Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

FELIX MÜLLER
BACHELOR THESIS

UNI1 MONOLITH TO COMPONENTS

Submitted on 2 July 2021

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 2 July 2021

License

Erlangen, 2 July 2021

Abstract

This thesis discusses the refactoring of the pre-existing Uni1 application from a monolith to multiple components. The refactoring results into two components, one for the account management containing the login, registration and account management and the other the old marketplace, where new projects can be created. In addition to that, another completely new component called Campaigner is implemented. This new component enables specific users to campaign their projects via emails which are user created. Lastly continuous integration and continuous deployment is added which deploys the application to Amazon Web Services.

Contents

1 Introduction

1.1 Uni1

1.1.1 Old Uni1

The application of this thesis builds upon a pre-existing NodeJs application with a ReactJs frontend, both written by Philipp Eichhorn. The main relevant part of the old system, is his marketplace component, which will be translated to another framework and reused.

1.1.2 New Uni1

The main purpose of the newer version of Uni1 remains the same as for the previously mentioned older version. This purpose "is to revolutionize how universities and companies collaborate and conduct business with one another" (Eichhorn, 2016).

The newer version consists of more components than the old one, was developed together with Nasser Eddin Nasser and is available at https://dashboard.uni1.de. It has to be noted that, unlike this thesis, the frontend is in German, like it was in the old application.

1.2 Thesis goal

The goal of this thesis can be split into three separate aims. The first is about the refactoring of the old/pre-existing application from a monolith into separate components for the Marketplace and the Account management, meaning the user authentication/login.
The second is to create a new component and integrate it into the, now split, application. This component should enable an authorized user to create a campaign, which sends campaign emails to a selected group of contacts. Contacts are a new model to be implemented, because it has to be possible to send campaign

emails to email addresses even though this email address is not used by any user. Of course, the recipient still has to be able to unsubscribe from these emails.
Lastly this component should also be able to receive and handle Out-Of-Office replies in order to display them and make it possible to detect whether a contact has changed its email address, for example due to change of employer, if a new email address is provided in the reply.
The last goal is to implement continuous integration and continuous deployment for the project, which tests the backend and deploys it to Amazon Web Services (AWS), where the new application is hosted.

1.3 Thesis structure

This thesis is structured using the provided design inspired by 'Design Science'[1].
After this introduction the thesis will begin with the problem identification chapter, where a short explanation will be given, about the reason for this thesis, or rather the application developed behind it.
This is then followed by the chapter listing the requirements for the new application, which will be later used for the evaluation.
The fourth chapter contains the solution design, meaning how the problems from the previous chapter are about to be solved. This includes the architecture of the new application, as well as different technologies used.
Next comes the main chapter detailing the implementation. In here each component that was implemented will be detailed as well as how the infrastructure was set up. This also includes the setup of the pipeline, meaning the continuous integration & continuous deployment.
The fifth chapter contains the demonstration, where four main workflows of the new application are detailed. These workflows are all demonstrated using the frontend, because that makes it easier to explain and understand.
In the sixth chapter the implemented solution, as detailed in chapter four, will be evaluated against the requirements, as previously mentioned. It will also highlight certain issues that stuck out after the implementation and propose future adjustments.
After the evaluation the thesis will finish with the conclusion.

[1] https://www.jstor.org/stable/40398896?seq=1

2 Problem identification

The following chapter gives a short explanation concerning the circumstances, which led to this thesis.
As previously mentioned, there already existed an Uni1 application, but this application was purely for the discovery of interesting projects or creating own project requests, lacking the possibility to promote these projects. This thesis is supposed to change that with the implementation of a new component for sending emails to a list of email addresses and, most importantly, manage Out-Of-Office replies. This handling of the replies makes it easier for the creator, of this so-called campaign, to detect if one of his contacts changed his email address or if there is a new contact person, due to changes in the company/university. Currently this also has to be done manually and is done, by Prof. Riehle as an example, using a large excel file with multiple hundreds of emails, sorting through the email inbox and updating outdated entries.
In addition to that some changes to the user were desired, mainly due to the unwanted strict separation between companies and university employees. Because of this, and the new component, a refactoring of the old application was required, in order to, firstly, apply the mentioned user changes. Secondly the new component brought up the desire to split the existing monolith into separate components for the user handling and the Marketplace in order to simplify the creation of other, future, components.

3 Objective definition

This chapter contains the requirements for the new application and is split into sections for each of the pre-defined resulting components, as well as a section for the continuous integration & continuous deployment.

3.1 Marketplace (Old application)

The Marketplace, meaning the old application, is to be reused after some refactoring and updating.
For the refactoring, it is necessary to split the existing monolith into separate components, such that it would be fairly easy to implement and integrate a new component, or even split the application into multiple microservices in the future without much additional effort.

3.1.1 Backend

The first requirement is the extraction of the account management, meaning the login and account editing, as well as the refactoring of the authentication and authorization, which will be detailed in the next section.
The next requirement is the removal of university and company logic, which entails the deletion of the models for the database, refactoring of the user model, as well as the removal of all related apis and filters.
Lastly the biggest requirement is, that the Marketplace works as well as it did before and without any features removed, aside from previously mentioned ones, like the removal of company and university.

3.1.2 Frontend

Analogue to the backend the first requirement is the extraction of all pages concerning the account management as well as the authentication. These pages are the login, the registration and the account management page, where the user can edit himself.
Aside from that, the main requirement, that is to be done for the frontend of this

component, is the translation to VueJs, in order to match the other components. This also requires research whether the old styling framework, called Semantic UI, can be ported to Vue. If it can not be ported, an alternative has to be found and used.
The next requirement is the merging of the different pages for university and company employees, in order to accommodate the role changes in the backend. This requires the identification of elements on both versions of these pages, that are to be displayed on the new, merged, page.

3.2 Dashboard

The Dashboard is the new component, which houses the login, registration and account management page and functions as a connection between all components.

3.2.1 Backend

The backend requirements start with the writing of the login, registration and account editing logic in this new component that works for all components, meaning the login and registration are only required once for all components. For this the extracted logic can be used as a starting point.
The next one is to refactor the existing authentication and authorization used by the api endpoints to work with the refactored logic.
The last requirement is the refactoring of the roles from university and company to roles for each component, meaning user for Marketplace component access, campaigner for Campaigner component access and admin for the Admin Console component access. It has to be noted, that the last component is not a part of this thesis, but rather of the master thesis by Nasser.

3.2.2 Frontend

The first requirement for this is the same as for the backend, meaning a working login, which counts for all components, where a login in this component would also count as a login in, for example, the Campaigner component.
Derived from this requirement, another one is a registration page that allows users to create a new account. One important part is the fact, that the login as well as the registration work based on a username, not the email of the user, in order to make it unnecessary for the user to remember the changed login data.
Another depending requirement is the account editing page, which allows the user to edit all of the important attributes of himself. These attributes are the first name, family name, email, phone number, title, and the tags he wants to receive campaign emails for, which is detailed in the section for that component. In addition to that, he needs some way to manage his email addresses used for

the Campaigner, meaning he has to be able to delete them, create new ones and disable campaign emails for one specific address.

3.3 Campaigner

3.3.1 Backend

Most of the following requirements are resulting from the frontend requirements. For starters a new model is required in the database called contacts. These contacts, more or less, only consist of an email, as well as the tags they have subscribed to, and will be used by this component to send emails. The important part is, that contacts can exist without any users they are assigned to, but each user has at least one contact with the email the user uses for his account.
Along with the contact model the tags, which were stored inside a project request of the Marketplace, need to be extracted into their own model, for the campaign emails recipient selection.
After this the first big, and perhaps most important requirement, is the implementation of a service for sending batch emails using the chosen email provider. This service has to send the emails to each recipient individually and not to a group. Additionally the creation of a simple email template, which contains predefined footer and a injectable body, is required.
The second, dependent on the first, requirement is the receiving and handling of Out-Of-Office (OOO) responses. This api has to parse the incoming OOO, save the relevant data, like sender, subject and body, and extract emails from the body, which can then be used in the frontend to easily create new contacts from the original contact which sent the OOO.
Due to the beforehand mentioned creation of new contacts, this component also needs an api for creating these new contacts which differs only in email from its original contact.
Last but not least, the next requirements are derived from the frontend and is about the backend part of the api, with the first one being a create api for creating a new campaign which saves it and initiates the email sending. The second one is a preview api which returns the already mentioned template along with its user provided body and the last one is the apis for retrieving one or all campaigns.
The last two requirements are about an additional attribute of the campaigns, the status. The first one is the status itself, which is supposed to indicate whether the campaign is still sending emails, finished sending or was closed by the user. The next one is for the user to be able to start and stop the email sending.

3.3.2 Frontend

Due to the fact, that this is a completely new component, it will require a little bit more work in terms of the number of requirements to fulfill.
The first requirement is a page where the user can create a new campaign. This requires the input of at least a name, subject and body as well as a recipient selection using the campaign tags, where empty selection means to send to all available contacts. After inputting all of the above, there has to be a preview before sending the emails.
The next requirement, or page, is the campaign details page, which displays all important information about the campaign and has some method for finishing the campaign as well as another for handling Out-Of-Office responses. Additionally the creator should be able to start and stop the email sending here, as mentioned in the backend subsection.
The received OOO can also be viewed in detail with its sender, subject as well as its content body on their own details page. Additionally, there are some actions available for the handling of the OOO, like deleting of the contact which sent the response. Another is the creation of new contacts using the emails, that were scrapped from the OOO response, and the last is to simply be able to tell the application to ignore this OOO.
The last required page, is the campaign list page, where all campaigns are listed and from which it is possible to open the previous two pages. The page itself is a list with one entry for each campaign which can be used to open the details page for the corresponding campaign. Each list entry has to show at least the previously mentioned status of the campaign.

3.4 Continuous integration & Continuous deployment

The requirement for the continuous integration is to adhere to the standard procedure of CI, meaning to test the code on every push to the repository.
Concerning the continuous deployment, the requirement is the designing of a concept for the deployment of the application to Amazon Web Service (AWS) and to implement it, such that the application gets deployed, when the code gets pushed onto a certain branch. This of course applies to the backend, as well as the frontend.

4 Solution design

The application uses the classic client/server architecture, meaning the frontend/client and backend/server are separated from each other and communicate using a REST Api.
It has to be mentioned, that Uni1 is dependent on AWS due to its usage of many different services, all of which will be named and the integration of each will be explained in the relevant subsections.

4.1 Backend

The backend is a NodeJs application using the Express[1] framework, which is one of the most frequently used NodeJs framework for Web-Applications.

The Database, used for storing the application data, is a MongoDB database hosted in MongoDB Atlas[2], setup by Nasser. The reason why Atlas was chosen, is the fact that Atlas is one of the cheapest and, at the same time, reliable providers for MongoDB Cloud Storage, providing the possibility to distribute your database cluster across multiple cloud storages and even providers. This means that part of your database is stored within, for example, AWS and another within Google Cloud.

For hosting, AWS ElasticBeanstalk[3] (EB) is used, where the application runs as a Docker[4] container. The usage of Docker makes it easier to deploy the backend outside of ElasticBeanstalk.
Additional AWS Services used by the backend are AWS CloudWatch[5] for logging of the application and AWS Simple Email Service[6] (SES) as the email provider

[1] https://expressjs.com/
[2] https://www.mongodb.com/cloud/atlas
[3] https://aws.amazon.com/elasticbeanstalk/
[4] https://www.docker.com/
[5] https://aws.amazon.com/cloudwatch/
[6] https://aws.amazon.com/ses/

for the Campaigner component, as well as every other email sending. Additionally AWS Simple Notification Service[7] (SNS) is used by SES for passing events to the backend over a REST Api. The last AWS service is used by the backend and frontend alike, called AWS Cognito[8], and is used for storing the users as well as authenticating and authorizing them in the frontend and backend.

Concerning the code, it has to be noted, that the original code architecture has been reused, this means, that, for example, the error handling remains mostly the same as before, using two middlewares[9]. One of these middlewares converts the errors, that are not of the custom error type, to custom internal server errors, while the other converts these custom errors into a REST error response using their respective error messages and error codes. All in all, there are only a few modifications that were made, like removing the now obsolete code, used for the account management, login and registration, and refactoring the api authentication and authorization middlewares for Cognito.
The biggest change is the refactoring of the directory structure to support the split of the backend into multiple components, which has been decided together with Nasser.

The components are integrated with each other on the database level, meaning they do not communicate with each other directly, using for example their REST Apis. The reason for that is mainly the simplicity and thus quick and easy implementation of this approach.

4.2 Frontend

This section will highlight the frontend, or rather frontends, which are implemented using the VueJs framework in combination with AWS Amplify[10].

4.2.1 Amplify

When hosting a frontend from AWS, using Amplify can have numerous advantages, like the built-in continuous deployment which detects pushes to a git repository and updates the running application. Another one is the fact, that Amplify has its own little backend, where modules for other AWS services, like the *Auth* module for connecting with AWS Cognito or the *Api* module for the AWS ApiGateway[11], can be added, simplifying the usage of those.

[7] https://docs.aws.amazon.com/sns/
[8] https://docs.aws.amazon.com/cognito/
[9] https://expressjs.com/en/guide/using-middleware.html
[10] https://aws.amazon.com/amplify/
[11] https://aws.amazon.com/api-gateway/

This backend implementation can be further extended with the usage of so called triggers provided by Amplify, like for example the *Pre Sign-up* trigger for the Auth module. These triggers will be pushed to AWS as AWS Lambda functions[12] written in JavaScript (*Lambda Triggers*, 2021).

The Amplify backend of Uni1 uses three of these triggers which will be described in the following. It is to be noted, that all of these triggers are Cognito triggers, meaning they will be triggered by certain events sent by Cognito.
The first used trigger is a predefined trigger called *Custom Message*, which will be added if the option *Email Verification Link with Redirect* is selected during the setup of Amplify. The trigger gets executed after the sign-up and is, like the name for the option suggests, for the sending of the verification code for the email, that was provided during the sign-up.
The next trigger is also a predefined one, called *Post Confirmation* and is added with the selection of the *Add User to Group* option. This trigger will be triggered after the user was confirmed, which happens after the email verification code from the previous trigger has been entered or an admin with access to Cognito, manually marks the email as verified (*Signing Up and Confirming User Accounts*, 2021).
The last one uses the *Pre Sign-up* trigger and was implemented in order to ensure that the given email is not already used. The reason why it was implemented this way, is the fact, that as of time of writing, the Amplify CLI setup did not support selecting the option *Also allow sign in with verified email address* which would mark the email as an alias which has to be unique. In addition to that, the alias also has the issue, that a new account with the same email can still be created, but the verification will fail (*Configuring User Pool Attributes*, 2021). Due to these issues, it was decided to implement the custom trigger which scans the Cognito User Pool for a user with the same email and prevents the sign up in case it finds one.

4.2.2 VueJs

While the frontends use VueJs[13] as their framework, they do not do so in the traditional way using JavaScript. Instead, it was decided, together with Nasser, to use the TypeScript variant, as well as the Class-Style Vue Components[14]. The reason for that, is the additional type safety, provided by TypeScript, which makes the development easier, as well as simply preferring a Class-Style implementation.

[12] https://aws.amazon.com/lambda/
[13] https://vuejs.org/
[14] https://vuejs.org/v2/guide/typescript.html#Class-Style-Vue-Components

For the state management it was decided to use Vuex[15]. The reason, why a state management was chosen at all, is mainly the fact, that the old implementation used Redux[16] and removing the state management would have resulted in additional workload. Vuex specifically was chosen due to its feature richness and community, as well as the TypeScript support, especially the last part which not all Vue packages have.

In comparison to the backend, it was decided, together with Nasser, to already split the frontends into separate VueJs applications and to follow the subdomain structure.
The subdomain structure means, that each frontend has its own subdomain instead of an path.

