composition with JIT - 2.72 MB



Figure 4.5: Bundle composition with AOT - 2.24 MB

For development purposes the JIT build is used nevertheless, as it allows for a significantly faster turnaround time between code changes. A third build option used for the end-user version of wahlkabine.at will be shown in the next section. The administration interface is rolled out with AOT compilation and a separate build process is used for the server side application logic.

A problem that every application faces and which should be considered early in the architecture is internationalization. Although wahlkabine.at is branded for and primarily used in Austria there have already been occasions, like the EU-parliament election in 2009, where the whole site needed to be translated to be part of the international network Vote Match Europe [21].

Being a mature framework, Angular has a way of supporting internationalization that can even be applied to legacy systems. All contents inside component templates get extended with an *i18n* attribute. For attributes that require translation other special attributes exist. There is even support for gender alternatives and pluralization in a way that conforms with the International Components for Unicode (ICU) Message Format. The content of these attributes can include a meaning and a description. Before building the application a CLI tool extracts all appearances of the *i18n* tags into a XLIFF file, which is the standard used for translations and has therefore wide tool support. For every unique meaning this file contains an entry annotated by the optional description to help the translator find the right translation for the given context.

For each language the application should be used in, a copy of this file must be made and enhanced with the respective translations. In our case the plug-in handling the AOT compilation is responsible for weaving this translation in and replacing all occurrences of *i18n* attributed content with the one from the XLIFF file. For multiple languages the different versions of the website must be delivered via different domains.

Another common problem is the handling of different environments such as the production server, a separate staging server and the local environment. The implemented Webpack build process therefore utilizes different environment files that define constants like the location of the front-end, the API or the connection string to the database. These contents are injected into the JavaScript bundles and can be used like global constants.

One last thing Webpack handles, besides the JavaScript, are style sheets. Like templates styles can be defined scoped to a specific component by declaring them in the annotations of a component. Webpack can extract these bits of style information into one single file. Another tool we used to make the style sheets more maintainable is Syntactically Awesome Style Sheets (SASS). This extension of normal CSS enables the usage for variables to enable a single definition of repeatedly used values, like colors or certain breakpoint definitions. Another helpful feature of SASS are mixins that allow the grouping of CSS declarations into a reusable expression e.g. for multiple browser specific definitions of a border radius.

4.2 End-user Interface

This section will point out some details about the implementation of the end-user interface. Some of the problems encountered here and their solutions are most likely applicable to other projects as well. For details and further insights about the conceptualization of the optical design see the work of Geier[3].

The usual `index.html` of an Angular application is rather empty and `wahlkabine.at` is no exception. All the content is hidden inside the JavaScript bundles and is waiting to get rendered. Considering the style of the start page depicted in Figure 4.6 we see that there are only two interactive elements visible. The menu button on the top right and the invitation to start the current questionnaire in the top center. The majority of the page is textual content, such as the newest articles on the bottom left and general information about `wahlkabine.at` on the bottom right.



Figure 4.6: End-user interface

4.2.1 Angular Universal

One of the reasons why we have chosen Angular for `wahlkabine.at` is a feature named Angular Universal. A goal of the Angular framework is to be platform independent, meaning it can be executed in a browser, a server or other environments. As our server is built with Node – a JavaScript runtime – we are able to execute almost the exact same code the client does to render the page on the server as well. This allows us to change the usual workflow of SPAs shown in Figure 4.1 to enhance the *perceived performance*.

4. IMPLEMENTATION

When the client requests a page from the server the response is the `index.html` including links to all resources. Additionally, the server renders the requested page itself by executing the Angular application and includes the resulting DOM into the body. As mentioned earlier, our application takes full control of the content by replacing the body tag, but this happens only after the whole framework code has been loaded and executed. In the meantime the browser is able to display the pre-rendered contents included in the first response.

Figure 4.7 illustrates what is happening at the initial load of `wahlkabine.at`. After around 300 milliseconds the loading of the `html` response (first bar) from the server is finished and the website is rendered without the need of executing any JavaScript. Although not even the banner graphic has finished loading the users already see the final site's structure. While they need to comprehend the content and the location of possible interactive elements the bundles finish loading from the server (next three bars). When the bundles have finished loading the page is rendered again on the client and replaces the pre-rendered version. As they should be exactly the same no jerking or flashing of the page is visible. This technique greatly improves the user experience especially for users with slower connections.

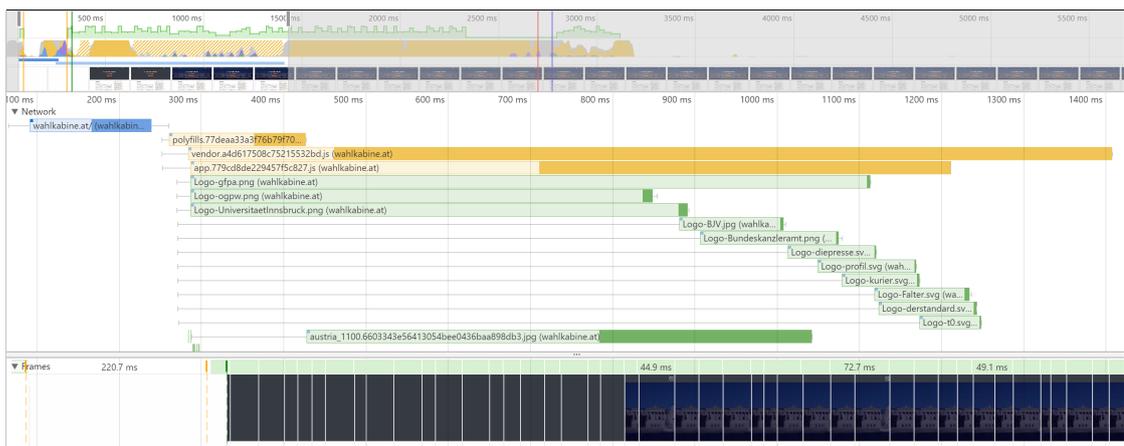


Figure 4.7: Universal application loading

A separate build process combines the universal application with the API code to a single JavaScript file that gets executed on the server. To prevent the server from rendering pages for every single request, the results are cached in a reverse proxy, as explained in Section 5.2.

Aside from increasing the perceived performance, this approach also solves another major problem of SPAs. Bots accessing the website such as crawlers for search engines do not execute JavaScript for various reasons. Social networks scrape the site's contents to be able to embed them and as mentioned in Chapter 2, sharing on social media is an important feature for `wahlkabine.at`. Whenever a bot accesses `wahlkabine.at` it receives

the pre-rendered result which is valid HTML also containing metadata tags for social networks and navigable links important for search engines.

To render most pages of `wahlkabine.at` various requests to the database are necessary. These requests can be executed on the server without network latency because the database is installed on the same physical machine. When rendering again on the client, the same database requests would be repeated as the same code is executed. To prevent these redundant requests the state of application, executed on the server, needs to be transferred to the client. For `wahlkabine.at` this is achieved by adding the results of the database requests as a script tag into the head section of the HTML response. All the data found in this script tag does not have to be requested again from the client application, effectively reducing the amount of server request and thereby the server load.

4.2.2 State Handling

Aside from the database responses another major part of the application state is the questionnaire. A common problem we came upon is the management of comprehensible state transitions. As different components have access to and manipulate the questionnaire's state, it is easy to lose track of all possible different state transitions. As this is a common problem, there exists a tool to mitigate this problem. Redux [22] is a library with a set of features to manage the application state much in the same way as a state machine. A so-called *store* is the single instance that keeps the state. With defined *actions* the state can be changed. How these actions influence the state is described in *reducers*.