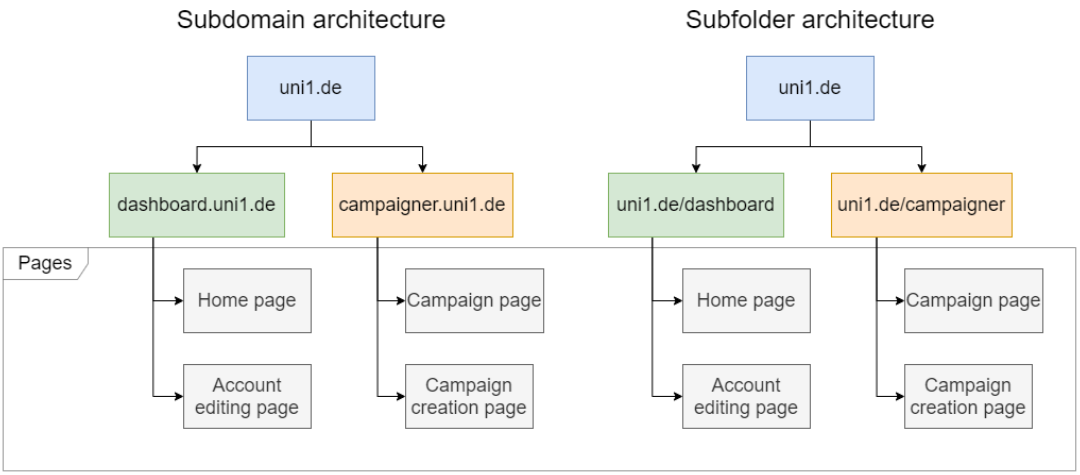

Figure 4.1: Subdomain vs subfolder using the new Uni1 application as an example

The reason why this architecture was chosen is, that thanks to this, it will not be necessary to refactor the frontend, should it be decided to switch to microservices, because they already are as separated as possible (without adding a lot of unnecessary complexity).
But this causes an issue concerning the login, because of the fact, that each application is its own separate VueJs application and thus cannot directly share their data / state. This means, that the login state, and other data, has to be shared using one of the browser storages, like the *local storage* or using cookies. Thankfully Amplify provides the option to store the authenticated user either in the cookie storage or a custom implemented one. Uni1 opted for the cookie

[15]https://vuex.vuejs.org/
[16]https://redux.js.org/

storage to share the Amplify data with all *uni1.de* subdomains, due to the easy configuration.
This means, that Amplify will save the current authenticated user in the cookie storage and will try to fetch him, before making a call to Cognito, in the case that the user gets requested. Thanks to that, if one application fetches the user from Cognito, all other application will use the user from the cookie store until the cookie becomes invalid.

4.3 Continuous integration & Continuous deployment

This section explains the software stack used for deploying the Uni1 application using Continuous Integration / Continuous deployment (CI/CD).
Additionally, this section needs to be separated between frontend and backend, due to the different methods CI/CD has been implemented for the frontend and backend.

4.3.1 Backend

The CI/CD workflow of the backend is based upon the GitLab pipeline[17], as well as AWS CloudFormation[18], which is used to manage the AWS services.

CloudFormation (CFN) is a service offered by AWS in order to help manage the AWS resources used by a project. The resources are defined inside a template[19] and CloudFormation will then create a so called CloudFormation stack which includes all of the services and resources described inside this template. This makes it easy to recreate these resources or duplicate them (using a different stack name). In addition to the easy creation, this also simplifies the deletion, because the deletion of the stack causes the deletion of all its resources (*What is AWS CloudFormation?*, 2021).

Due to the usage of CloudFormation it is not necessary to manually create, for example, the ElasticBeanstalk (EB) instance, or AWS SES, because they will be setup by the first pipeline.
Unfortunately, it is not possible to remove all manual work, some things still have

[17] https://docs.gitlab.com/ee/ci/pipelines/
[18] https://aws.amazon.com/cloudformation/
[19] https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/gettingstarted.templatebasics.html

to be done manually, like the creation of the IAM users[20] and roles[21], which the pipeline uses for CloudFormation.

4.3.2 Frontend

The reason why the frontend has to be discussed separately, is the fact that it does not use the GitLab-CI for deployment, but rather the in Amplify built-in git-based deployment[22]. This means that Amplify will detect pushes to the specified branch on the git repository and start its pipeline on its own.
After Amplify is finished, the application will be served using the in amplify defined url from S3, similar to how the old application was hosted, but using Amplify instead of CloudFront[23].

[20]https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users_create.html
[21]https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_create.html
[22]https://docs.amplify.aws/guides/hosting/git-based-deployments/q/platform/js
[23]https://aws.amazon.com/cloudfront/

5 Implementation

In this chapter the implementation of each component will be explained in further detail. It will focus on the components implemented by me, but due to the close cooperation between Nasser and myself, there are some overlaps, which will be mentioned as well.
First the refactoring of the old application into the new Marketplace component will be detailed, followed by the extracted auth component and closing with the new Campaigner component.

5.1 Marketplace component

This section discusses the porting and setup of the application using the existing NodeJs application as a starting point.

5.1.1 Backend

This part was mostly done by Nasser, but will be described, due to it being the ground work on which the other backends build upon.

Old auth removal
The first thing that was done for the backend was the removal of the old auth implementation, which was done by Nasser, but needs to be addressed for the sake of completeness.
This included the removal of the old authentication and authorization middleware, of the whole old user api which was used to register new user, delete and edit them, and much more. All of this was no longer required due to the switch to Cognito, which will be detailed later.
The next things this refactor entailed, was the refactoring of the MongoDB user model. During this refactoring all attributes were removed, that were either no longer required, or moved to Cognito, with most of them moved to Cognito. Examples are the 'firstName', 'lastName', 'sex', 'phoneNumber', 'email' and

'password' using their Cognito equivalents 'given_name', 'last_name', 'gender', 'phone_number', 'email' and of course the password. The only really removed attribute was the 'studentInfo' caused by the removal of the student user type and the fact, that this did not cause any major issues.
Newly added was the 'usersub' attribute which stores the 'sub' attribute of the Cognito user and is used to unambiguously reference one of them, thus their equivalent of an id and used in many api calls.

Directory restructuring
The next refactoring that was done, also by Nasser, was the refactoring of the directory structure to support the split of the backend into multiple components. There are now subdirectories for each component in the root of the backend, as well as a directory called 'shared' where, for example, the main database models, like the user, as well as utils, like the error utils, reside. This folder would have to be included into each component, should the backend be separated into microservices.
Here it was also decided to remove the separate 'uni1-api-clients' package, mainly because the frontend would be written in TypeScript anyway. Instead, the REST Api has been built analogous to the file structure, meaning that the main server script file loads the api from the main router files inside each component and these router files contain the api for the corresponding component.
In addition to that, each component has a similar main server script, whose only difference is, that it only loads the router file of component it is part of, making it possible to start each of them on its own. The new directory structure is visualized in the appendix *here*.

Dockerization
The last thing Nasser did on the backend, was to dockerize it, because it was decided to run the backend as a Docker application in AWS, unlike the old backend as a Node application.
This was another of the reasons, why it was decided to scrap the 'uni1-api-clients' package, due to problems getting '*npm link*' to work inside the Docker files.

Marketplace refactoring
The last point, which was supposed to be done by me, was to refactor the existing Marketplace backend to fit the changed roles and the decision to remove the user dependence on either a company or an educational unit (edu)/university. Unfortunately, there was not enough time and it was decided to focus on the Campaigner and because of that this is still missing.
This means, that the company and edu models are still present and the backend has to be refactored, removing all company and edu dependency.

5.1.2 Frontend

Style porting research
The main thing that was done for the marketplace frontend, was the porting of the look and feel of the old application from ReactJs to VueJs, which used the SCSS package of Semantic UI.
This entailed, first and foremost, researching about the feasibility of reusing Semantic UI as the styling framework with VueJs. The results of that were, that for once it would be possible to use the so called *semantic-ui-sass* package, which would allow for the implementation of Semantic UI components using SCSS. This would have three drawbacks, the first being the fact, that the package has been depreciated, as of time of writing. The second is, that it only includes the default theme of Semantic UI and the last is, like the name suggests, it only contains SCSS files, meaning the functional part of advanced elements like dropdown have to be implemented separately.

Other than the scss powered package, there is also a package called *semantic-ui-vue*, but this is still in development and would make some styling modifications difficult that were done to, for example, the navigation bar, due to it being a component package. However, being a component package in comparison to a SCSS package, it has the advantage of included functionality in the form of JavaScript.

Comparing both packages we opted for the scss package due to the old application also using a scss package, which made it easy to port. But the official package is also used whenever a component, like a dropdown, with JavaScript was needed. In order to be able to use the packages, a custom type declaration file was written that allows the plugin to work with TypeScript. One example where the package was used is for the dropdowns in the Dashboard.

After it was decided to use the scss package, all Semantic UI components that were implemented in the old application, were translated from ReactJs to VueJs, which also included the translation from JavaScript to TypeScript. The last part was caused by the decision to use VueJs with TypeScript and with its Class-Style Components.

Component porting
The next step was to identify all non Semantic UI components, that would have to be translated, in order to style the new application the same way. These components are, for example, the main header of the application, its left sided navigation bar, or the details page using the project details page as an example.

Marketplace porting
The last thing to do was the porting of the main Marketplace UI, meaning all pages and functionality. Due to the before mentioned changes made to the user roles and the removal of company and edu models, it was necessary to merge the existing pages, which were split into separate pages for company and edu users containing different functionality, into a single page for all users.
Unfortunately, again, this was not possible time wise and because of the decision to focus on the Campaigner component.

5.2 Dashboard component

In this next section, the implementation of the Dashboard component will be detailed. This component contains the logic that connects all components and, due to that, has the highest overlap between Nasser and myself. Amplify was also setup and initialized during the implementation of this component, which is caused by the fact, that Amplify also initializes Cognito and this component is the central authentication component.

5.2.1 Backend

New auth middleware
The central point of this, was the implementation of a new express middleware (StrongLoop & other expressjs.com contributors, 2017) which authenticates and authorizes the user. This middleware was implemented by Nasser and validates the signature and claim of the JWT Token sent by the frontend (*Verifying a JSON Web Token*, 2021). Additionally, this middleware also appends the database user entry, of the user which sent the request, to the express request object, like the middleware of the old application did. This makes it unnecessary to afterwards readout the user again from the token should he be needed during the handling of the request and was an addition to the middleware made by me.
The last thing this middleware does, is to verify, that the requester is authorized to execute the given api. For this the middleware gets an array of authorized roles, which are authorized to execute the api and compares them to the groups the user belongs to, according to Cognito. Should the user belong to at least one of these roles/groups, the middleware will pass the request to the actual endpoint.

Registration endpoint
Next was the endpoint for registering users, which was necessary because of the fact, that not everything could be saved in Cognito without any issues, one such attribute are the contacts, that are assigned to a user.
Theoretically Cognito allows the creation and saving of custom attributes, but this unfortunately would make it a little bit harder to retrieve connected data, for

example all users together with their contacts. In addition to that, these custom attributes can neither be required (*Configuring User Pool Attributes*, 2021) nor can the users be queried with them (*Managing and Searching for User Accounts*, 2021), but each user has to have at least one contact.
This api will create a user in the database after some checks concerning the contacts saved in the database. Should one of these checks fail, the user will be deleted in Cognito (and the database if the api fail afterwards). These checks are necessary because the fact, that contacts can exist without any users and will now be further detailed.
First the api will check, whether there already exists a contact which uses the same email as the new user would use for his account. Should this not be the case, the api will simply create a new contact before creating the new user, and assign this contact to the new user. Otherwise, the api will check if this contact is already assigned to a user, in which case the api will throw an error, thus deleting the user in Cognito and return the error. But should the contact be unassigned, the only thing the api will do is to update the contact, if the user has entered a phone number during registration and the contact has none saved, because the fact, that this is an optional attribute. After this the user will be created in the database and the registration, in the backend, was a success. A more detailed flow of the whole registration/signup api can be found in the activity diagram in the appendix *here*.

Account management endpoint
The last important thing to do for this component in the backend, was the creation of an api, that would allow the user to edit his account attributes, like his name, email, title and so on. In this case, unlike for the signup, everything is done via the backend api, which is responsible for updating the user in Cognito. After some assertion that the given data is valid, like that the user even exists or that at least the required attributes for a user are given, the api will assert again, that the email is not used. For this it will first query Cognito for a user with the given email and if one is found, assert that the username of this user does not equal the user who is editing himself (in case there was no update of the email). Unfortunately, this has to be done like that, because Cognito does not support queries with more than one condition (*Managing and Searching for User Accounts*, 2021). Should the usernames not match, an error will be thrown and the editing will be rejected, because the email is already used by another user. In case no user was found, it will be checked, whether any contact uses this email and if that is not the case, the assert function will return a value indicating, that a new contact has to be created. But if a contact was found, the api will check whether this contact is assigned to any user and if it is, either throw an error in case the user is a different one that the user that initialized the request, or return stating that nothing needs to be done. If the contact is unassigned the function

will return saying that there is an assignment needed.
After this assertion, either a new contact will be created, or the existing is assigned to the user followed by the actual update of first the user in the database and then of the Cognito user. Should the email be updated, Cognito will also send a new verification email to this new email, but this will be discussed further in the frontend section of this component.

Additional endpoints
Other than the points listed above, the Dashboard backend also contains some fundamental apis and utils, which are not that important. One example of a such an api is the one, which serves all tags to the frontend and is used by the account management page for the tag dropdown.

5.2.2 Frontend

Amplify
The main point here, was the setup of Amplify, which includes Cognito, using the Amplify CLI[1]. Cognito can also be setup separately and then imported into Amplify, but because both had to be setup at that time, this was the easier choice. Only the most important settings will be highlighted in the following, because, for example, the name of the Cognito User Pool, which is more or less the database that saves the user, is not really important.
The first important option is during the initialization of Cognito, and is the selection about how users are able to sign in. Here 'Username' was chosen, which restricts user, such that they are only able to sign in using their username, not their email. The reason why username was chosen, is that a user might change their email, if, for example, they switch university or company, and thus have to remember the new log in. The technical reason for that was, that allowing users to sign in with their email, would mark the email attribute in Cognito as a so called 'alias' and requiring the email to be somewhat unique. The reason why it is somewhat unique instead of unique is, that this type of uniqueness is not checked during the registration, but rather during the email verification. This means that Cognito would still allow a new user to be created with an already used email, but fail in the second step during account validation in which case the user would be forced to abandon the new account creation and start anew (*Configuring User Pool Attributes*, 2021).
After this came the creation of the groups for the Cognito users. Here the three groups/roles were created for the different applications. The 'User' for users with access to the Marketplace component, the 'Campaigner' group for access to the Campaigner and 'Administrator' for the Admin console.
The next important option is concerning the user attributes for Cognito, more

[1] https://docs.amplify.aws/cli

precise, the attributes that are stored in Cognito for each user and this app can access. Here all attributes were selected that were removed from the old application and moved to Cognito, meaning 'Email', 'Family Name', 'Given Name', 'Gender' and 'Phone Number', in addition to the meta data 'Email Verified' and 'Phone Number Verified' which indicate whether the email or the phone number has been verified using one of the available verification methods. The latter option was selected in case of future expansion with, for example, two-step authentication using SMS, and because of its low additional effort.
This is also part of the next important option, where the capabilities were selected, that are wanted to be enabled. Here 'Email Verification Link with Redirect' was chosen for the email verification and 'Add User to Group' in order to add a user to a default group after sign up.
The last important options are concerning the Lambda Triggers for Cognito. First it has to be selected that it is wished to configure them and then the 'Pre Sign-up' was enabled in addition to the already enabled 'Custom Message' and 'Post Confirmation'. 'Custom Message' is used for the 'Email Verification Link with Redirect' option and 'Post Confirmation' for 'Add User to Group'. 'Pre Sign-up' will be a custom written NodeJs function for AWS Lambda and detailed in the following.

Pre Sign-up Lambda
Next the custom NodeJs Lambda was implemented, which is used to verify, that the email inputted during the registration is not used by any other Cognito user. The reason why this lambda was used for this, is that the only way to mark the email as unique in Cognito, is to mark it as a so called alias. An alias can be used as a substitute for the username during the log in, but unfortunately, aside from the previously mentioned technical issues, the Amplify CLI does not support the creation of alias up to now ("How do you want users to be able to sign in when using your Cognito User Pool? Username + email + phone", n.d.).

Now the main Amplify backend has been setup and is ready to be used by this Amplify application. In order to use the same Amplify backend with other applications, the other applications simply have to follow the official Amplify guide for applications with multiple frontends, that uses '*amplify pull*' to 'import' the backend (*Multiple frontends*, 2020). This is necessary, because otherwise each application would have their own lambdas and, in worst case, their own Cognito instance which is used for the log in.

After this it was time to finally implement the frontend itself using the ported Semantic UI components mentioned *here*.

Auth UI

The first and most important thing, was the implementation of the log in UI, as well as sign up and everything related.
There is an official VueJs package for Amplify which contains UI components for this use case, in fact there are two, a legacy and a latest version (*Amplify UI Components*, 2020), but unfortunately it turned out to be very difficult, up to impossible, to customize them. The customization was needed in to different cases, the first was to style the log in UI in the same style as the rest of the application. This is in fact possible, because the newer of these packages offers a way to customize the CSS styles. The other reason was, that the sign up needed some custom logic after the Amplify sign up succeeded (backend sign up as discussed in previous section), which, by itself, is also possible with the newer package, but there was also the requirement of certain backend only attributes, like the title, to be entered on sign up and the Amplify api uses a certain model that is passed to it. This together with the CSS customization made it easier to just create own Vue components inspired by the legacy components found on GitHub ("amplify-js/packages/aws-amplify-vue/", 2020) and the log in components of the old ReactJs application. The reason why the legacy components were used as inspiration instead of the newer ones, is the fact, that the newer package only consists of a statement, that is transcompiling the new ReactJs components ("amplify-js/packages/amplify-ui-vue/", 2020).
It has to be mentioned that not all components were implemented, that exist in the Amplify-Vue package, for the simple reason, that not all are needed for this application. The only components implemented are the 'SignIn', 'SignUp', 'ConfirmSignUp', 'ForgotPassword' and a new one which is not a part of the legacy package but only of the latest, the 'VerifyEmail' ('VerifyContact' in official Amplify package).

Homepage

Next was the homepage which contains the links to the other applications. This one mainly consists of a dynamically generated list of buttons which are described inside the config located inside 'src/config' and each adhering to the following json schema[2].

```
1 {
2   "$schema": "https://json-schema.org/draft/2020-12/
      schema",
3   "$id": "/config/applicationButton/schema.json",
4   "title": "Application Button",
5   "description": "An application button for the
      Dashboard homepage",
```

[2]https://json-schema.org/

```
6   "type": "object",
7   "properties": {
8     "name": {
9       "description": "Name of the application",
10      "type": "string"
11    },
12    "url": {
13      "description": "Url of the homepage, e.g. 'https:
          //campaigner.uni1.de'",
14      "type": "string"
15    },
16    "description": {
17      "description": "A short description about the
          application",
18      "type": "string"
19    },
20    "icon": {
21      "description": "The Semantic UI icon used,
          excluding 'icon' suffix and color, size
          variation, e.g. 'clipboard list'",
22      "type": "string"
23    },
24    "groups": {
25      "description": "Array of user roles, that are
          allowed to see/access the component for this
          button",
26      "type": "array",
27      "items": {
28        "type": "string"
29      },
30      "minItems": 1,
31      "uniqueItems": true
32    }
33  },
34  "required": ["name", "url", "icon", "groups"]
35 }
```

This makes it very easy to add new components to the Dashboard, because it does not require any direct HTML or TypeScript/JavaScript changes.
In addition to that, this page currently also contains a button, implemented by Nasser, called 'Entwicklerwerkzeuge (Dev Tools)' enabling an admin to create a new tag. This is temporary and only due to the time-related inability to imple-

ment the whole Marketplace frontend, as already mentioned, where new tags are to be created in the future on the fly during project creation.

Account management page
The last thing to do for this component, was the account management. This page was more or less ported from the old application with minor adjustments. One adjustment was the addition of the tags input where the user can select tags, he wants to subscribe to, using a dropdown with included search bar. Another change was a new tab for contacts, where the user can manage all contacts assigned to himself. Here it is possible to create new contacts, delete them or disable them for campaign emails.

5.3 Campaigner component

This section is about the Campaigner component, the new component that was implemented as a part of this thesis.

5.3.1 Backend

Email provider
The first thing was the implementation of the email provider, which was the main part of this component and also where most problems occurred. The main problem was, that the first chosen email provider lacked the requirement to receive Out-Of-Office (OOO) replies and only supported the receiving of manual/normal responses. Unfortunately, this was not completely clear at the time, due to the lack of knowledge that there is a fundamental technical difference between these types of responses.
After this it was decided to switch to AWS Simple Email Service (SES), mainly because of the fact, that it can at least detect OOOs according to its bounce documentation for the bounce type 'Transient' and its sub type 'General' (*Amazon SNS notification contents for Amazon SES*, 2021). In addition to that, the other reason why SES was chosen is simply because the application already builds heavily upon AWS and this way it is at least centralized.
Additionally, to SES there is another AWS service needed, if the application is to be notified about anything concerning SES, like bounces, replies, complaints and more. This additional service is called Simple Notification Service (SNS) and is a basic subscription service, meaning that services can send a message to a certain topic in SNS and it will send this message to all clients subscribed to this topic. SNS is highly integrated into SES and thus the setup was fairly simply due to the extensive documentation and the usage of CloudFormation in this project.
After starting with SES it became clear, that there is a problem caused by the fact, that the 'Email receiving' feature would also be needed, but this feature is

only supported in three regions, with Ireland (eu-west-1) being the only European region among them (*Amazon Simple Email Service endpoints and quotas*, 2021). Due to this it was decided to split the application into two parts, closer detailed in *this section*, but luckily SNS supports cross-region subscriptions and they are very easy to setup using CloudFormation (*AWS::SNS::Subscription*, 2021).
At this point the main structural problems were solved and it was time to setup SES and implement the logic to send and receive emails in the backend.

SES basic setup
First SES has to be setup in AWS, but unfortunately only a small fraction of this is possible with CloudFormation, due to the fact, that SES requires setup with the domain that is to be used[3], and none of these things that are possible with CloudFormation alone, is strictly needed for email sending[4].
First the domain had to be registered, which took some time due to some issues with the configuration of the domain, which prevented AWS from accessing it (*Setting up a custom MAIL FROM domain*, 2021). This got resolved after the transfer of the domain to Ionos, which was done by Professor Riehle.
Next SES was moved out of sandbox in order to increase the sending limit and to be able to send emails to non-verified recipients (*Moving out of the Amazon SES sandbox*, 2021).
Lastly DKIM was setup, in order to authenticate the sent emails, because otherwise almost all emails would be caught in most SPAM filters (*Setting Up Easy DKIM for a Domain*, 2021).
Now it is finally time to implement the emails sending and receiving using SES.

SES sending
The sending part was implemented first because of the fact, that this is a lot easier to implement thanks to the official AWS SDK provided for NodeJs[5].
The other requirement connected to the email sending, the implementation of an email template, also does not pose any problem thanks to SES providing the possibility to upload and use email templates. The email template used is a very simple template loosely based upon an old GitLab notification email for its layout, because the text itself is one injectable parameter. This email also contains an unsubscribe link in the footer that the user can use to disable campaign emails for the corresponding contact[6]. This email template is created in SES using CloudFormation[7].

[3] In the case of this application *uni1.de* registered with Ionos
[4] Only for email receiving, logs, or email templates
[5] https://aws.amazon.com/sdk-for-javascript/
[6] **Note:** this has been temporarily removed few weeks before submission date for real life testing of application
[7] CloudFormation chapter

The sending itself was implemented in a utils file and supports the sending of bulk emails with the created template and separate template data for each of the recipients.
After creating an email sending job, the api will return a json containing an array with the recipient and a messageId for the sent email. These message ids will be saved in the campaign, in order to be able to draw a connection between a bounce or OOO reply and a campaign.

SES receiving
The receiving was a little bit more to setup because of the fact, that this required the setup of SNS subscription handling.
This included the setup of SNS itself within AWS, consisting of the SNS topic, in the AWS region of SES, to which SES publishes the new messages, and a SNS subscription for the backend, in the AWS region of the backend. Because of the usage of CloudFormation, this was not done manually, but rather using the CloudFormation template, further detailed *here*.
After a SNS subscription is created, it has to be confirmed using the confirmation message which is sent to the specified endpoint afterwards. Luckily this also is fairly easy thanks to the AWS-SDK, which provides a function that has to be called with the TopicArn, the unique AWS identifier for that specific topic, and the token received in the confirmation message (*ConfirmSubscription*, 2021). Of course this means, that the endpoint that handles the OOO replies, also has to be able to confirm the subscription first.
In addition to the SNS setup in AWS, SES also needs some adjustments for email receiving as well as bounce handling.
For the bounce handling a so called 'Configuration Set' has to be created. Unfortunately this is only partly possible using CloudFormation, because while the creation itself is support, it is not supported to select a SNS topic as its destination (*AWS::SES::ConfigurationSet*, 2021) and thus this has to be done manually after the first pipeline has finished and the AWS resources have been created (*Set up an Amazon SNS event destination for event publishing*, 2021). This configuration set, then has to be passed to each api call of the AWS-SDK that sends an email like this *Step 3: Specify your configuration set when you send email* (2021).
Next the pass through of received emails was implemented using a 'Receipt Rule Set'. This can be setup completely with CloudFormation and only requires activation because only one of these rule sets can be active at a time (*Creating a receipt rule set for Amazon SES email receiving*, 2021). This rule set was configured, such that it sends all received emails to the same, previously created, SNS topic as the bounces, in order to minimize the number of AWS resources.
Finally, all AWS services are setup, ready to go and emails can be received.

Out-Of-Office handling

After the emails receiving was setup it was time to implement the handling of the SES messages, which was split into the handling of bounces and of the replies itself, because they are received using the same endpoint.
This means, that the first thing the endpoint does with SES messages, is to check whether the message is a bounce or an OOO reply and if the message is none of the two, it will return a 400 error with message 'Unknown event-/notificationType'.
In the case of a bounce, the system will first try to retrieve the campaign that sent the email which caused the bounce using the stored messageId. If no campaign is found, the endpoint returns a 'ResourceNotFound' error, but if a campaign was found, it will extract the necessary data from the message and create a new bounce document in the database, containing the data and referencing the campaign.
In the other case, namely in the case of an incoming email, the endpoint will first check the virus and spam header and abort if one of these checks done by SES failed. If the two headers are fine, the endpoint will continue with the check, whether the message contains an auto-submitted email, meaning an OOO reply, or a normal email, because normal emails will be ignored.
After this it is finally time for the data extraction, which depends on the multiple factors like the type of the 'auto-submitted' header and currently implemented/-tested are OOO replies from Gmx, Gmail and Outlook. The reason for that is, the fact that each provider implements their OOO reply differently, which will be highlighted using Gmail and Outlook (Gmail and Gmx are fairly similar to each other).
The first big difference is the type of the auto-submitted header[8] for which Gmail uses 'auto-replied' while Outlook uses 'auto-generated'. This is relevant, because from my observation with Gmx and Gmail, which use the same header type, OOO with the 'auto-replied' header also trigger bounces and the bounce then sends the OOO. This results into the issue, that the sender of Gmail OOO replies is an AWS internal mailer demon, in my case 'MAILER-DAEMON@eu-west-1.amazonses.com' and not the real sender. Because of this, the sender of the OOO reply has to be extracted from the corresponding bounce, while in the case of 'auto-generated' there is no bounce and the sender can be extracted directly.
The second difference is the 'Content-type' header of the body. Gmail for example only sends the HTML which is how the application stores them in the database. Gmx meanwhile sends a simple text with the content type 'text/plain' and because of that is then converted into a very simple HTML with the whole text content wrapped inside a 'div' tag. Outlook on the other hand, does it even different and sends both of these types inside a multipart body. This multipart body then has to be split into its parts and the HTML part is extracted and used.

[8]https://www.iana.org/assignments/auto-submitted-keywords/auto-submitted-keywords.xhtml

These two differences make it very difficult to guarantee a complete support of all OOO replies.
The last things left for the endpoint to do, are again uniform for all different replies and simply entail the extraction of emails from the body, ignoring duplicates caused by clickable mailto links, the creation of a new reply document for the database, containing the data, and the updating of the bounce, if existing, with the new reply id in order to create a relation between one bounce and one reply of the same campaign.
A detailed activity diagram displaying the above mentioned OOO handling can be found *here* in the appendix.

5.3.2 Frontend

This frontend uses the same ported Semantic UI components as the Dashboard component as well as the same Amplify backend, which will be further highlighted in the first subsection.

Amplify

The first thing that had to be done after the initial VueJs setup, was the Amplify setup. Fortunately Amplify supports to have multiple frontends for the same Amplify backend, which is mainly used in the case, that the application has a web frontend and a separate app for Android and/or IOS.
In general, this is very easy to setup and only requires initializing Amplify for the project using *'amplify init'*, to generate the basic *aws-exports.js*, and afterwards running *'amplify pull'* to pull the backend of another application (*Multiple frontends*, 2020).
The only things that have to be setup during *'amplify pull'* is to select the type of this application, in case of the before mentioned use case of web app and Android/IOS app, but in this case is also VueJs. Additionally, it will be asked, whether the Amplify backend will be modified, which can be answered with no, because all modifications will be done using the Dashboard Amplify app, where it was originally setup.
This was everything needed for the setup of the frontend, aside from the CI/CD part detailed in the last section of this chapter.

Campaigns list page

After this it was time to start with the frontend itself and the first page is the campaign list page, being the homepage for the component, which lists all campaigns. It also has some simple filters for filtering the shown campaigns to, for example, only show campaigns created by oneself, whose title or description contain certain words, by the tags that were used for recipient selection or to only display ongoing campaigns.

The details page of a campaign can be opened by selecting one entry of this list and is explained further down this section.
Lastly this page also contains the button which opens the campaign creation page detailed next.

Campaign creation page
This page is used for creating a new campaign and consists of two steps. The first step is to enter all data about this new campaign, like the title and a short description, both are only used as meta data and have no influence on the emails being sent. There are two inputs for the email, containing the subject of the email and its body, which is inputted inside a textarea and thus supports newlines. The last input is a dropdown with a search bar where the tags are to be selected, which determine who will receive the campaign email. An empty tag dropdown will cause the email to be sent to all contacts, that have the email receiving enabled. After all required fields have been filled, it is possible to advance to step two, otherwise the fields will show an error. In the next step the preview of the email is shown inside a box with the subject above it. Here it has to be noted, that this might look different to how the email will really look, due to differences concerning how certain email provider and email applications interpret and support HTML emails. For example, many desktop email clients have problems with the 'border-radius' CSS property see here for a list "CSS Guide/Email Clients/border-radius" (n.d.). If the preview looks fine, the user can start the campaign using the corresponding button, but if some changes are wanted it is possible to go back to the input using the cancel button.

Campaign details page
Next is the before mentioned details page for the campaigns and it contains the most important information and also enables the user to close this campaign, if he deems it finished. It has to be noted, that finished campaigns can still be edited, this status is mainly for the filtering at the moment.
This page is split into three tabs with the first one containing the mentioned information about the title and description that were entered during creation, the status of the campaign, the creator, subject and tags, which were also inputted by the creator. This tab also contains a small statistics element at the top, which displays the number of bounces and OOO replies, that were received for this campaign.
If these numbers are higher than zero, the bounces and replies can be seen in the second and third tabs, which otherwise display an empty table.
Both of these tables contain the most important information about the corresponding bounce or reply and open the details page if a row was selected. The only noteworthy thing is, that the bounce table also contain a tooltip which can be open by hovering over the info icon in a cell of the bounce type column. This

tooltip contains the short description about the bounce type. The details pages itself are further detailed next, beginning with the bounce details page.

Bounce details page
The bounce details page contains the most important information about the corresponding bounce, similar to the campaign details page. Here the receiver of the email that caused the bounce is listed, as well as the bounce type, defined by SES, and a short description of this type of bounce, that is also displayed in the tooltip of the before mentioned bounces list.
Additionally, there is another entry which is only displayed if this bounce was caused by an OOO reply and this entry contains a button that links to the reply details page of the related reply.
Lastly this page also contains two action buttons. The first of these buttons causes the contact, that caused this bounce, to be deleted, but is only enabled, if this contact is not a main contact, which means its email is used as the account email of a user. The second button causes the bounce flag, that was set due to this bounce, to be removed again. This essentially means, that the bounce will be ignored and is mainly intended for temporary bounces like 'Mailbox full'.

Reply details page
The reply details page has the same action buttons for deleting the contact or ignoring the bounce flag (if there is a bounce for this OOO). It also has a fairly similar layout with the information displayed at the top, this time containing the sender of the OOO, the receiver, which is the campaigner email, as well as the subject and content of the reply.
The major difference is the second part, where a list of email addresses is displayed, that were scrapped from the body of the OOO reply. Of course, this list is empty if none were found, but if at least one was found, the list contains this email address together with an indicator, whether this email is used by a contact stored in the application. If this indicator indicates, that the email address is not yet used, the list entry will also contain a button, with which the email address can be added, in the form of a new contact, using the contact of the OOO sender as a basis for the tags and other data. Additionally, there is also a button next to the section headline of the list, that makes it possible to add all scrapped email address at once. This button is only displayed if at least one entry is in the list.

Unsubscribe page
The last page to be mentioned is the public unsubscribe page, that the unsubscribe link in the email opens. This page only calls the unsubscribe endpoint for the corresponding contact in the backend and displays the result, meaning either something like 'Success' or 'Error'.

Dashboard link
Lastly the newly written frontend has to be made accessible from the Dashboard, which is fairly simple and described in detail *here*. For this component the following entry was added to the Dashboard config:

```
{
  "name": "Campaigner",
  "url": "https://campaigner.uni1.de",
  "description": "Applikation für Marketing",
  "icon": "envelope",
  "groups": ["Campaigner"]
}
```

5.4 Admin console component

This component was mainly implemented by Nasser and the section only mentions some modifications and refactoring done by myself, but generally it is not a part of this thesis and can be looked up here Nasser (2021).
The main thing done by me, was in the frontend the main structure with the ported Semantic UI components, as well as the authentication which is using the Dashboard component as login.
The only thing done in the backend by me, were some bugfixes related to the contact api, the first being an assertion, that the main contact of a user cannot be deleted. The other thing was adding the missing unassignment to the delete contact api because otherwise a user would reference a nonexistent contact which would crash certain apis, like the account management which would fail during the changing of the account email.