For performance reasons the bundle size needs to be as small as possible. This is why `wahlkabine.at` handles the questionnaire state in a very Redux-like fashion without including the library itself. Angular together with ReactiveX already provides everything needed to build the core components of Redux. An injectable service that is treated as singleton by Angular contains a behaviour subject which is the construct that stores our state. All components that depend on the questionnaire's state inject this service and subscribe to the subject. Whenever a component emits an action that leads to a state change all other components are informed without duplicating any information or creating tight coupling between components.

SPAs could be realized without the need of changing the route the browser is pointing to. In fact, AngularJS handled routing to different components by just adding an anchor to the location of the page it was called on. Angular manipulates the browser history directly to make changes in components look like route changes on regular websites. This is important because especially on mobile devices the browser back navigation is used to navigate between different pages of an application. To enable the user to switch between questions with the browser back action, after every answer, a fake navigation to another page happens to change the browser history.

4.2.3 Logging

Logging all required information is a difficult feature with the chosen architecture. As the server holds no state, only the clients know all information that should be logged. Requesting a specific election does not necessarily mean that the user started the included questionnaire. Reaching the result page triggers no event on the server that could be logged. The solution was to let the client perform an API call whenever the questionnaire was finished including the given answers in the original order. This leaves the problem of users starting the questionnaire and quitting the application before finishing it. Fortunately, browsers offer a *beforeunload* hook to execute code before the window is closed. However, the code executed for this event has limitations. Thus, a synchronous version of the API call had to be implemented in contrast to the otherwise asynchronously executed calls.

For security purposes the server side application, providing the API to the Angular application, has a custom database user with restricted access. Only reads from the collections are allowed for this user, with the only exception being the metric data collection. To prevent an eavesdropper from sniffing the arguably sensitive logging data, all communications between server and client are protected by Hypertext Transfer Protocol Secure (HTTPS).

4.2.4 Glossary

Another feature that was quite difficult to implement is the glossary popup as shown in Figure 4.8. The content displayed inside the popup is encoded in the attributes of a custom HTML element surrounding the text that should be highlighted. Using JIT compilation the popup could be constructed as dynamic component on the client. For the above shown improvements wahlkabine.at uses AOT compilation, so a different solution was needed. Before binding the custom HTML content coming from the API into the template, it is modified by a regex replacement rule. This replacement turns the attributes into a function call that later opens the popup. This way the style of the glossary feature can be changed without having to edit the content in the database.

4.3 Administrator Interface

This section demonstrates the administrator side of wahlkabine.at with its features and implementation details. Here, the performance is secondary, but the need for security is a major concern. Again Webpack is used for building the Angular application in a manner, similar to the end-user interface, but techniques like Angular Universal were omitted.

The administrator interface is built with Bootstrap [23], a library that offers many CSS rules and some JavaScript snippets to build responsive interfaces. This allows the administrator interface to be decently usable on a wide range of devices.



Figure 4.8: Glossary entry popup

4.3.1 Security

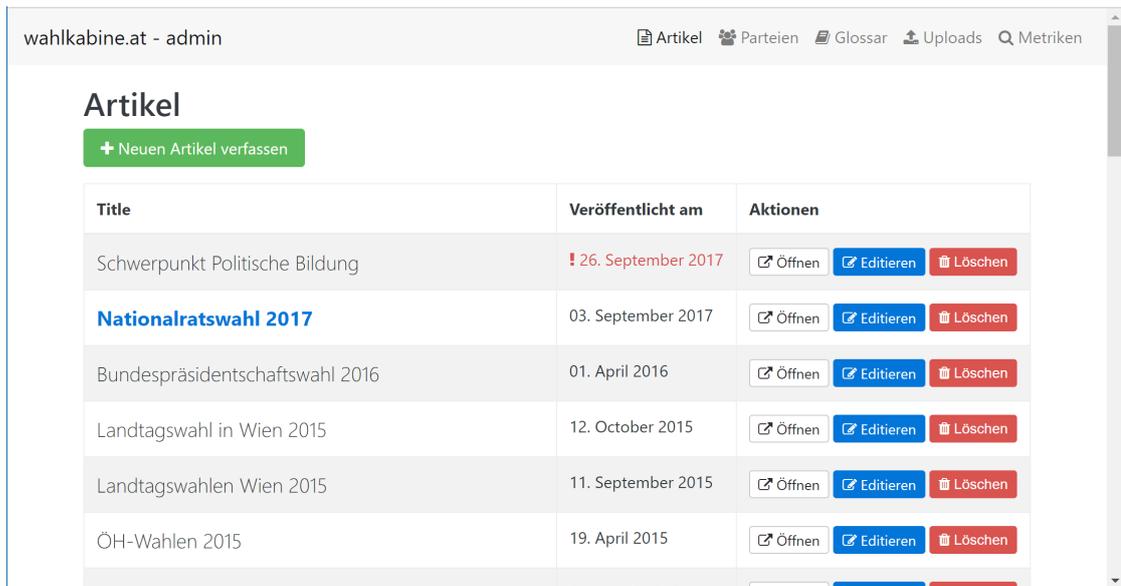
To ensure security, a separate database user is configured for the administrator application with only the minimal set of permissions. All REST APIs are secured with basic HTTP authentication. As a weakness of this mechanism is the little protection of the transmitted password, strict transport layer security or rather HTTPS is enforced to secure the password from eavesdroppers. An additional protection against brute force attacks is explained in Section 5.2.

Angular provides so-called Route Guards that control if a specific route can be accessed. The administrator interface uses such a guard to check if the admin user has already provided his credentials when navigating to any page that requires elevated rights. If the credentials are missing, the Angular routing component redirects the user to a login, and upon entering a username and password redirects back to the page that was originally requested.

4.3.2 Collection Overviews

The administrator application is structured in the style of the data model. In the top right of Figure 4.9 the tabs are named according to the database collections they edit. Every tab is structured similarly. An overview shows an ordered list of documents contained in the respective collection. In this view the administrator has the option to add, edit or delete items where in the latter case a modal dialogue would check if this action was performed by mistake or intentionally.

The article overview has additional features: It helps the administrator understand the state of the start page by marking articles that will be published in the future with a red publishing date as these articles are not visible to users. The title that appears in the banner of the start page is highlighted blue. The other feature is a link to open article pages which circumvents the reverse proxy's caching mechanism to see the newest state after editing an article. With this direct link articles can be viewed in the end-user



Title	Veröffentlicht am	Aktionen
Schwerpunkt Politische Bildung	! 26. September 2017	Öffnen Editieren Löschen
Nationalratswahl 2017	03. September 2017	Öffnen Editieren Löschen
Bundespräsidentchaftswahl 2016	01. April 2016	Öffnen Editieren Löschen
Landtagswahl in Wien 2015	12. October 2015	Öffnen Editieren Löschen
Landtagswahlen Wien 2015	11. September 2015	Öffnen Editieren Löschen
ÖH-Wahlen 2015	19. April 2015	Öffnen Editieren Löschen

Figure 4.9: Administration interface

interface that are not discoverable by end-users. This eliminates the need for a special preview feature in the administrator interface.

4.3.3 Editing View

The editing views make use of another type of route guard to prevent the user from leaving the page when unsaved changes are present. Figure 4.10 shows an overview of the article editing view. The top section contains all necessary fields of an article such as the title, publication date and content. By default, a URL segment is generated from the title that replaces white spaces and special characters. If necessary, this can also be overridden. The second section can be added if the article contains an election. Each section and subsection is an own component that is called for validation of its inputs depending on whether it has been added. This allows for better maintainability, as the interface can easily be extended or modified to adapt to model changes.