5.5 Continuous integration & Continuous deployment (CI/CD)

In this last section, the implementation of the continuous integration and continuous deployment will be detailed. It is split into the three main subsections containing the different technologies used to achieve CI/CD.

5.5.1 GitLab-CI

The first subsection is about the main technology, used for continuous integration, the GitLab-CI which is used as the main pipeline for the backend.

The pipeline uses four different Docker images for the different stages, due to different requirements for each stage. These images are listed in the following.

- **node:latest**[9]:
  This is the standard NodeJs Docker image and it is used for all jobs during the 'test' stage.
- **banst/awscli:latest**[10]:
  The next most used image is the most commonly used one for the AWS CLI and is used in the application for the upload to S3 and for the creation of a new ElasticBeanstalk application version.
- **meteogroup/cfn-create-or-update:latest**[11]:
  This one is not a big one and is only a wrapper for the *cfn-create-or-update* npm package from widdix[12]. This image is used for the two CloudFormation stages in order to make the deploy process easier due to the missing native create or update feature of AWS CloudFormation.
- **coxauto/aws-ebcli**[13]:
  This last one is used for the deployment stage of the backend, due to the need for the AWS EB CLI, which the first image did not have.
  Note: This image also contains the AWS CLI and thus could replace the first image, unfortunately this was not noticed by me until the writing of this thesis.

All necessary variables have been stored inside the variables section of the *gitlab-ci.yml*, except for the variables that are required for the AWS authentication in the AWS CLI. These variables, namely the *AWS_ACCESS_KEY_ID* and the *AWS_SECRET_ACCESS_KEY* were moved by Nasser, along with the *DATABASE_URL*, into the variables inside the repository settings in order to avoid them from being stored in the code in plain text.

Now for the general workflow of the pipeline.
The pipeline was configured to first test each backend component separately, if there were any code changes in the component. After this comes the upload stage, which contains two separate jobs with the first one being the more important one. This first one uploads the backend to S3, because ElasticBeanstalk requires the code to be either pushed to S3 or CodeCommit ("create-application-version", 2021). The second is only executed, if the email template has been changed,

[9] https://hub.docker.com/_/node
[10] https://hub.docker.com/r/banst/awscli
[11] https://hub.docker.com/r/meteogroup/cfn-create-or-update
[12] https://github.com/widdix/cfn-create-or-update
[13] https://hub.docker.com/r/coxauto/aws-ebcli

causing the template to be uploaded to S3, which will be further detailed in the next subsection about CloudFormation. Next are the two CloudFormation stages, the first one creating or updating SES and SNS topic followed by the ElasticBeanstalk and the SNS subscription. The reason for the separation into two stages, was the SNS subscription which needs the SNS topic that will be created it the first CloudFormation template and as such have to run sequential. The last two stages are the deployment stages for the backend and only run as long as the CloudFormation job for the backend did not, because in the case of the first deployment this would deploy the backend a second time. The first of these two stages builds the backend by creating a new application version of ElasticBeanstalk using the zipped backend upload to S3 earlier. The second will then deploy the new version using *eb deploy*.

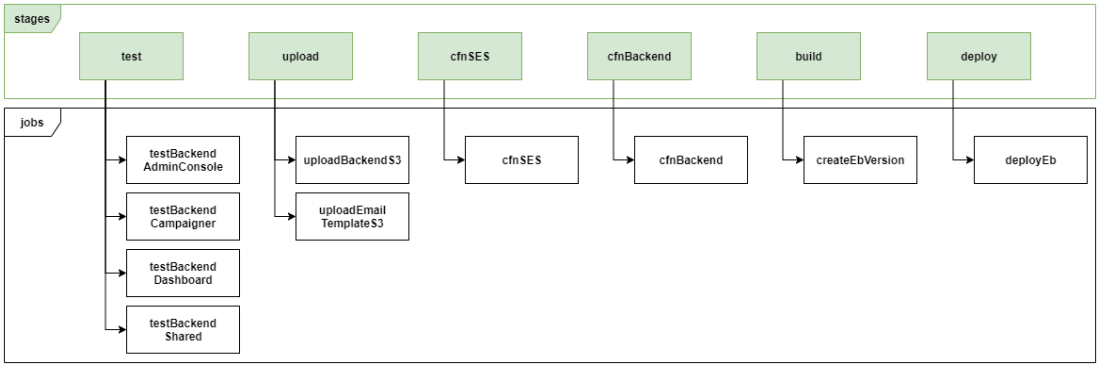

Figure 5.1: All stages of the GitLab pipeline

5.5.2 CloudFormation

Now the implementation of the CloudFormation in Uni1 will be detailed.
As mentioned in the previous chapter the reason for CloudFormation is to reduce the need for manual creation of AWS resources. For this application, CloudFormation was split into two separate CloudFormation templates and the reason for that is the issue, that the ElasticBeanstalk instance runs in the AWS region *eu-central-1*, which corresponds to Frankfurt, but SES is only partial available in that region. The important part for this chapter is the fact, that CloudFormation does not support SES in Frankfurt at the time of writing, which, unfortunately, is not well documented and only clearly visible when using the template designer from AWS (or when CloudFormation fails).

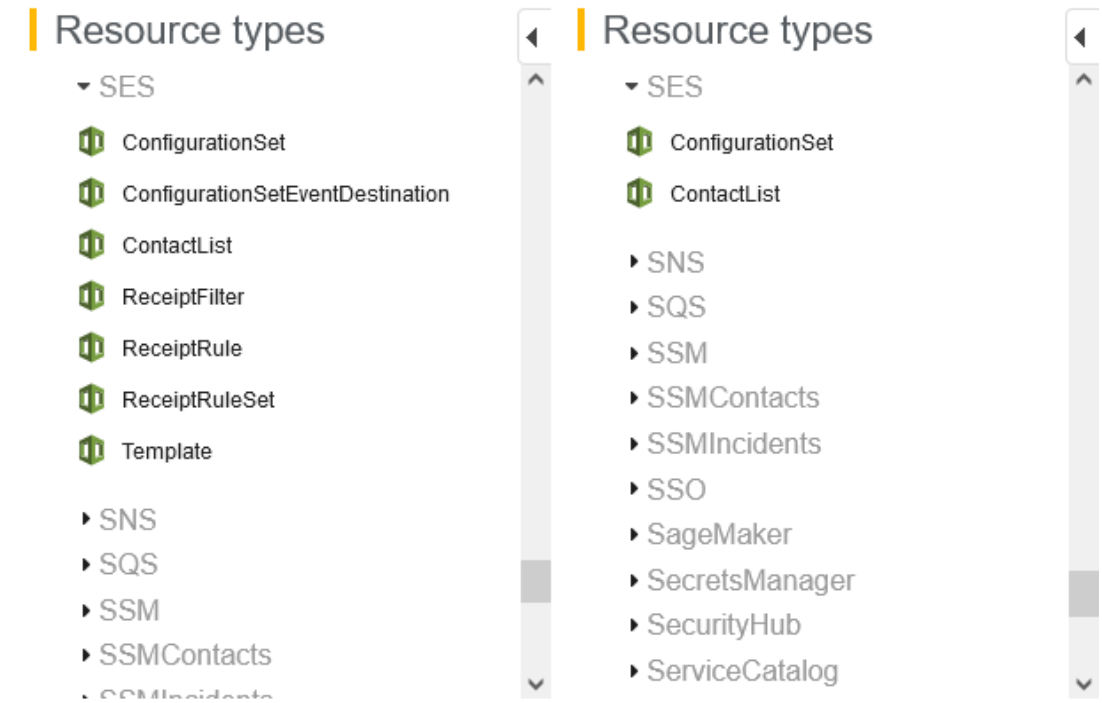

Figure 5.2: Available CloudFormation resources for SES in *eu-west-1* (left) and *eu-central-1* (right) according to template designer

Because each CloudFormation template is for one region, there are two for this project, one for *eu-west-1* which creates everything for SES and SNS, except for the SNS subscription. This includes the email template, the Configuration Set and the Recipient Rule, together with its set, for SES, as well as the SNS topic for the connection to the backend. The only noteworthy thing here is, that in order to create an email template with CloudFormation it is necessary to previously upload this email template to S3 and reference the S3 url in the CloudFormation template. The only way to get around that, is to use a third-party preprocessor like *cfn-include*[14]. These make it possible to split CFN templates into separate files and merge them before usage, but in this case the max size limit of CFN templates has to be considered. If a template succeeds 51200 bytes, it has to be uploaded to S3 when using the CLI commands *CreateStack*, *UpdateStack* or *ValidateTemplate* (*AWS CloudFormation endpoints and quotas*, 2021). At the moment this limit would not be exceeded, but it was deemed easier to just upload the email template separately because it makes up the majority of the full size. The second template is for *eu-central-1* and creates the ElasticBeanstalk instance with its basic configuration, including HTTPS which was added by Nasser, as well as the SNS subscription for the SNS topic created in the first. As previously

[14]https://www.npmjs.com/package/cfn-include

mentioned, the SNS subscription is the reason why these two templates have to be separate in the pipeline as well and the reason for that is the fact, that the subscription has to be created in the region that hosts the target of this subscription.

5.5.3 Amplify

This subsection is about how Amplify was used for the continuous deployment of the frontends. There is not much to say, because it was decided to use Amplify's 'Git-base deployment', mainly because of the easy setup thanks to the documentation from AWS (*Git-based deployments*, 2020).
The only reason why additional modifications were necessary is because of the fact, that this application consists of multiple frontends inside a monorepro, but luckily there is also a documentation for that. In case of monorepro project, a repository with multiple amplify apps, it is necessary to specify the code location of each amplify app. For Uni1 it was decided to use an *amplify.yml* file in the project root, which specifies each application together with the repository path to its corresponding code and all other build settings Amplify needs and *Configuring build settings* (2021) can be used for this. In this file, it is also possible to change what script Amplify uses for deployment and this was the main reason why this solution was used. The reason why it was necessary to change the script is a bug inside the Amplify Auth module that causes the pipeline to crash due to missing environment variables and can be fixed by adjusting the default script as described in this **GitHub issue**.

6 Demonstration

This section contains four short demonstrations displaying different use cases of the application, located in the section for the corresponding component.

6.1 Dashboard

In this first section two demonstrations are shown, concerning the frontend of the Dashboard component.
Both of these demos start on the homepage of the Dashboard at https://dashboard.uni1.de, which the user will be redirected to, after a successful login.

6.1.1 Application navigation

The first demonstration is displays how to navigate between the different applications, like the Campaigner or Marketplace.

On the homepage of the Dashboard the user is greeted with two segments below the header. The first segment contains the, for now, static placeholder and, in case the user is an administrator, it also contains a temporary button labeled 'Entwicklerwerkzeuge (Development Tools)'. As mentioned *here* this button is currently only a temporary solution for creating new tags.
The buttons, that link to each application, are located in the second segment and vary depending on the permissions of the user. Below are two examples, the first one showing the Dashboard how it looks for a user with all possible roles, and therefore permission, and the second for a user with the least permission, meaning only access to the Marketplace.

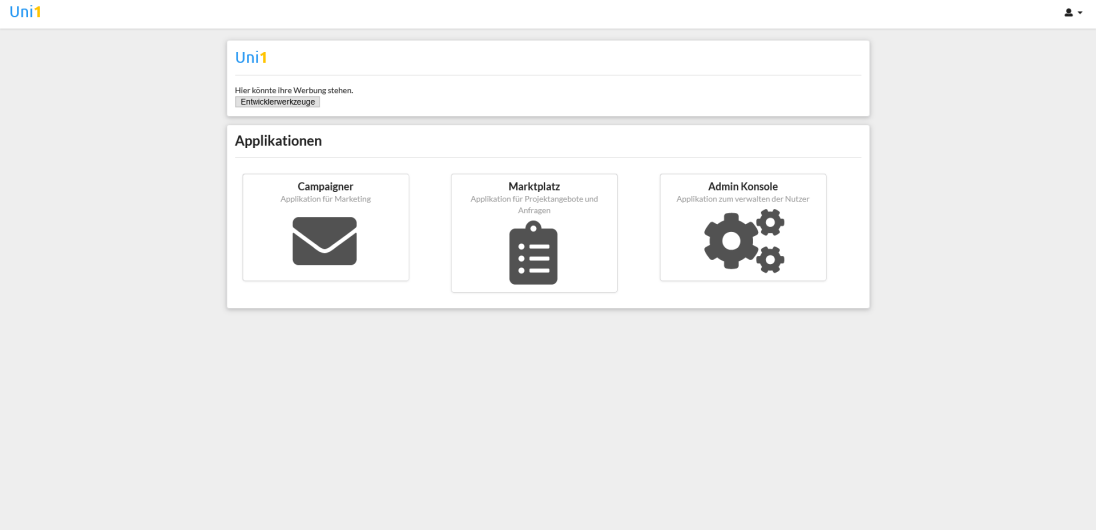

Figure 6.1: Dashboard home for admins

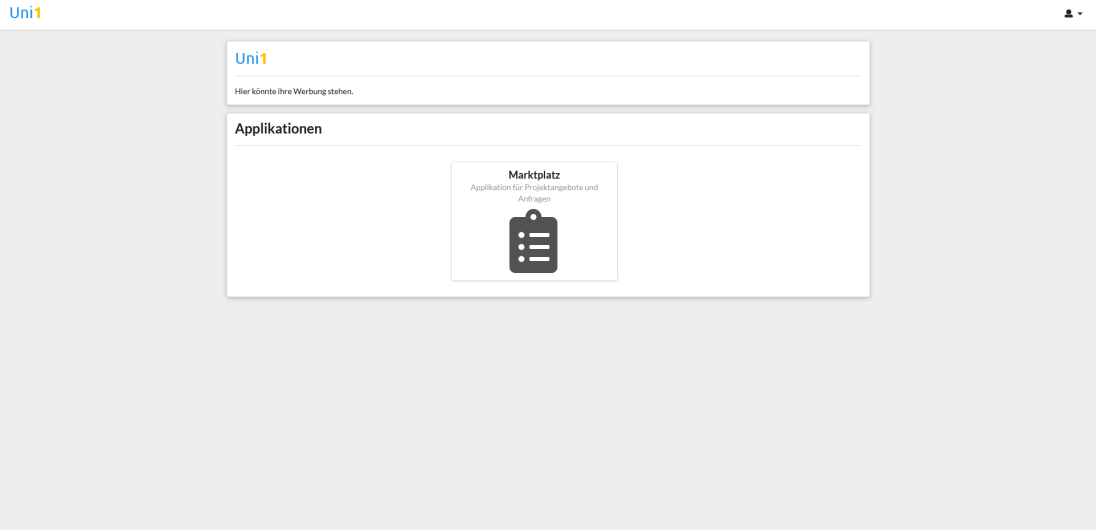

Figure 6.2: Dashboard home for normal users

In order to navigate to another application of Uni1, like the Campaigner, it is possible to either use the url, with the subdomain corresponding to the application, or the before mentioned button. In this example the button will be used to open the Campaigner, which is accessible using the url https://campaigner.uni1.de.

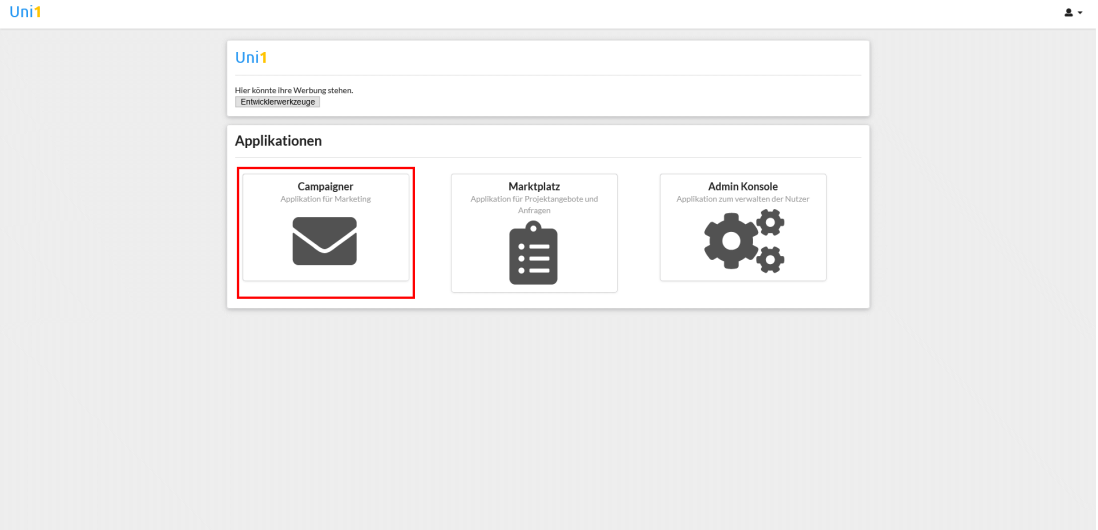

Figure 6.3: Opening of Campaigner using the button

Afterwards the user will see the homepage of the Campaigner.
From here the easiest way to return to the Dashboard is to use the Dashboard button in the upper left, as shown below.

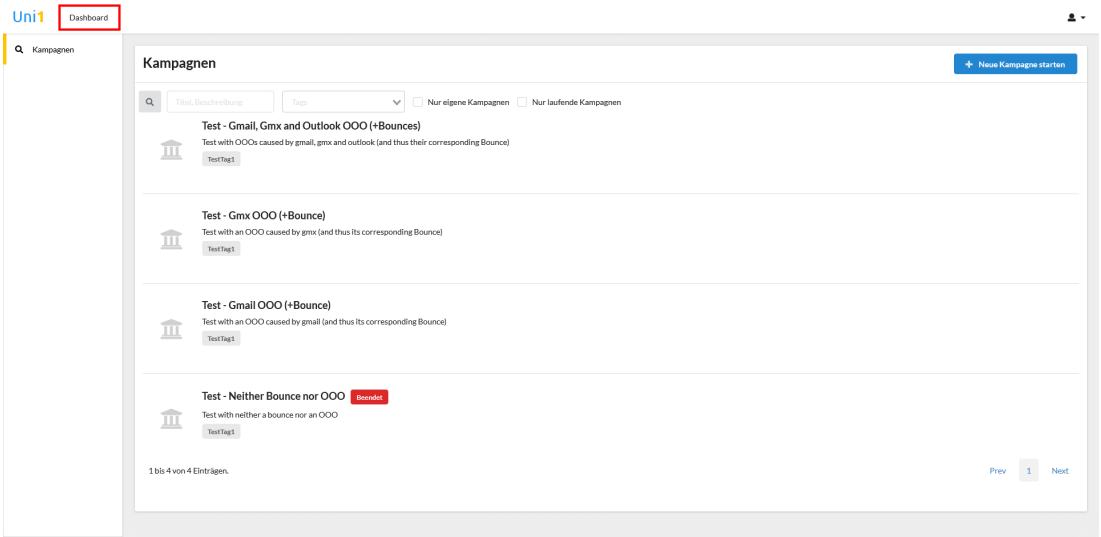

Figure 6.4: Campaigner homepage with marked dashboard button

After returning to the Dashboard, it is possible to open another application the same way.

6.1.2 Account management

The next demonstration revolves around the new account management, how to edit values of the own user and manage the contacts assigned to oneself for the Campaigner.

The Account management can be reached from every component in the same way, by opening the user dropdown in the upper right of the header and selecting the 'Konto Einstellungen (Account Settings)' option.

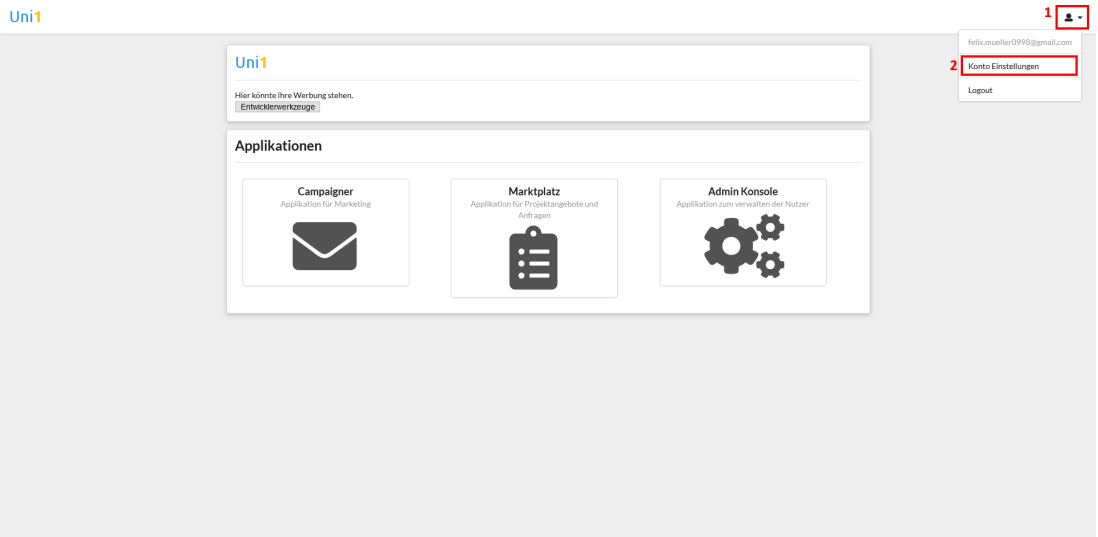

Figure 6.5: Dashboard homepage navigation to account management

This will open the page in the Dashboard as seen below. On this page it is possible to edit the main values like your first name ('Vorname'), last name ('Nachname'), email, phone number ('Telefonnummer'), title ('Title'), gender and also manage the tags you are interested in and are used for the emails of the Campaigner.

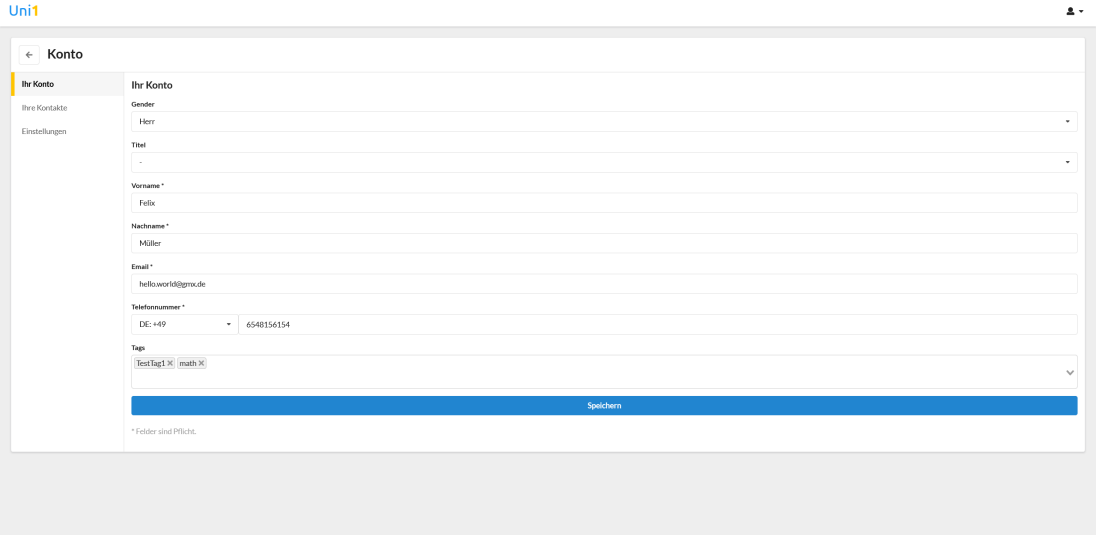

Figure 6.6: Account management main page

In general, nothing more is required from the user after pushing the save button ('Speichern') to update his user. The exception to that is, if you change your email address, which causes Cognito to send another verification email to this new address, as explained in the implementation chapter.
Due to this you will see the dialog below, where you are required to either input the sent verification code, or to log out.

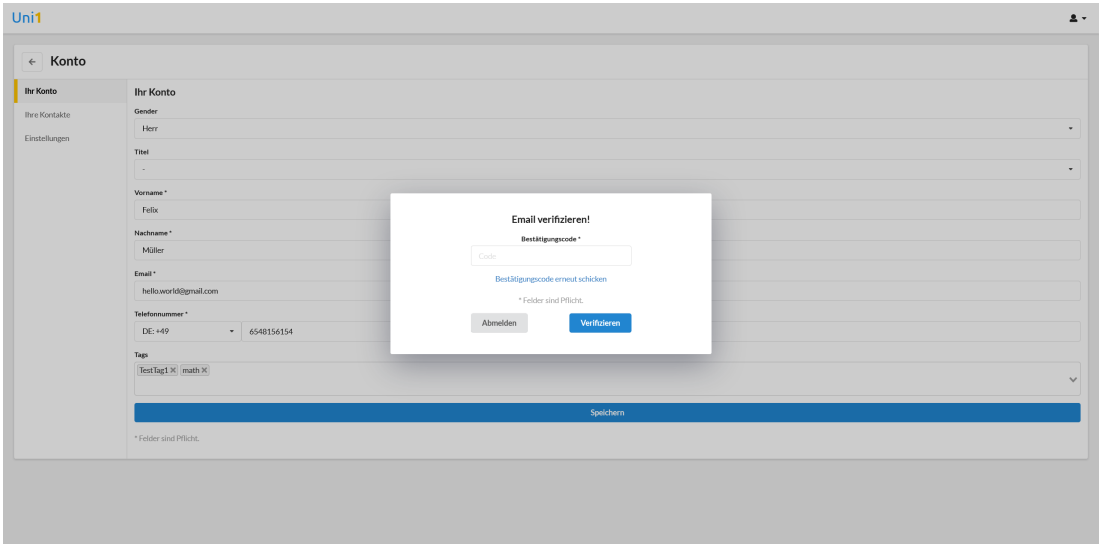

Figure 6.7: Verify email dialog after email change

Pressing 'Abmelden (Log out)' has no major effect, because then the user will be asked to enter the verification code immediately after the next log in, similar to

after the sign up.

Aside from the default page 'Ihr Konto (Your Account)', there are two more tabs, the first one being called 'Ihre Kontakte (Your Contacts)'.

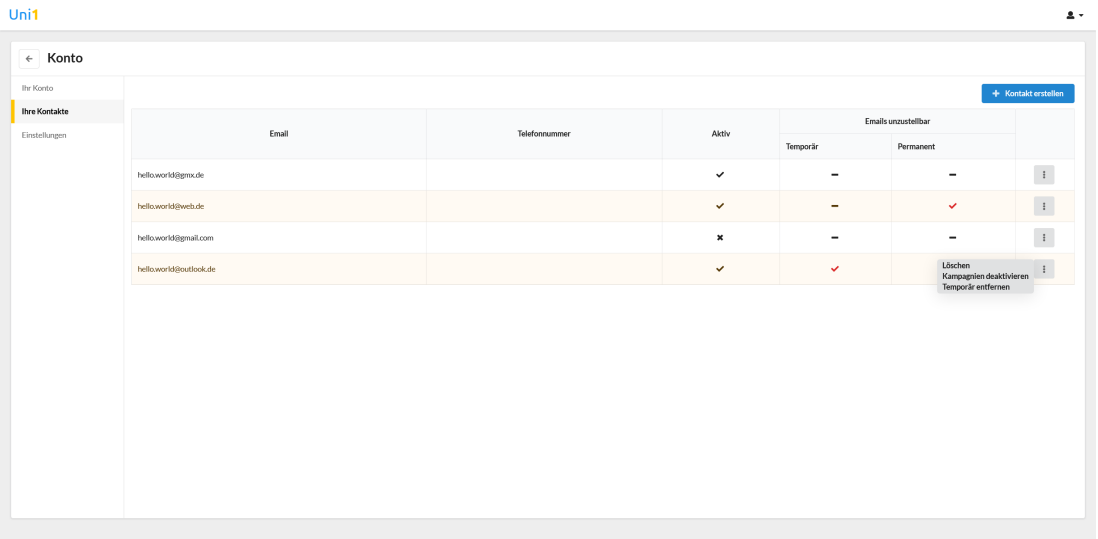

Figure 6.8: Account management contact management page

Here it is possible to create new contacts using the button in the upper left, or to use the action menu for each contact to delete this contact, dis- or enable it for campaign emails or to remove the temporary bouncing flag. The last action is needed, if, for example, at one point the inbox of this email address is full and thus it can no longer receive any emails, returning a temporary bounce to the sender. After you have cleaned up the inbox you can clear the flag and again receive campaign emails (if this was the only reason why it failed before).

The last tab is the 'Einstellungen (Settings)' tab, where it is possible to change the password.

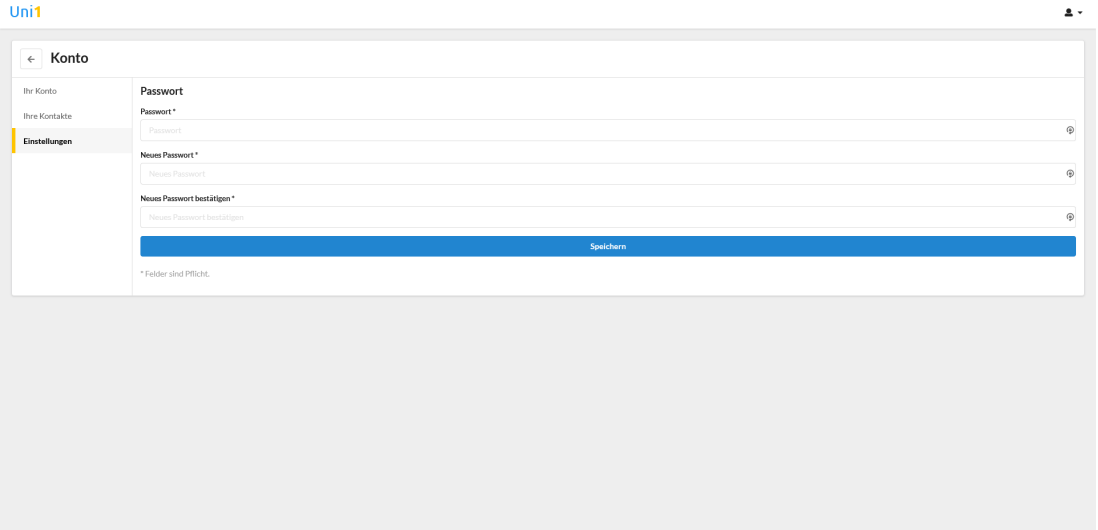

Figure 6.9: Account management settings page

6.2 Campaigner

This section is all about demonstrations concerning the Campaigner component. Both demos start on the homepage of the Campaigner located at https://campaigner.uni1.de.

6.2.1 Campaign creation

In this first subsection it will be shown, how a new campaign is created.

In order to create a new campaign, the user has to push the 'Neue Kampagne erstellen (Create new campaign)' button in the upper right of the Campaigner homepage.
After this a new page will open were all necessary information about the new campaign have to be inputted.
The first two input fields of this page are for meta data of the campaign with the first being called 'Titel der Kampagne (Title of campaign)', where a short title which describes the campaign in a few words has to be entered. The next field, 'Kurze Beschreibung (Short description)', is where a short description is to be entered that further details the purpose of this campaign for other users.
Next are the inputs, that are relevant for the sent email. The first is labeled 'Betreff der Email (Subject of email)' and has to contain the subject, that the campaign emails are supposed to have. This is followed by the 'Text der Email (Body of email)', a textarea input containing the body of the emails, including newline support. At last comes the 'Tags der Kampagne (Tags of campaign)'

dropdown, where it is possible to restrict the recipients using the tags, which each user can subscribe to.
Shown below is the page with exemplary data filled into each field.

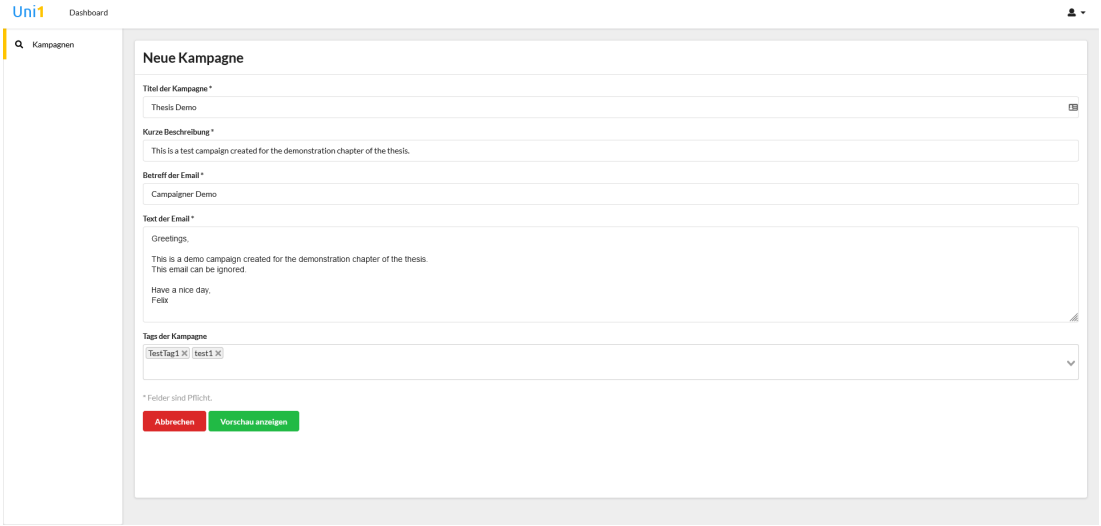

Figure 6.10: Campaign creation part, where the data has to be inputted, with exemplary data

After everything has been inputted, it is possible to advance the creation with the pushing of the green button labeled 'Vorschau anzeigen (Show preview)'. After this the page below will be shown, detailing how the email will look like for each recipient (ignoring differences caused by different email provider and email applications).