The content is entered into a HTML WYSIWYG editor. A simple rich text editor would have sufficed for most use cases. Such an editor would have required us to save the editor input in addition to its HTML equivalent to enable the administrator to edit previously created content again. Another benefit of the more complicated HTML editor is its capability to embed all types of content. In consultation with the editors of wahlkabine.at, a set of features was defined that allow maximum flexibility while at the same time minimizing the risk of breaking the site's design principles.

Figure 4.11 shows the section for editing the questions of an election. The same editor as for the article content is used with one extension. A button to insert glossary entries

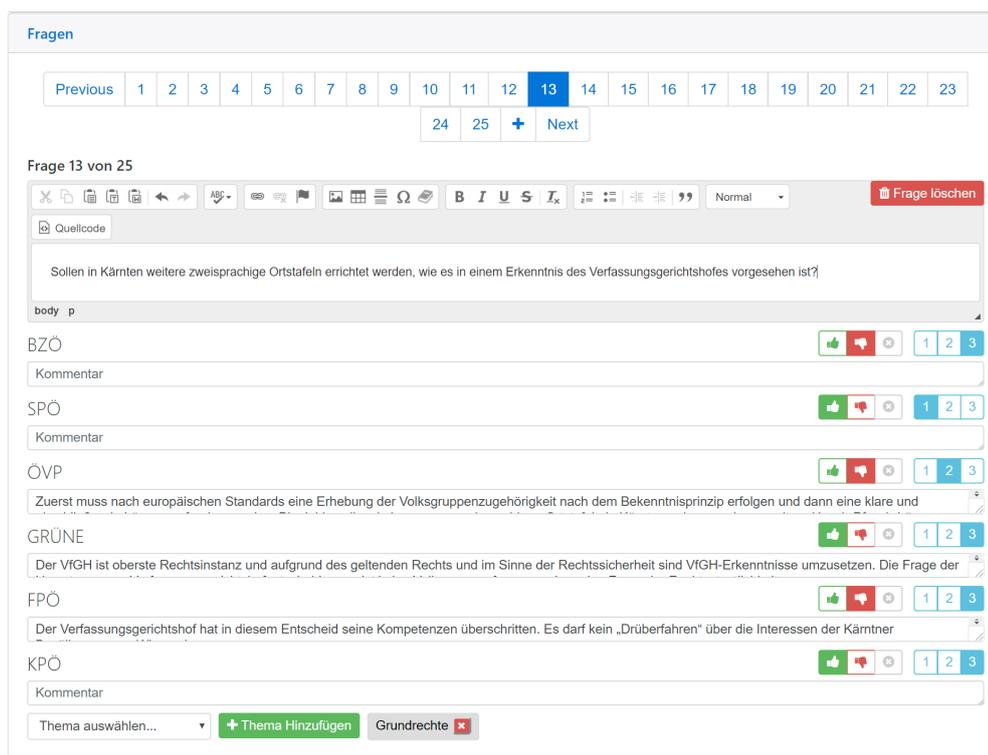


Figure 4.11: Questions editing interface

a detailed statistic of each election can be calculated by selecting one from the list.

Figure 4.12 depicts an example of this detail view. As the entries in the metrics collection are in the order of millions they cannot just be transferred to the client for the calculation. Thus, the statistics are calculated in a map-reduce fashion on the server. The map step takes every answered question and multiplies the consent with the given weight. In the reduce step all resulting scores are added up for each question. The second section in Figure 4.12 shows this combined score together with a bar chart normalized to the highest score. In the first section the scores for each question are multiplied with the answers and weights of the participating parties for each question to show an overall result similar to the result page of the end-user interface.

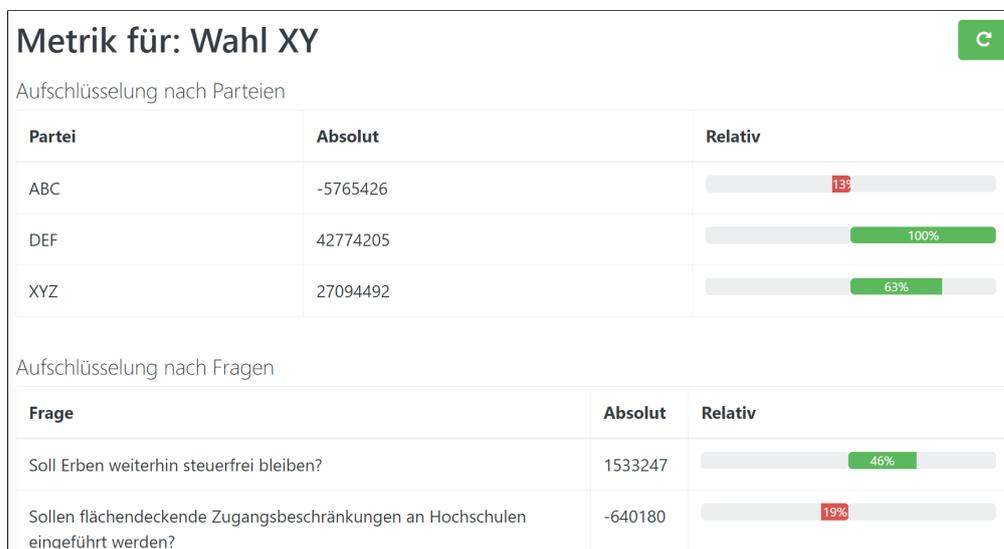


Figure 4.12: Metrics detail view

Hosting

This chapter will show the operational side of wahlkabine.at. In contrast to Chapter 4, which explained the client side, this chapter demonstrates the tools used on the server side. Furthermore, performance improvements are shown that can be applied to many other projects as well. Although it was not necessary in the course of this work, the technology stack was chosen to be scalable. This chapter will demonstrate how this could be done with wahlkabine.at. Last but not least the migration process will be explained and how it helped to maintain the high pagerank in search engines.

5.1 Node Applications

As further explained by Cantelon et. al. [24], the biggest performance improvement of Node in comparison to similar technologies stems from the fact that Node is an event-based system that uses non-blocking I/O-operations. What that means is that Node can serve multiple requests concurrently without getting blocked by a single long running or I/O intensive request as the underlying libraries perform the I/O-operations asynchronously. Whenever there are no events happening, or in other words, there are no requests to handle, the Node application will sleep. This fits our requirement for performance and is one of the reasons why Node was chosen for the application layer.

As recommended by Cantelon [24], when building web application with Node, the usage of the Express framework [25] allows for easier development of standard REST interfaces and tasks such as serving static content. On top of express other layers, such as the security mechanism in the administrator interface, can be easily added.

For the server application Webpack bundles all necessary code into a single file that is executed by Node. It is important to note that the Node process should be executed with restricted rights. Otherwise potential security flaws in Node itself or our application could be exploited to take over the server.

Although the application code is platform independent, there are some third party packages that are not. To handle all included external packages wahlkabine.at uses the npm package manager [26]. To prevent platform-dependent problems from arising, this package manager is executed on the target platform as well as the whole build process. The version information are all stored inside the package.json file and can be constrained in different ways. From exact versions to specific version ranges or even just the latest currently available version, different configurations are possible. This can lead to problems when versions are changing between two installations. Even when only fixed versions are used, sub dependencies could be changing. To prevent these problems, npm introduced a new feature during the course of this work. Lock files are generated that store the installed versions as well as all versions of sub dependencies.