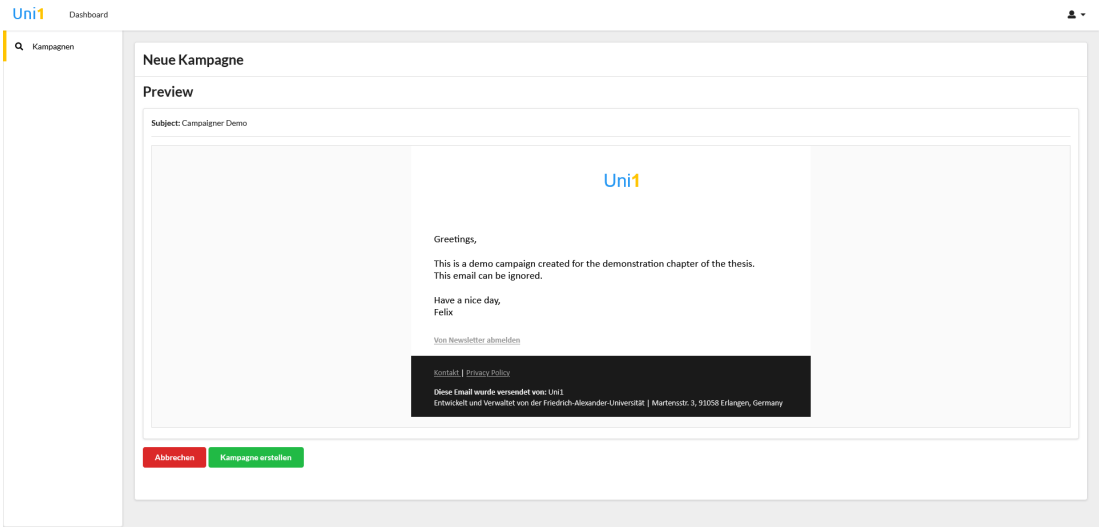

Figure 6.11: Preview of the email

It is possible to go back to the input page of the campaign creation, should the user wish to edit, for example, the body of the email, for which it is simply necessary to press the red 'Abbrechen (Cancel)' button.
After pressing the green 'Kampagne erstellen (Create campaign)' button, the new campaign will be created, including the start of the email sending, and the creator will be redirected to the campaign list page, which now features his new campaign at the top.

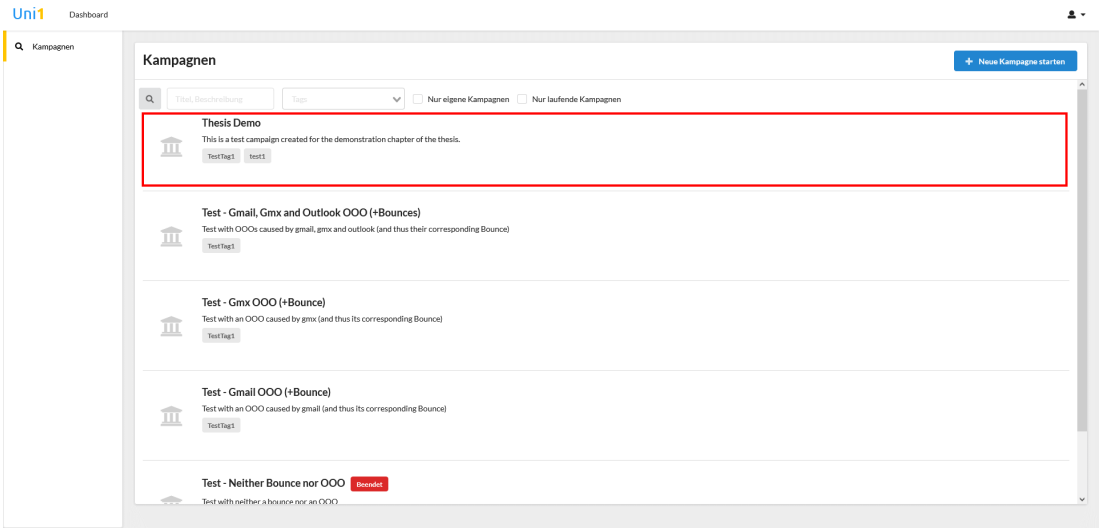

Figure 6.12: Campaign list after creation of new demo campaign

6.2.2 Campaign details

After a campaign has been created, it is possible to open its details page which shows the most important data. This data contains the meta data given during the creation, namely the 'Titel (Title)', and the 'Kurzbeschreibung (Short description)'. Meta data not inputted during the creation includes the 'Status (Status)' indicating whether the creator has closed this campaign or not, as well as the 'Besitzer (Owner)', meaning the creator of the campaign. The last things the page displays are the 'Betreff (Subject)' of the campaign email and the tags, which were used to determine the recipients.

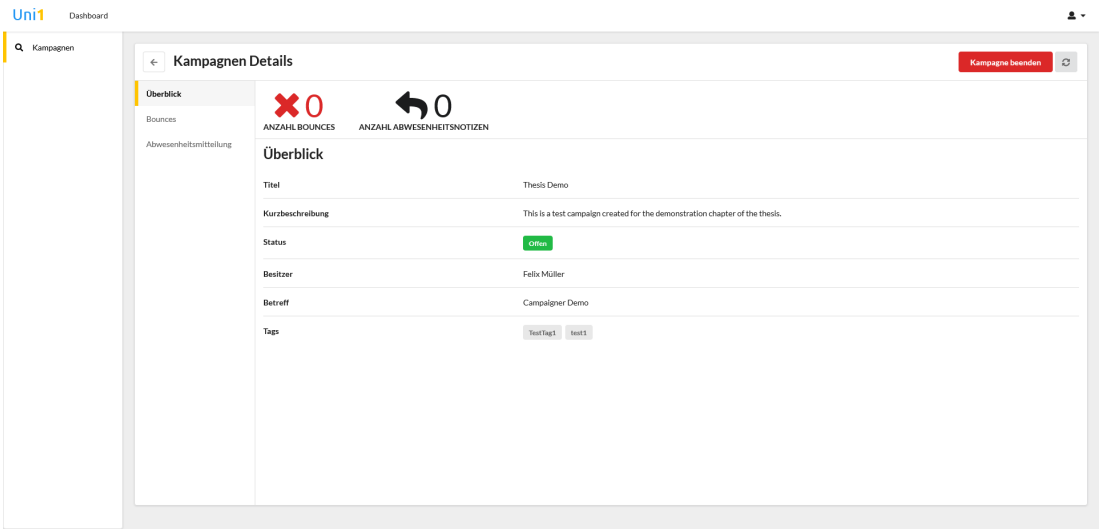

Figure 6.13: Campaign details page for the campaign, created during the campaign creation demonstration

This page also shows the number of bounces and Out-Of-Office (OOO) replies that were received for this campaign, but because none were received for this campaign they both show the number zero. Because of this, the other two tabs only contain empty lists, after all nothing was received and because of that a new campaign was created with bounces and OOO replies, that will be used for this demonstration from now on and whose details page can be seen below.

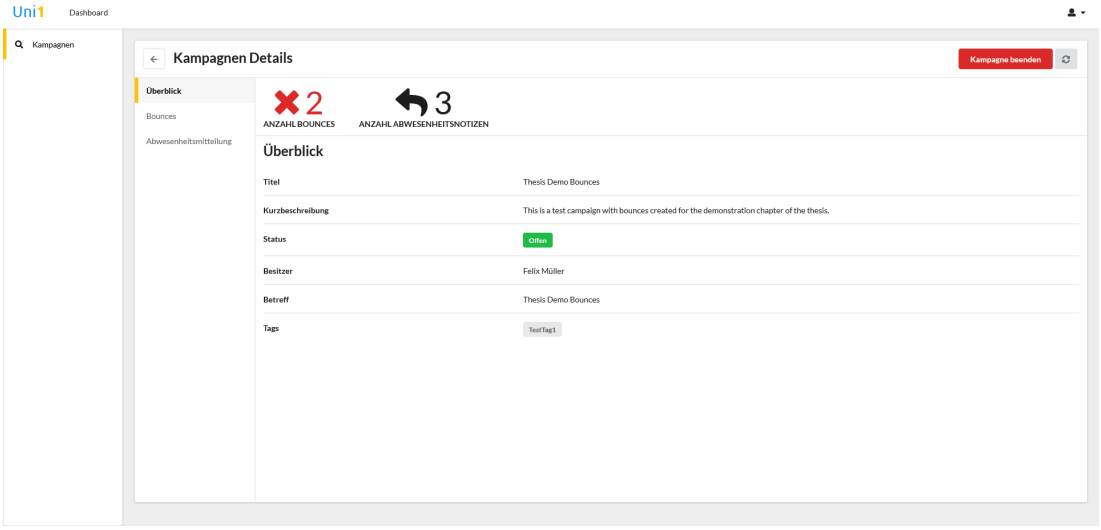

Figure 6.14: Campaign details page for a new campaign which received bounces and OOO replies

As you can see, the statistics show two bounces and three OOO responses, meaning that we can now show the tab 'Bounces' with actual data. If there are no bounces, this page will only contain the header of this list.
This list has one entry for each bounce and displays the 'Empfänger (Recipient)' of the email that returned the bounce, so essentially the email that caused the bounce, the bounce type, as classified by AWS SES, and the 'Datum (Date)' the bounce was received. The bounce type column also has a little info icon and hovering over it with the mouse opens a little tooltip with a short description about the bounce type.

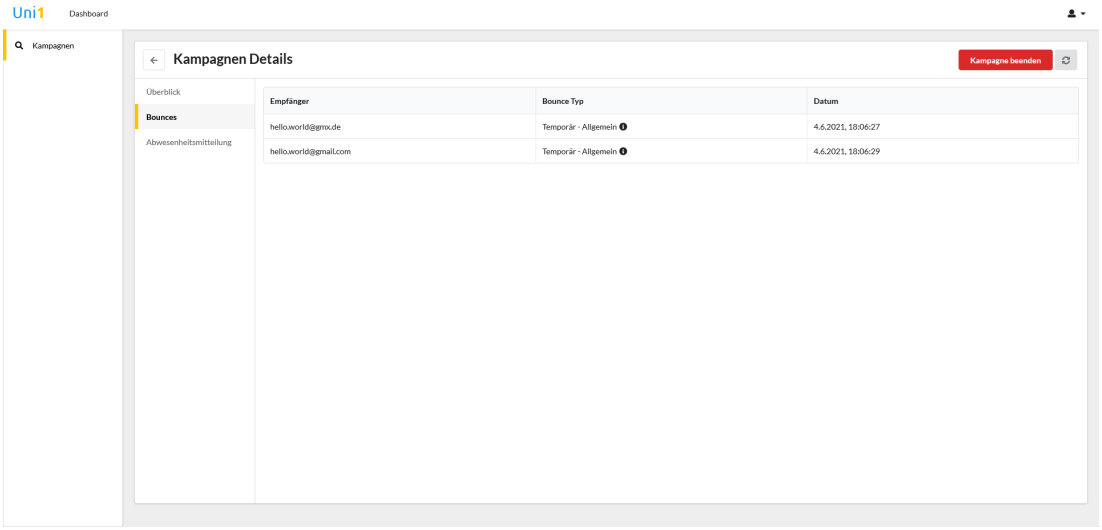

Figure 6.15: Bounces list for campaign with two bounces created due to OOO responses

Selecting an entry of the list opens the bounce details page for this bounce.
Here all the details about the bounce are displayed, like the 'Empfänger (Recipient)' and the bounce type from the table, as well as a description about the bounce type.
In addition to that there are a few actions available. Two of these actions are in the upper right, with the first one allowing the deletion of the contact which caused the bounce. This action is primarily for the case, that this is a permanent bounce caused, for example, the fact that the email address does not exist. In this case the button is disabled, because this contact is the main contact for a user, meaning the account of a user is using this email address. The next action is labeled 'Bounce Flag entfernen (Remove bounce flag)' and pressing this button essentially tells the backend to ignore this bounce. This button is for the opposite type of bounce than the previous one, meaning for temporary bounces like 'Mailbox full'.
The last action 'Abwesenheitsmitteilung öffnen (Open OOO reply)' is part of the

table which displays the bounce information and is only shown, if this bounce was caused by an OOO reply. This allows the user to quickly open the details page of the corresponding OOO reply.

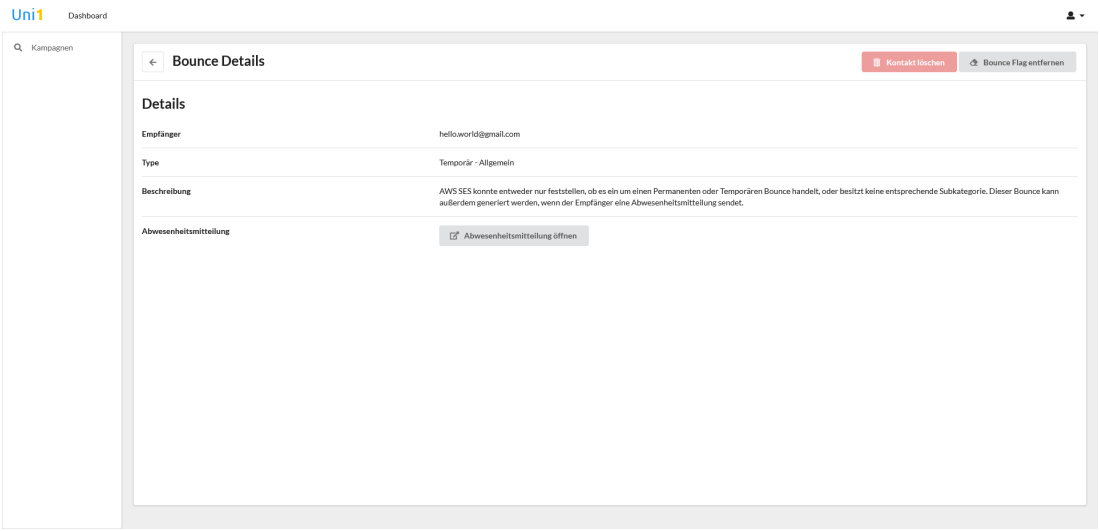

Figure 6.16: Details page for a OOO bounce

The other tab 'Abwesenheitsmitteilung (OOO reply)' on the campaign details page is analogue to the bounce list with the difference, that it contains the OOO replies. This list also has different columns, namely 'Sender (Sender)', 'Betreff (Subject)' and again the 'Datum (Date)'.

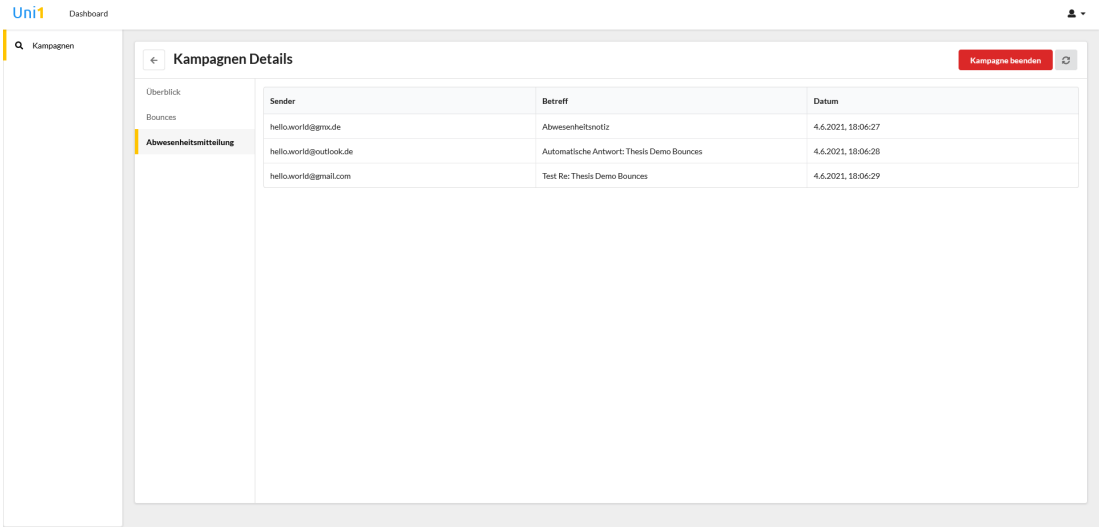

Figure 6.17: OOO replies list for campaign with three OOO replies

This page too allows the opening of the OOO reply details page on entry select and it also displays the general information.
Aside from the 'Sender (Sender)' and the 'Betreff (Subject)' from the list page before, it also contains the recipient, which is always the official Uni1 campaigner email, and the content of the OOO reply.
Another thing that is identical, are the actions in the upper right for deleting the contact or removing the bounce flag.
The one additional thing that this page has, but the bounce page does not, is linked to the content of the OOO reply. If the OOO reply contains an email in its body, this email can be seen in the list at the bottom, together with an icon to its left and sometimes a button to its right. The icon on the left is either a green check mark, indicating that a contact with the corresponding email already exists, or a red cross, indicating the opposite. The button, labeled 'Hinzufügen (Add)' will only be display, if no contact for the email was found and allows the user to create a new contact with the given email address on the fly. This new contact will use the same tags for the Campaigner as the contact which replied with the OOO and is assigned to the same user, if the other contact is assigned to one. In the case that there is at least one email in this list, there will also be a button to the right of the header 'Email Addressen', labeled 'Alle hinzufügen (Add all)', which will create new contacts for all email addresses in the list at once.

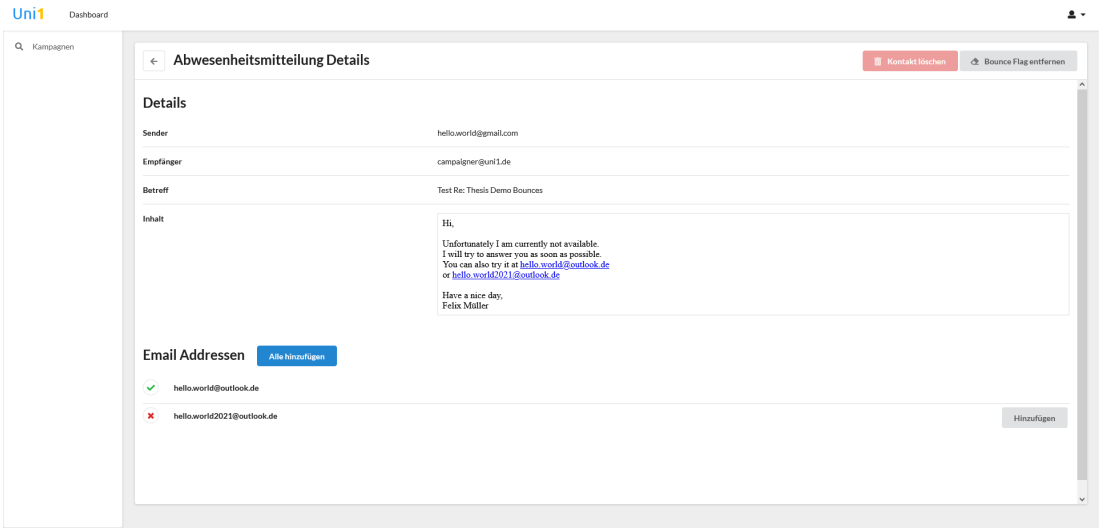

Figure 6.18: Details page for OOO reply sent by main contact (**Note:** The sender of the campaigns has been changed a few weeks before the submit date of this thesis to 'dirk.riehle@uni1.de')

7 Evaluation

In this section, the new Uni1 application will be evaluated and the known issues. and suggestions on how to fix them, will be highlighted.
Lastly it, again, has to be noted, that this application relies even more heavily on AWS than the previous one, meaning that it would be very hard, up to impossible, to switch to a self-hosted solution, or even to other cloud providers like Azure[1]. The main reason for that is the usage of Cognito, whose removal would require extensive refactoring. Same accounts for the changing of the email provider, because the handling of the OOO replies would have to be refactored.

7.1 Marketplace (Old application)

For the old application it has to be noted, that unfortunately it was not possible to port the old application, due to the lack of time and the decision to focus on the Campaigner. Because of this, the old backend merely had the old authentication and authorization disabled as well as everything, that would refrain the application from compiling, as described in the implementation section.
What was possible, was the porting of the Semantic UI implementation, such that the new VueJs applications have the same style as the old one.

7.2 Dashboard

In case of the authentication/Dashboard component, all major requirements have been fulfilled. The account management works with the same features as the old one and has some additional ones caused by the Campaigner component. However it has to be mentioned, that there are some issues concerning the relation between the backend implementation and Cognito, caused by the fact mentioned in the implementation, that not everything can be saved inside Cognito, causing an issue during registration. As long as the registration succeeds, or fails because of Cognito, everything works fine. The problem arises when it fails during the user

[1]https://azure.microsoft.com/en-us/

creation of the backend, at which point the api will cleanup and delete the already created Cognito user. The reason why the Cognito user was already create, is the fact, that the Cognito sign up is executed before the backend signup. The issue now is, that after its signup, Cognito also sends the email with the verification code, causing the verification code email to be received by the user, even though the registration failed. This is obviously not wanted, but unfortunately it was discovered too late. One suggested solution would be to customize the Lambda, Cognito uses to send the verification email, and to integrate the backend signup into it. This however has first to be tested, concerning feasibility, because this Lambda would have to be able to send HTTP requests, and cost wise, because this would increase the runtime of the lambda and thus the price.

7.3 Campaigner

The Campaigner is similar to the Dashboard, in the case that all major requirements have been implemented with some impossible/impractical parts or issues. The impractical part was about the status, caused by the fact, that it is not possible to know for sure if all emails have been sent. The reason is the fact, that SES only tells you that your request for the sending of the emails has been received and accepted, but not when it is finished. In order to detect that, it would be required to enable the email received notifications, analogous to the bounce notification, and then log these events for each email. This would mean too much workload for the backend, in my opinion, because it is possible to create multiple campaigns for all contacts, which can be hundreds if not thousand emails. Because of that, the only status implemented are an equivalent of 'open' and 'closed'. The next one is the impossible part and more or less also concerning the status, to be exact about the start and stop the email sending functionality. The reason why this is not possible, is the fact that SES does not have any functionality to stop sending emails.
Concerning the issue, there are two and the first one is potentially a major one. This first is about the handling of OOO replies and has been addressed a few time throughout the thesis. More precise, it is unfortunately not possible to guarantee, that the current OOO handling will work with all OOO replies, due to differences between each email provider. This is the part of the Campaigner with the potentially highest maintenance effort.
The second issue is only a small oversight, which should be mentioned either way, and is the fact that the public unsubscribe page still displays the 'Kampagnien (Campaigns)' tab in the navigation bar on the left. Trying to open it will only display the insufficient permission page, but this navigation bar should be removed either way.

7.4 Continuous integration & continuous deployment

Lastly for the CI/CD there is not much to evaluate. The pipeline itself works and deploys the code on merge to the master branch, while testing on all branches, as long as changes were detected in that component. The only thing that can be noted is the fact, that, as mentioned, the pipeline currently uses four different Docker images for the stages. Here it would be possible to create a custom image for the pipeline, such that only one would be required for all stages. This image could then be uploaded to the GitLab Container Registry (GitLab, n.d.).

8 Conclusions

This thesis discussed and detailed how the pre-existing Uni1 application was refactored into multiple components as well as the implementation of such a component. Furthermore, it detailed the implementation of an application with multiple frontends hosted on different subdomains using Cognito and Amplify for authentication. In addition to that, it highlighted how a complex email system with Out-Of-Office reply handling can be achieved using SES as the email provider. Lastly the thesis demonstrates, how such an application can be deployed to AWS using continuous integration and continuous deployment.

Appendix A Backend new directory structure

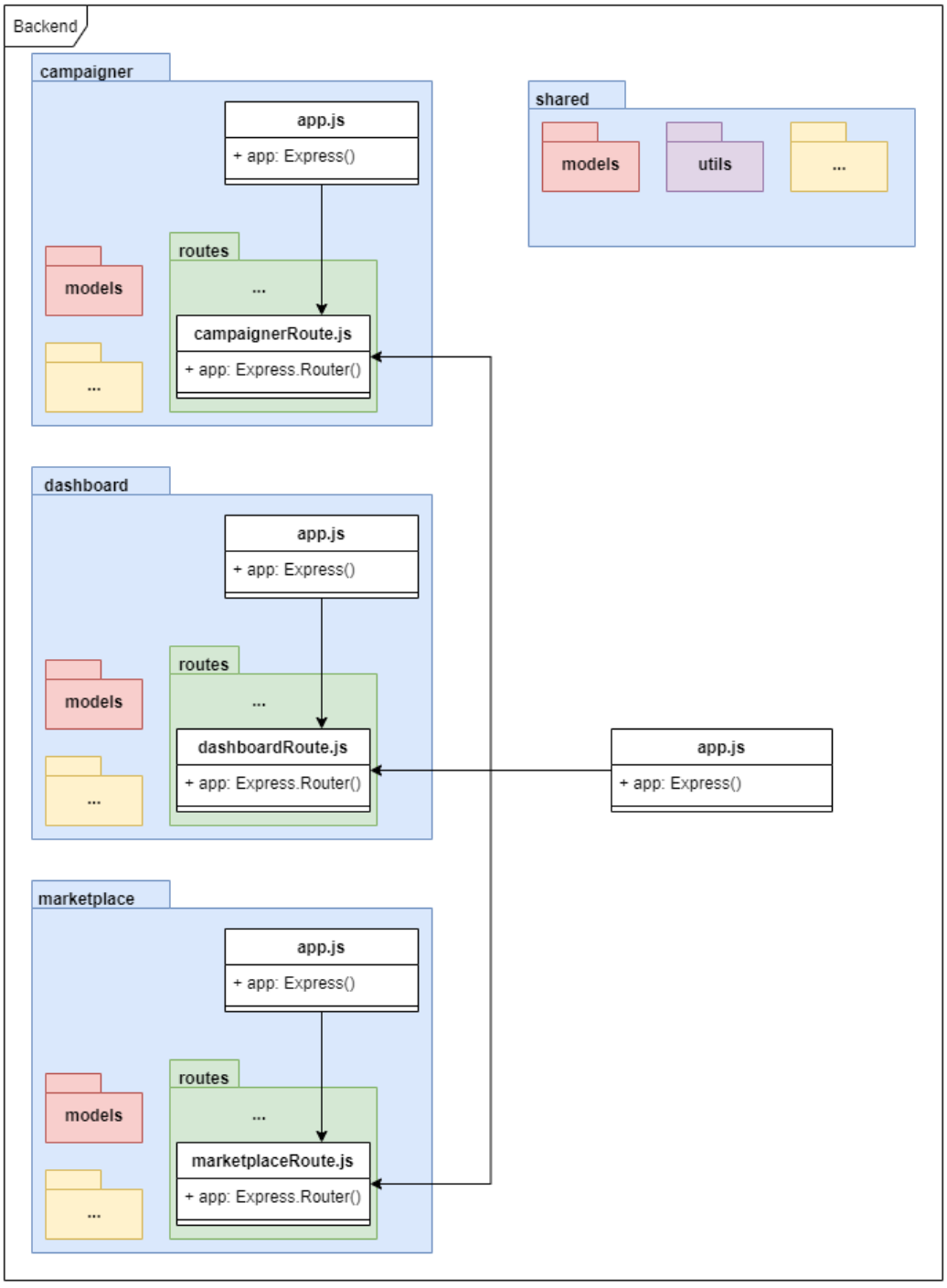

Figure 8.1: New directory structure of backend with separate subdirectories for each component. A more detailed explanation can be found *here*

Appendix B Backend signup diagram

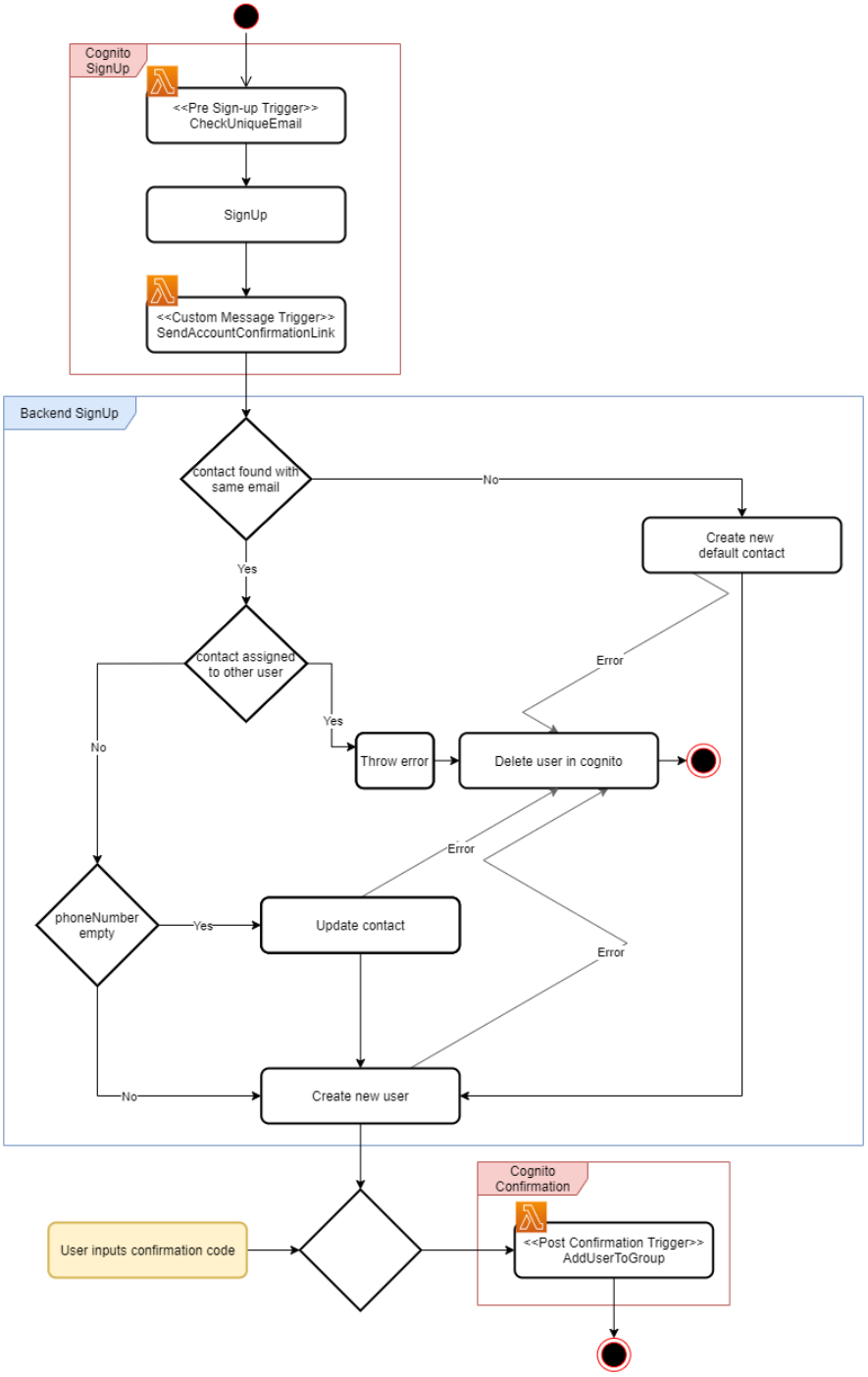

Figure 8.2: Activity Diagram of complete signup flow, including backend and Cognito signup

Appendix C OOO handling diagram

Note: For simplicity most of the error handling has been removed and only the most important checks remained!.
Removed were, for example, the error handling if one helper function, like the header extraction, returns nothing.