By default Node is installed with support for the english language only. For wahlkabine.at the libraries necessary for full language support are just installed as another npm package. Then, the appropriate environment variable is set to instruct Node to use this package. Otherwise Node would have to be built from source with specific compiler options which would make the maintenance much harder.

In the case of wahlkabine.at there are two Node applications running on the same server, one for each of the two interfaces. Their Node processes are handled with the process manager PM2 [27]. The purpose of this tool is to monitor the applications, write the outputs in the respective log files and restart the applications should they crash for unexpected reasons. Another benefit of using a process manager like PM2 is the possibility to save configurations and automatically restore them after the system restarts.

5.2 Reverse Proxy

When running Node applications it is preferable to not expose them directly to the outside, but to hide them behind a reverse proxy. This section explains what benefits could be gained from this approach.

Nginx [28] is a web server and reverse proxy that excels at serving static files. Figure 5.1, taken from [29], shows the throughput of Nginx in comparison to Apache at different concurrency levels. Similar to Node, the advance in performance compared to the Apache server comes from an asynchronous architecture.

The configuration of Nginx can be held relatively simple as many optimizations are done by default in newer versions. For example the number of worker nodes are automatically fitted to the number of available CPU cores. Almost all configurations are written as nested blocks of so-called directives. On the top level one or more server blocks can be defined with different names. This enables the handling of multiple subdomains on the same machine.

In case of wahlkabine.at this was used to separate the two user interfaces. The administrator is reachable through the subdomain `https://admin.wahlkabine.at` while the end-user interface is located through `https://wahlkabine.at`. For users still

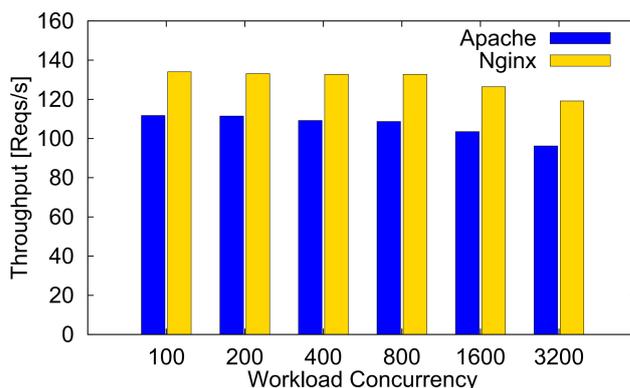


Figure 5.1: Nginx vs Apache throughput

using the sub-domain `www`. a third server block was configured that redirects all of their requests to the correct domain. Configurations shared between directives were put into separate files which get included by each block to apply the Don't-Repeat-Yourself principle.

All servers blocks of `wahlkabine.at` use HTTPS, redirect to the secure version if a user requests the site using HTTP and apply the strict transport security header to all responses. This header instructs browsers to use only HTTPS for a specified time. The required certificates for the secure connection are acquired from Let's encrypt [30], which is a certificate authority that gives away its certificates for free. It is sponsored by big tech companies and has the noble ambition to make the Internet a safer place. The certificates would otherwise pose running costs and additional maintenance work.

To obtain a certificate one has to prove the ownership of a domain to Let's encrypt. This can be done with a tool that also renews the certificates automatically. As Nginx has become a popular web server, the usage of Let's encrypt certificates with it has been heavily automated and became rather easy to setup.

Inside a server block, different so-called location blocks can determine how certain paths of a domain are handled. For `wahlkabine.at` Nginx tries to host all static files without relying on the Node application. Only when Nginx is not able to handle a request it relays it to the respective Node application running on a higher port. This allows for easier control of cache mechanisms for different kinds of files, which are explained in detail in the next section.

Another benefit of having this layer in between the clients and the server application is for Denial of Service (DoS) protection and prevention of brute force attacks. In [31] Specht and Lee identified several countermeasures against DoS attacks one of which being to mitigate its effects. This can be done by balancing the load created by such an attack between multiple instances of one's service. The other option is to throttling clients, meaning to stop accepting requests from a client, if it exceeds certain bound of

fair use. Both of these approaches can be easily implemented by using a reverse proxy.

For the administrator application Nginx measures the rate in which requests are arriving from a specific address. Short, small bursts of requests are allowed as they occur when loading the application for the first time. When the burst limit is reached, and requests occur faster than 1 per 2 seconds, they are blocked. This effectively nullifies the chances of breaking the administrator interface's username and password combination by guessing.

5.3 Performance Improvements

Chapter 3 and 4 showed how high-performance can be achieved with the right choice of tools and their correct implementation. This section will point out performance improvements that concern the way the website is delivered to the clients.

Section 4.2 explains how the end-user interface is rendered on the server and the client. The server side rendering occurs only once, the result being cached in the above mentioned Nginx. Concerning performance this means that the server has to deliver static content most of the time. The speed at which this can be accomplished is limited by the available bandwidth to the client and the size of the transmitted files. The server `wahlkabine.at` is hosted on, already has an upload speed of 1 Gbit/s and the client connection speed is a variable that cannot be influenced by a website's provider. Thus, the common goal of all following techniques is to reduce the number of bytes a server needs to transfer to the client.

5.3.1 Image Compression

In Figure 4.6 the start page is depicted showing a relatively large image as a banner. Especially when adjusted to high resolution screens such graphics can easily take up multiple megabytes. With the proper compression technique and an adequately chosen image quality loss such files can be brought down to the range of around 100 kilobytes. Also not all devices need the high resolution version of an image as they either do not possess the necessary screen resolution or have a different aspect ration leading to large areas being clipped of. For smaller screen sizes different versions of an image can be created and defined inside the CSS. This way only what is really visible is transmitted to the client.

In case of `wahlkabine.at` this leads to differences in file sizes ranging from 39 kilobyte for the smallest version (300 pixels width) up to 167 kilobyte for the largest version (1100 pixels width). For icons it can be beneficial to store them in the format of vector graphics. Vector graphics are rendered on the client and can be scaled to any size without loss of quality. Also they can be inserted directly into the CSS leading to fewer requests to the server.

5.3.2 Gzip Compression

The biggest files transmitted in case of wahlkabine.at are the JavaScript bundles containing the application code. On the one hand these files can be compressed with code minification techniques integrated in the build process, as mentioned in Chapter 4. On the other hand they can be compressed with gzip compression.

As defined in [32] this compression is intended to be used on streams of data and therefore became a standard to compress communications between server and client in the web. Gzip is a general purpose compression method and can be applied to almost any file type, but it excels at text files containing redundancies such as the recurring tag names inside HTML.

Table 5.1 shows how much reduction could be achieved for wahlkabine.at’s JavaScript bundles. It is used on all other types of files as well with the effects being only marginal. Even for dynamic resources Gzip can be applied as the compression algorithm is very efficient. The level of compression can be adjusted depending on how much computational resource is available on the server. A trade-off has to be found between the increased latency due to more computational work and the speed up due to smaller files necessary to be transmitted.

file	original size	gzip compressed
app.js	1.1 MB	162 kB
polyfills.js	348 kB	68.6 kB
vendor.js	3.0 MB	537 kB
total	4.44 MB	767.6 kB

Table 5.1: Gzip compression of JavaScript bundles

5.3.3 Caching

Another technique supported by browsers to reduce the amount of transmitted files is caching. Instead of requesting every resource on every visit, the browser stores the files that probably will not change until the next visit. This behaviour can be controlled by the server through applying corresponding headers to the responses. These headers can even control the behaviour of possible intermediate caches on the way to the client.

In section 5.2 it was pointed out that the reverse proxy is a good location to control these headers. Nginx can be used to serve files differently depending on their type or origin, e.g. the Node application, by using different location blocks. In wahlkabine.at this was used to set the cache control header depending on the files origin.

A typical behaviour expressed with the cache control header is to let the client cache a resource for a specific time until it must be revalidated from the server if the resource has changed. If the server also provided an entity tag (ETag), the client can simply compare if this tag has changed instead of having to download the whole resource again. This

leaves the developer with choosing a time span short enough, such that no stale versions are used by the clients for too long, but at the same time as long as possible to reduce the server load. However, there is an easy trick to prevent the client from even having to check for updated resources which was used for `wahlkabine.at` as well.

All static resources' file names are extended by a hash calculated depending on their content. If, after a new build, a file changes, its name does so too. Now only files that cannot be renamed on changes like the `index.html` or `downloads` have shorter expiration dates. All other files are instructed to be cached for the maximum timespan.

For users visiting the website for the first time all these considerations are meaningless, but for recurring users this reduces the server load drastically while at the same time providing a better user experience. A user requesting `wahlkabine.at` for the first time has to download about 1 megabyte of resources issuing multiple requests to the server in doing so. When visiting again only a single request has to be executed fetching the `index.html` with its much smaller size (292 kilobyte). All other resources are loaded from the local cache.

5.3.4 Using Existing Resources

Not all resources must be loaded from the website's server. JavaScript resources are commonly distributed over CDNs. If two websites use the same resource accessed by the same URL, modern browsers share them between different pages instead of downloading them again, thus reducing data traffic for the clients and speeding up the page load.

For `wahlkabine.at` one of the requirements is to not use any CDNs partly because of users' privacy concerns and partly because of the extra dependencies that must be maintained. Therefore the possible shared dependencies were packed inside the vendor bundle and client side caching is used to prevent the clients from loading them too often.

Another resource even more reliably available on the client are fonts. Every system and platform has its own characteristic font family. For `wahlkabine.at` all those fonts are listed in the corresponding CSS instruction. The browser iterates through this list and picks the first font it knows. This serves two purposes. As mentioned above, the fonts available on the client machine do not have to be loaded from the server. This increases the user experience and perceived performance, as all textual content can be displayed even before all of the other resources finished loading. Another benefit for the user experience is that the site feels even more like a native application.

According to Tim Frick [33] system fonts are the best option for websites when it comes to sustainability. If the design choice requires the loading of other fonts he recommends web fonts over embedded fonts as they can benefit from the same shared usage as JavaScript libraries mentioned above.

5.3.5 HTTP/2

The increasing demands of modern web applications led to the conquering of problems within the hypertext transfer protocol (HTTP). A new major version [34] was introduced and is becoming supported by browsers more and more. This leads to the question if servers should already adopt to the new standard as well.

HTTP/2 tries to improve the performance of websites by allowing a single TCP connection to be used for multiple streams concurrently. Furthermore, it uses a binary format and header compression to reduce the overhead when doing multiple requests. Another new feature, called server push, allows the server to send resources to a client without them requesting the resources explicitly beforehand. The client caches these additional resources and saves the round trip to the server when they need it.

Nginx does not support server push, as of the time of writing. However, almost the same benefit can be achieved with another technique called inlining. Styles or at least parts of it can be included directly into the requested HTML as an inline-CSS. This can already safe one very important request. As mentioned in Section 4.2, wahlkabine.at includes the API responses as an inline script block too. The bundling of JavaScript resources into three files and the inclusion of all icons into the CSS further reduce the amount of requests, and therefore the overhead, due to headers.

Varvello et. al. analyzed the support and usage of HTTP/2 in [35] with the realization that many of the biggest websites on the Internet support the new standard, but only a few of them use it primarily to deliver their content. Saxcé et. al. showed that HTTP/2 often reduces the performance of websites, when used in cellular networks, because it is negatively impacted by package loss [36]. However, both sources also mention that together with other techniques, such as the above mentioned inlining the performance is improved nevertheless through HTTP/2.

5.4 Scaling Out

Although many efforts have been made to increase the website's performance without having to increase the computational resources, there is a limit to what a single server can handle. When this limit is reached, it is easier, and often cheaper, to add an additional machine (scaling out) than to replace the machine with a faster one (scaling up). This is especially the case when the ongoing operation must not be interrupted. wahlkabine.at's technologies were already chosen with scalability in mind. In the following it will be shown how the database and the application layer can be scaled out independently.

5.4.1 Database Scaling

When creating a cluster of MongoDB instances, two different goals can be achieved depending on the configuration. If a collection increases in size and many writes must be handled, the collection can be split up over multiple instances. Each of the instances then

manages a part of the data, a so-called shard, and executes the operations designated to its share. Similar to an index, a sharding key dictates which shard records which documents. Thus, sharding can improve the write and read performance, but additional instances are necessary to act as configuration servers.

The other goal is to increase the fault-tolerance by replicating the data on multiple instances. MongoDB accomplishes this by using one instance as a primary node and several others as secondary nodes. All write operations are handled by the primary node and propagated to the secondary nodes. Should the primary node fail, an election is held between the secondaries to designate a new primary. To aid this election process an extra arbiter node can be necessary to have an uneven number of votes for the election.

Although almost all of the operations are handled by the primary node the secondary nodes can increase the read performance as they all have a complete set of data. Mongo database drivers have different strategies for selecting an instance from a replica set to read from. If, for instance, the nodes are located in different geographical locations the driver can select the nearest node to read from, as it presumably has the lowest latency. Depending on the selected write and read quorum, as well as other configurations, it is possible that a read on a secondary node returns stale data.

For larger applications both of the presented techniques can be combined to obtain their respective benefits. In this case each instance of a shard would be represented by a whole replica set.

5.4.2 Application Scaling

The platform independence of `wahlkabine.at` makes it relatively easy to deploy on any cloud computing environment. This was initially planned should the self hosted server not suffice for the load, as it was the case for the previous version of `wahlkabine.at`.

Chieu et. al. presented a simple architecture and algorithm, that allows applications to scale according to user demands in a cloud environment [37]. This architecture requires a load balancing node that redirects the clients requests to one of several application servers all operating on the same database. In the case of `wahlkabine.at` the necessary replication of the application layer and load balancing between them is rather easy, as no state information is stored on the server. Otherwise the load balancer would need to store some kind of information about the sessions as well. The task of load balancing could be accomplished by the Nginx as it acts as an entry point to the server. To maximally utilize multi processor machines the PM2 process manager can automatically spawn as many instances of the same Node application as there are processors available on the given machine.

According to Chieu's architecture an additional manager entity must monitor the utilization of each machine. If a certain capacity is reached new instance of the application server are started to compensate for the additional demand. To cope with sudden changes in load it is important to be able to startup instances quickly. The Node application

used for wahlkabine.at already does this as no ORM and no complicated initialization procedure is necessary. In cloud environments the machines are virtualized to instantiate and dispose them quickly as well as to optimize the load of the underlying physical machines.

5.5 Migration

Prior to this work, wahlkabine.at covered 37 elections and other events. Theses included 830 questions with 5304 answers in total. Additional meta information such as the editors and participating parties with their names, abbreviation, and homepage, as well as all comments to their answers were included in the old website's pages as well.

Thankfully it was possible to export the answers and questions into comma-separated values (CSV) files by the site's providers. As part of this work a program was written that crawls through the previously existing version and extracts all necessary meta information in addition to loading the data from the CSV files. This program then built documents according to the new data structure (2.1) to import them in the new database.