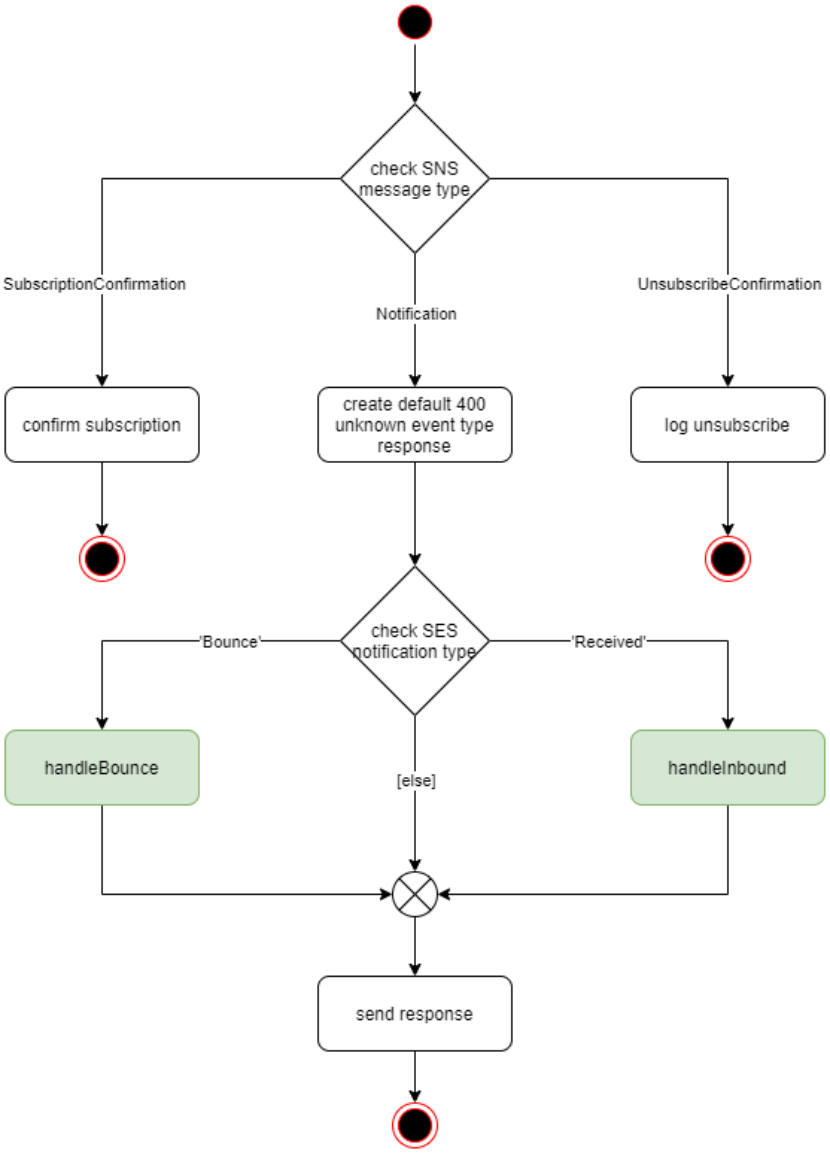

Figure 8.3: Root diagram where the OOO-handling starts

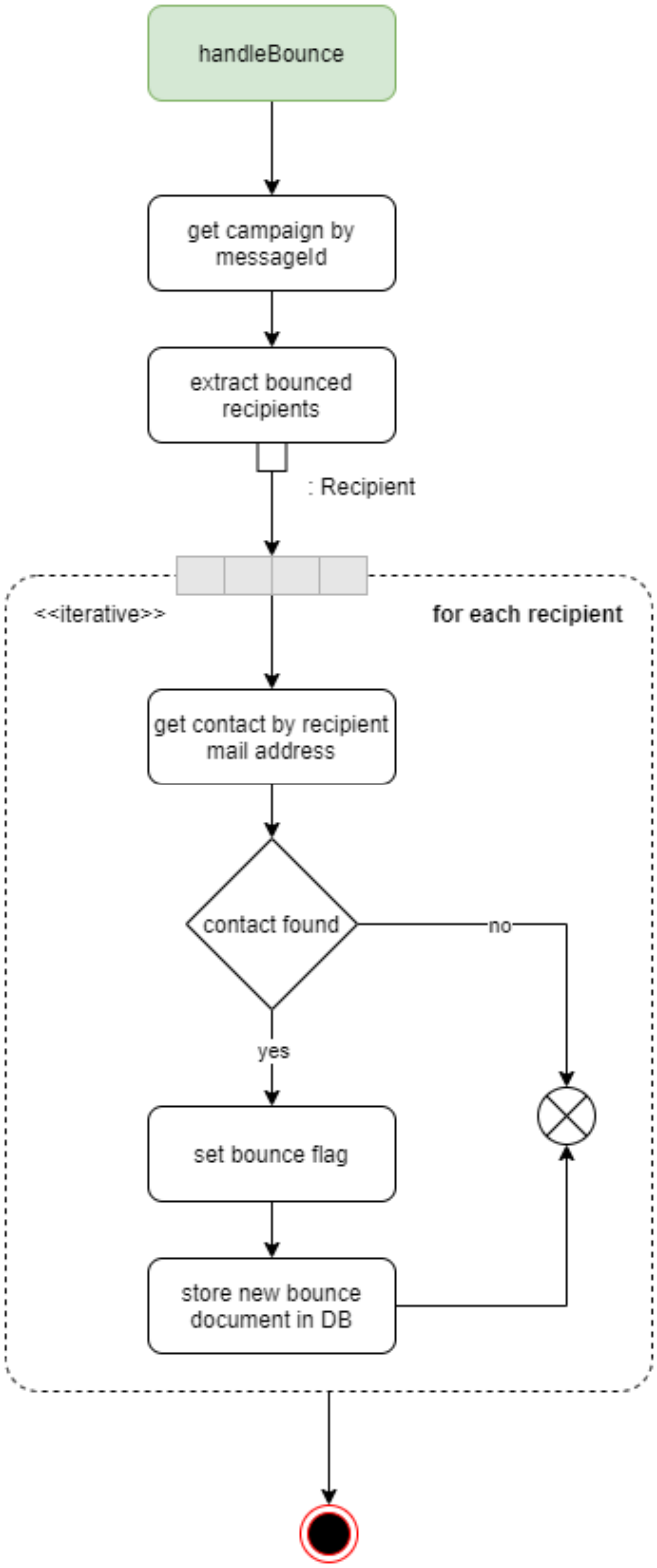

Figure 8.4: Diagram for handling of bounces

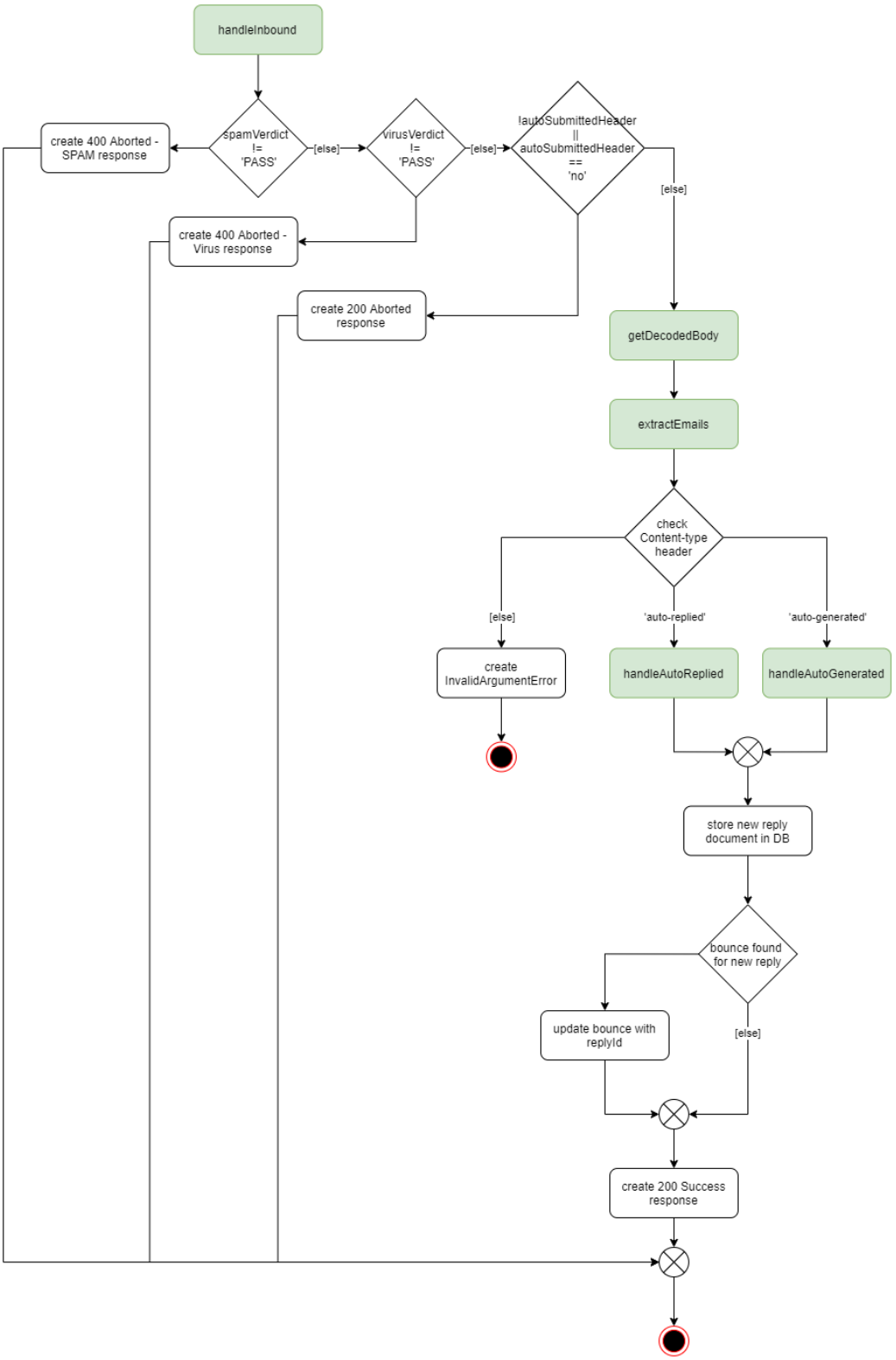

Figure 8.5: Diagram for handling of inbound emails with references to other functions displayed in the following diagrams

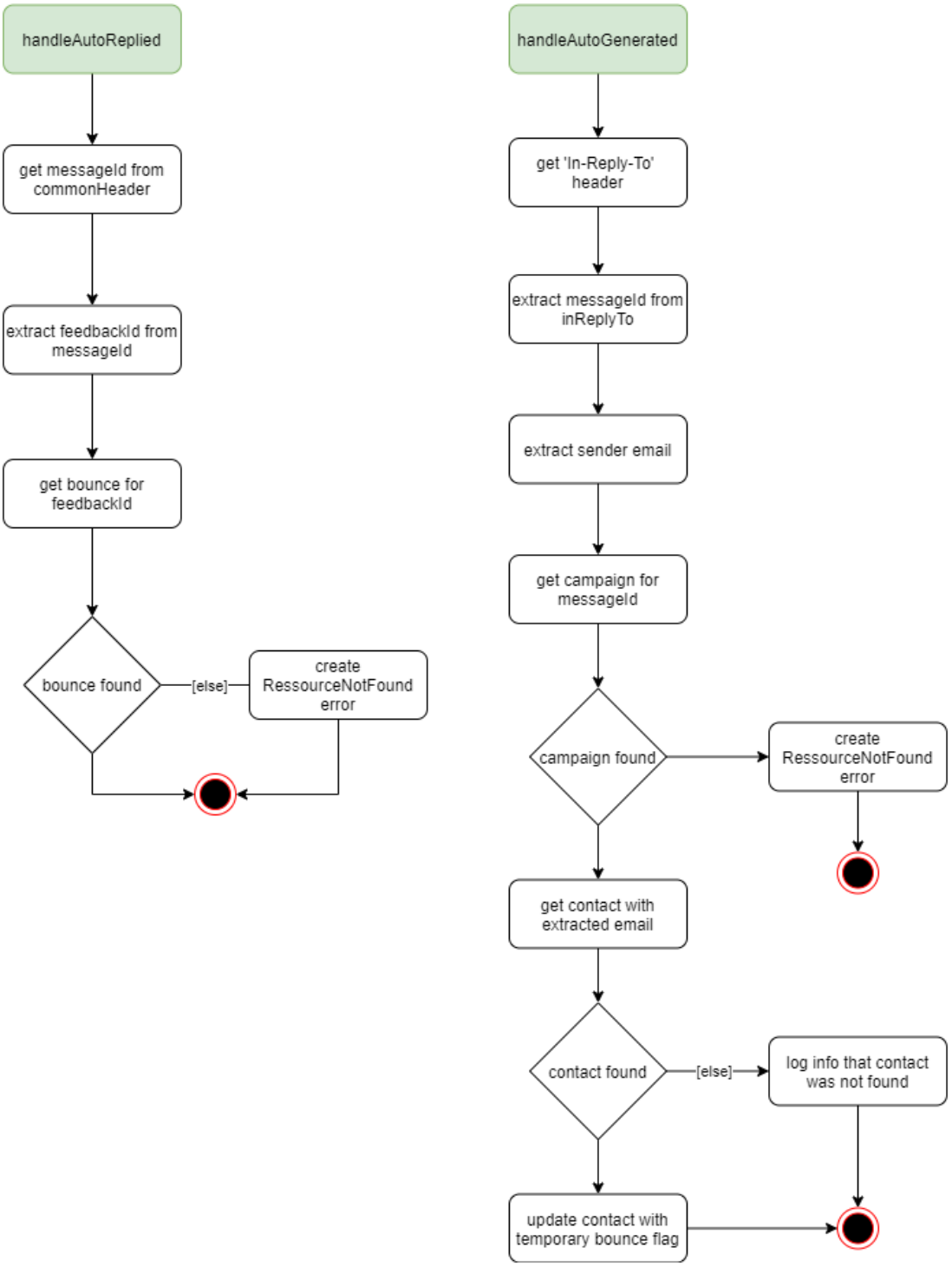

Figure 8.6: Diagrams for the different sub-process of data extraction depending on the autoSubmittedHeader

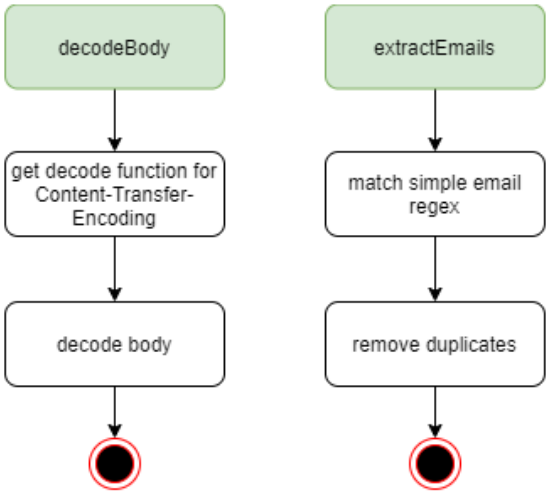

Figure 8.7: Diagrams of small helper functions used in the upper diagrams

References

Amazon simple email service endpoints and quotas. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/general/latest/gr/ses.html

Amazon sns notification contents for amazon ses. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/ses/latest/DeveloperGuide/notification-contents.html

Amplify ui components. (2020). Amazon Web Services, Inc. https://docs.amplify.aws/ui/q/framework/vue

Amplify-js/packages/amplify-ui-vue/. (2020). https://github.com/aws-amplify/amplify-js/tree/master/packages/amplify-ui-vue

Amplify-js/packages/aws-amplify-vue/. (2020). https://github.com/aws-amplify/amplify-js/tree/master/packages/aws-amplify-vue

Aws cloudformation endpoints and quotas. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/general/latest/gr/cfn.html

Aws::ses::configurationset. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-ses-configurationset.html

Aws::sns::subscription. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-sns-subscription.html

Configuring build settings. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/amplify/latest/userguide/build-settings.html

Configuring user pool attributes. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/cognito/latest/developerguide/user-pool-settings-attributes.html

Confirmsubscription. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/sns/latest/api/API_ConfirmSubscription.html

Create-application-version. (2021). *Amazon Web Services, Inc.* https://docs.aws.amazon.com/cli/latest/reference/elasticbeanstalk/create-application-version.html

Creating a receipt rule set for amazon ses email receiving. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/ses/latest/DeveloperGuide/receiving-email-receipt-rule-set.html

Css guide/email clients/border-radius. (n.d.). https://www.campaignmonitor.com/css/box-model/border-radius/

Eichhorn, P. (2016). *The uni1 immune system for continuous delivery* (Master's thesis). Professorship for Open Source Software - Friedrich-Alexander-Universität.

Git-based deployments. (2020). Amazon Web Services, Inc. https://docs.amplify.aws/guides/hosting/git-based-deployments/q/platform/js

GitLab. (n.d.). *Gitlab container registry*. https://docs.gitlab.com/ee/user/packages/container_registry/

How do you want users to be able to sign in when using your cognito user pool? username + email + phone. (n.d.). https://github.com/aws-amplify/amplify-cli/issues/1546

Lambda triggers. (2021). Amazon Web Services, Inc. https://docs.amplify.aws/cli/usage/lambda-triggers

Managing and searching for user accounts. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/cognito/latest/developerguide/how-to-manage-user-accounts.html

Moving out of the amazon ses sandbox. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/ses/latest/DeveloperGuide/request-production-access.html

Multiple frontends. (2020). https://docs.amplify.aws/cli/teams/multi-frontend

Nasser, N. E. (2021). *Uni1 application to containers* (Master's thesis). Professorship for Open Source Software - Friedrich-Alexander-Universität.

Set up an amazon sns event destination for event publishing. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/ses/latest/DeveloperGuide/event-publishing-add-event-destination-sns.html

Setting up a custom mail from domain. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/ses/latest/DeveloperGuide/mail-from.html

Setting up easy dkim for a domain. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/ses/latest/DeveloperGuide/send-email-authentication-dkim-easy-setup-domain.html

Signing up and confirming user accounts. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/cognito/latest/developerguide/signing-up-users-in-your-app.html

Step 3: Specify your configuration set when you send email. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/ses/latest/DeveloperGuide/event-publishing-send-email.html

StrongLoop, I. & other expressjs.com contributors. (2017). *Using middleware*. https://expressjs.com/en/guide/using-middleware.html

Verifying a json web token. (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/cognito/latest/developerguide/amazon-cognito-user-pools-using-tokens-verifying-a-jwt.html

What is aws cloudformation? (2021). Amazon Web Services, Inc. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html