Another goal of this migration was to keep the high pagerank wahlkabine.at has built up over the years. The pagerank of websites on Google is based on a simple thesis, as described by Langville and Meyer: Interesting pages are the ones linked often by other interesting pages [38]. Although Google and other search engines have evolved far beyond this metric to calculate their pagerank it has still a major impact.

This is why, to maintain the page score, the above mentioned program also created a mapping of all existing pages and subpages to their respective new locations on wahlkabine.at. From this mapping a list of redirect rules was created and added to the Nginx configuration. Thus all existing inbound links to wahlkabine.at stay valid and contribute to the pagerank.

The following chapter shows the impact of all the performance improvements mentioned in this chapter and how the now totally different architecture compares to the previous version of wahlkabine.at.

Evaluation

In this chapter the performance of the new site will be compared to the previous version of `wahlkabine.at`. The qualities of scalability and maintainability will be covered as well. Unfortunately, no meaningful load test could be executed on the old system as it was not fully functional any more after the last regional election it was used for. What can be compared though are the initial loading times of the start pages and the number of users over the course of weeks preceding an election.

6.1 Performance

To get an unbiased measurement of the both versions' performance they are tested with an external tool that simulates the initial request of the site for a user that has no files cached and owns a high bandwidth connection. Pingdom [39] is a provider for monitoring solutions and offers a convenient tool to speed test websites from different locations around the world.

Figure 6.1 and 6.2 show the results of those test when using the nearest location to the Vienna based server (Stockholm). The test reveals that the loading time of the new page is only two thirds that of the old version. This is achieved although the total size has roughly doubled. Further details about the result shown by this tool reveal the cause of the longer loading time. The previous version of `wahlkabine.at` does 53 almost sequential requests to the server whereas the new version only needs 16. Figure 6.2 shows 26, but 10 of those are icons embedded in the CSS and therefore loaded without issuing a request to the server.

Another thing that has to be considered are returning users. The old version prevented the caching of some resources because it did not use the cache breaking hashed file names, as shown in Section 5.3.3. This results in around 210 kilobytes needing to be downloaded when visiting the website again. The new version would theoretically perform better, but

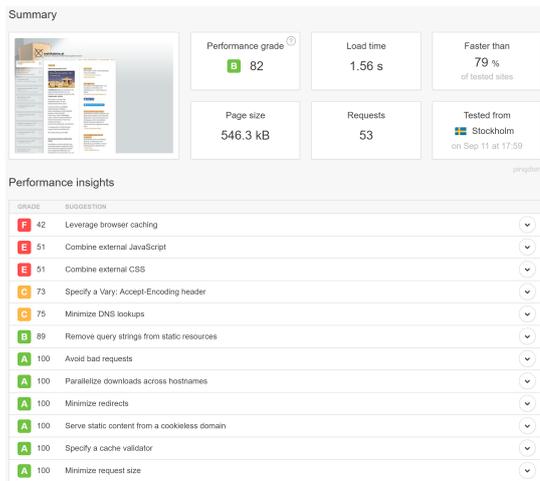


Figure 6.1: Speedtest old version

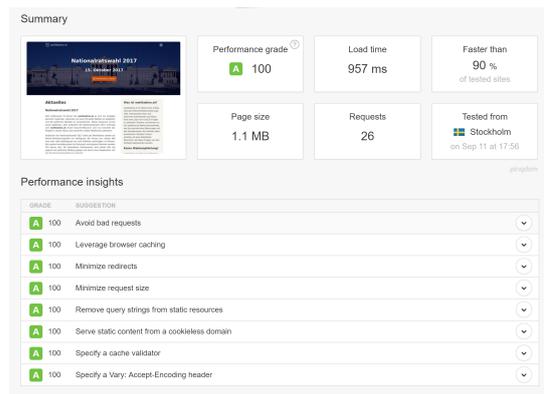


Figure 6.2: speedtest new version

due to the inlining of the CSS the single html response that is loaded for returning users has 292 kilobyte. In general the optimizations used were done in favour of new users instead of recurrent ones, because a large part only uses the page once.

Furthermore, the new version results in less overall data traffic. As usage statistics have shown the majority of users only visit the site to answer the questionnaire of the newest election, which is why this data is included when loading the start page. To walk through the common use case of the website no further communication with the server is necessary. This eliminates the previously existing problem of mobile users having to restart the questionnaire after losing connection.

Figure 6.3 shows the logged questionnaires in September. This chart does not distinguish between prematurely aborted or finished questionnaires as both generate an equal amount of load for the system. The following comparison should help to get a feeling for the performance of the new version of wahlkabine.at. The total number of users requesting the site during the weeks preceding the last regional election in 2015 were 160,000, which took the previous version to its limits. The same number of users were reached in only 48 hours after publishing the questionnaire for the general election of 2017, on the same hardware resources.

6.2 Scalability

For the general election in 2013 the old version needed to be hosted on external resources to cope with the total number of one million users. The new version has served, as of the 6th of October 2017, 867,000 users on the self hosted server without signs of flagging.

Figure 6.3 depicts how heavy fluctuations in server load can be and how fast they can occur. The only peak that could be somewhat foreseen was the one happening on the 4th

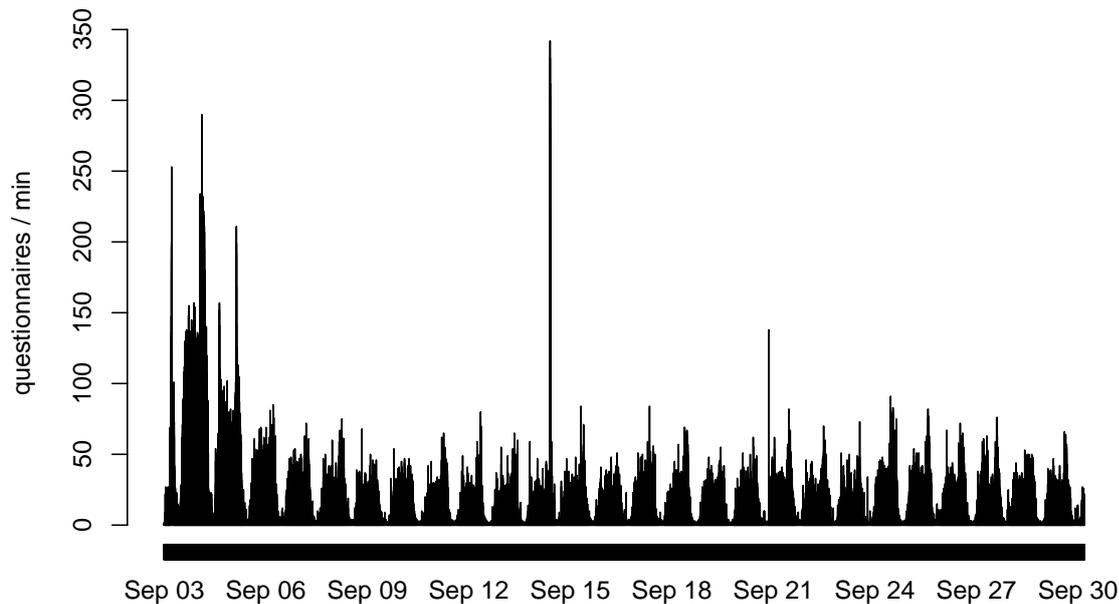


Figure 6.3: Usage September

of September due to a press release about the relaunch of the website. Because journalists of major Austrian newspapers were involved in the compilation of the questionnaire, the launch was mentioned in their respective newspapers as well. However, an Austrian news anchor was even faster and spread the information on Twitter the night before, which can be seen as the first peak in Figure 6.3.

In Figure 6.4 the peak happening on the 14th of September is depicted. The popular news show ZIB (German: Zeit im Bild), starting on 20:00, mentioned various VAAs in a segment about the upcoming general election. The smaller peak in 6.4 reflects the mentioning of `wahlkabine.at` in the news overview at the beginning of the show.

What can be learned from this graphic is how fast applications need to be able to scale to accommodate load peaks. Suppose the application is hosted in a cloud environment, as proposed in 5.4.2, and each instance can handle about 150 clients per minute. If instances are started as early as at 60% of the total capacity, the startup of another instance would have to happen in a matter of 1 to 2 minutes before the system is overloaded.

The architecture was chosen with this startup time in mind, although it does not matter for the functionality of the website if log entries are written fast enough. The real performance bottleneck would be the reverse proxy. With a page size of 1.1 megabyte and an upload of one gigabit, the current server could deliver the website to little over 100 clients per second. This should eliminate the need for scaling out for years to come.

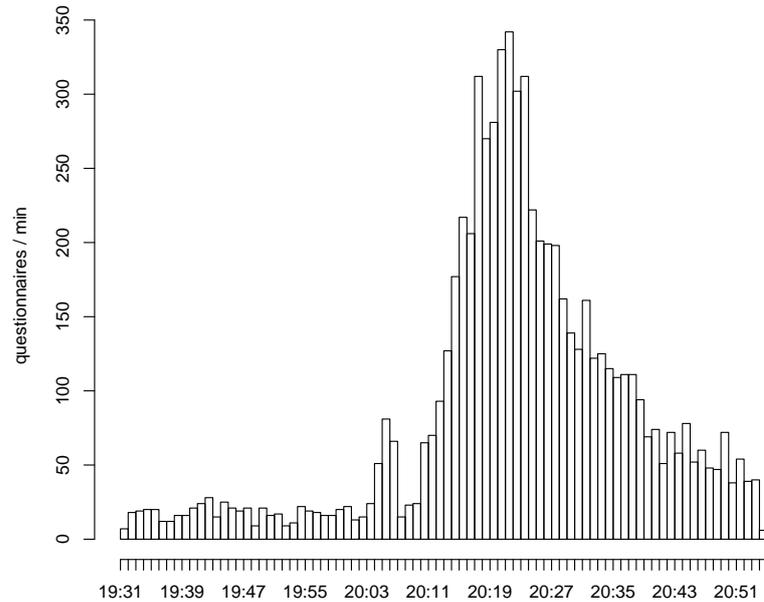


Figure 6.4: Peak load on 14th of September

6.3 Maintainability

Besides performance and scalability the architecture was chosen to be long-term maintainable. A common language for the server and client code was chosen to easier find developers being able to understand and change the software. All tools were selected with their popularity and community size in mind to increase the probability of them being still supported in the next decade. The database and application structure were chosen in a way that allows for uncomplicated implementation of additional fields needed for future features. The subdivision of the application code into small independent components is inherent to Angular applications and further contributes to a long-term maintainable code base.

However, only time will tell if the chosen technologies will be supported in the next decades and although Angular is supported by Google there is no guarantee for it being maintained in the near future.

A final remark must be made concerning the user experience of `wahlkabine.at`. Aside from all the technological means shown in this work it was due the work of Alexandra Geier [3] that the new version has an aesthetic which appeals to users.

Conclusion

7.1 Achievements

This work presented aspects on how to choose technologies for high performance applications, with respect to serving as much clients as possible concurrently. All layers of the classic three tier architecture were covered. For the running example of `wahlkabine.at` the classic MEAN-stack was implemented with slight deviations. Instead of AngularJS, Angular2 was chosen as the front-end framework.

This work elaborates how Angular's inherent SPA-style can reduce the computational load of servers and enhance the user-experience. With a special feature called Angular Universal, even the perceived performance could be enhanced. Pre-rendering the SPA on the server also allows bots to interpret the website, which is a the common problem amongst SPAs.

This work points out the usefulness of the Nginx reverse proxy for hosting the web application. As shown in the previous chapter, the new version is able to serve several hundred users per second with the same hardware as the previous version. Many of the presented techniques to improve the website's performance can be translated and used on other projects as well.

The most important goal achieved concerning `wahlkabine.at` is that it can continue to be run on the self hosted servers and therefore stay cheap in terms of ongoing costs. Performance-wise there are still many concepts that could be looked into and maybe reasonable to apply to this kind of application mentioned in Section 7.3.

Prototypes could be presented early on in the development process due to a self hosted staging environment. This aided the requirement analysis as many ideas on how to implement certain features could simply be drafted and played through with test data instead of relying on static mock ups.

Because of the chosen technologies and the learnings from the staging system, the installation of the stack on the production environment was relatively easy, although it had a different operating system. What would have improved the portability even more would be the usage of Docker containers to bundle the application [40]. Since the single self hosted server proved to be powerful enough, the additional virtualization coming with the usage of Docker would have been more of a hindrance performance-wise.

The newly created administrator interface now enables the editors to import articles and questionnaires themselves. Aside from relieving the developers this allows the editors to test the questionnaires in the application. Previous to this work this had to be done in spreadsheets.

7.2 Problems

Angular Universal started out as a side project and has been integrated into the core of the framework later on. Unfortunately, at the time of this work, there is still no mechanism to transfer the state of the server's application to the client. To compensate this lacking feature a third party dependency was included. This dependency had dependencies itself to the server and client platform code of Angular, which leads to the increased bundle size shown in Table 5.1. Although this has a bad impact on the page size and consequently the loading time of the website, it is more important to have this feature and reduce the server load due to fewer API calls. Hopefully a better way of handling the state transfer will be included into the Angular core in the near future.

The production environment is hosted on a server running CentOS 7. Although all used technologies are available for this operating system as well, the application could not be run in the same way as on the Ubuntu staging server. SELinux prevented Nginx from connecting to the node application and accessing the files inside the build folder so this security mechanisms was disabled. Furthermore, a bug with PM2 and CentOS prevented the node application from restarting whenever the server is rebooted. The most frustrating issue, however, was the lack of HTTP/2 support of the Nginx because of an outdated OpenSSL version included in CentOS. The workaround for this problem would be to compile the Nginx from source with special compiler options, but this would violate the requirement of maintainability.

7.3 Outlook

Aside from fixing the state transfer solution, the most promising performance improvement would be to include service workers [41]. Theses can be seen as proxies running on the client intercepting the communication to the server or issuing communication themselves. They would allow an application, like wahlkabine.at, to be mostly run offline as well which obviously would further reduce the load of the server. A first preparation to include them was already done by including the manifest file generation into the build cycle.

Another improvement would be to make the website applicable to Google's AMP cache [42]. To do so the HTML would need to incorporate some changes and an additional JavaScript would need to be included. In return for this efforts, parts or even the whole site's content could be delivered from Google's servers should a client use the right browsers to view them.

Images can take up an enormous part of a page's size. This is why Google developed a new image format to compress images even further called WebP [43]. It supports both lossless and lossy compression and can, according to Google, decrease the file size up to 26% compared to PNG and 34% compared to JPEG. Unfortunately this format is at the time of writing only supported by the Chrome and the Opera browser. If the browser usage changes over the next decade optimizations, such as these, could become superfluous.

Building a maintainable and future-proof application will always be one of the hardest disciplines in software development as it requires the developers to anticipate how software will evolve and which technologies will still be used in years to come.

List of Figures

2.1	Class diagram	12
3.1	Performance of read operations	18
3.2	Performance of write operations	18
3.3	Performance of fetching all keys	18
3.4	Throughput of different frameworks	20
3.5	Front-end frameworks item load time	22
3.6	Front-end frameworks item update time	22
3.7	Front-end frameworks initial load time	22
4.1	SPA server communication	23
4.2	JIT performance	26
4.3	AOT performance	26
4.4	Bundle composition with JIT - 2.72 MB	27
4.5	Bundle composition with AOT - 2.24 MB	27
4.6	End-user interface	29
4.7	Universal application loading	30
4.8	Glossary entry popup	33
4.9	Administration interface	34
4.10	Article editing interface	35
4.11	Questions editing interface	36
4.12	Metrics detail view	37
5.1	Ngnix vs Apache throughput	41
6.1	Speedtest old version	50
6.2	speedtest new version	50
6.3	Usage September	51
6.4	Peak load on 14th of September	52

Acronyms

AOT ahead of time. 25–28, 32, 57

API application programming interface. 21, 25, 28, 30, 32, 33, 45, 54

CDN content delivery network. 2, 44

CLI command-line interface. 25, 28

CMS content management system. 9, 14–16

CSS Cascading Style Sheets. 2, 10, 14, 24, 26, 28, 32, 42, 44, 45, 49, 50

CSV comma-separated values. 47

DOM document object model. 21, 24, 25, 30

DoS Denial of Service. 41

ETag entity tag. 43

HTML Hypertext Markup Language. 2, 10, 11, 14, 21, 23, 24, 31, 32, 34, 43, 45, 55

HTTP hypertext transfer protocol. 45

HTTPS Hypertext Transfer Protocol Secure. 32, 33, 41

ICU International Components for Unicode. 28

JIT just-in-time. 25–28, 32, 57

JS JavaScript. 14

JSON JavaScript object notation. 20

MVC model view controller. 24

NoSQL Not only SQL. 16, 18, 19

ORM object-relational mapping. 20, 47

REST representational state transfer. 20, 33, 39

SASS Syntactically Awesome Style Sheets. 28

SEO search engine optimization. 11

SPA Single Page Application. 5, 21, 23, 24, 29–31, 53, 57

URL uniform resource locator. 9, 11, 34, 44

VAA Voting Advice Application. 1, 51

WYSIWYG what-you-see-is-what-you-get. 13, 34

Bibliography

- [1] Diego Garzia. The effects of VAAs on users' voting behaviour: An overview. *Voting Advice Applications in Europe: The state of the art*, pages 13–33, 2010.
- [2] wahlkabine.at. <https://wahlkabine.at>. Accessed: 2017-09-16.
- [3] Alexandra Geier. Design and implementation of a future-friendly user interface for voting advice applications. Master's thesis, Vienna University of Technology, 2017.
- [4] Keith H. Bennett and Václav T. Rajlich. Software Maintenance and Evolution: A Roadmap. pages 73–87, 2000.
- [5] Blogger. <https://www.blogger.com>. Accessed: 2017-09-16.
- [6] Vikram Dhillon. Static Site Generators. In *Creating Blogs with Jekyll*, pages 21–33. Apress, 2016.
- [7] M. Jazayeri. Some Trends in Web Application Development. In *Future of Software Engineering, 2007. FOSE '07*, pages 199–213, May 2007.
- [8] S. K. Patel, V. R. Rathod, and J. B. Prajapati. Comparative analysis of web security in open source content management system. In *2013 International Conference on Intelligent Systems and Signal Processing (ISSP)*, pages 344–349, March 2013.
- [9] Jing Han, Haihong E, Guan Le, and Jian Du. Survey on NoSQL database. In *2011 6th International Conference on Pervasive Computing and Applications*, pages 363–366, October 2011.
- [10] Y. Li and S. Manoharan. A performance comparison of SQL and NoSQL databases. In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 15–19, August 2013.
- [11] MongoDB. <https://www.mongodb.com>. Accessed: 2017-09-29.
- [12] Zachary Parker, Scott Poe, and Susan V. Vrbsky. Comparing NoSQL MongoDB to an SQL DB. In *Proceedings of the 51st ACM Southeast Conference*, pages 5:1–5:6, 2013.

- [13] Stackshare. <https://stackshare.io>. Accessed: 2017-09-17.
- [14] Node. <https://nodejs.org>. Accessed: 2017-09-26.
- [15] PHP. <http://www.php.net>. Accessed: 2017-09-26.
- [16] Django. <https://www.djangoproject.com>. Accessed: 2017-09-26.
- [17] K. Lei, Y. Ma, and Z. Tan. Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js. In *2014 IEEE 17th International Conference on Computational Science and Engineering*, pages 661–668, December 2014.
- [18] Eric Molin. Comparison of Single-Page Application Frameworks. 2016.
- [19] M. Takada. *Single page apps in depth, Mixu's single page app book*. 2013.
- [20] Webpack. <https://webpack.github.io>. Accessed: 2017-09-26.
- [21] Vote match europe. <http://www.votematch.eu>. Accessed: 2017-09-25.
- [22] Redux. <http://redux.js.org>. Accessed: 2017-09-26.
- [23] Bootstrap. <http://getbootstrap.com>. Accessed: 2017-09-26.
- [24] Mike Cantelon, Marc Harter, TJ Holowaychuk, and Nathan Rajlich. *Node.js in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2013.
- [25] Express. <https://expressjs.com>. Accessed: 2017-09-26.
- [26] NPM. <https://www.npmjs.com>. Accessed: 2017-09-26.
- [27] PM2. <http://pm2.keymetrics.io>. Accessed: 2017-09-26.
- [28] Nginx. <https://nginx.org/en>. Accessed: 2017-09-26.
- [29] Qi Fan and Qingyang Wang. Performance comparison of web servers with different architectures: a case study using high concurrency workload. In *Hot Topics in Web Systems and Technologies (HotWeb), 2015 Third IEEE Workshop on*, pages 37–42. IEEE, 2015.
- [30] Let's encrypt. <https://letsencrypt.org>. Accessed: 2017-09-27.
- [31] Stephen M. Specht and Ruby B. Lee. Distributed Denial of Service: Taxonomies of Attacks, Tools, and Countermeasures. In *ISCA PDCS*, pages 543–550, 2004.
- [32] L. Peter Deutsch. GZIP file format specification version 4.3. 1996.
- [33] Tim Frick. *Designing for Sustainability: A Guide to Building Greener Digital Products and Services*. " O'Reilly Media, Inc.", 2016.

- [34] Mike Belshe, Martin Thomson, and Roberto Peon. Hypertext transfer protocol version 2 ([http/2](http://http2.org/)). 2015.
- [35] Matteo Varvello, Kyle Schomp, David Naylor, Jeremy Blackburn, Alessandro Finamore, and Konstantina Papagiannaki. Is the Web HTTP/2 Yet? In *International Conference on Passive and Active Network Measurement*, pages 218–232. Springer, 2016.
- [36] H. de Saxcé, I. Oprescu, and Y. Chen. Is HTTP/2 really faster than HTTP/1.1? In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPs)*, pages 293–299, April 2015.
- [37] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal. Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment. pages 281–286, October 2009.
- [38] Amy N. Langville and Carl D. Meyer. *Google’s PageRank and beyond: The science of search engine rankings*. Princeton University Press, 2011.
- [39] Pingdom. <https://www.pingdom.com>. Accessed: 2017-09-30.
- [40] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.
- [41] Service Worker. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API. Accessed: 2017-10-01.
- [42] Google AMP. <https://developers.google.com/amp>. Accessed: 2017-10-01.
- [43] WebP. <https://developers.google.com/speed/webp>. Accessed: 2017-10-01.