Alphamail

Group 4

Malthe Tøttrup
Department of Computer Engineering
Aarhus University
Aarhus, Denmark
201907882@post.au.dk

Daniel Christopher Biørrith
Department of Computer Engineering
Aarhus University
Aarhus, Denmark
201909298@post.au.dk

Jens Villadsen Fisker
Department of Computer Engineering
Aarhus University
Aarhus, Denmark
201908671@post.au.dk

Lukas Koch Vindbjerg
Department of Computer Engineering
Aarhus University
Aarhus, Denmark
201906015@post.au.dk

Instructor: Lukas Esterle
Department of Engineering
Aarhus University
Aarhus, Denmark
lukas.esterle@eng.au.dk

CONTENTS

I. Introduction

In today's world communication over great distances is often trivialized and its importance under appreciated. We use emails every day as we communicate with remote colleagues, distant relatives, doctors and many other vital parties of society. When doing this, you, like many others, most likely use a tech giants email-service such as Google's Gmail or Microsoft's Outlook. Using such an email-service is often effortless, as the graphical interface has been developed and improved for decades and has a plethora of quality of life features to help you. Behind this pretty interface, however, hides a massive amount of hours of coding and debugging which this project sets out to explore as we create our own email-service.

In this paper you will explore how we implemented our email-service and the design decisions we took in the process, how the server is build and how it communicates with the client, which packages we used and what they did to help us, and finally this paper allows us to look behind the pretty interface of email-services to help us understand the mechanisms behind them.

II. Implementation

The Alphamail email-service[1] is as web-application, designed to run in the browser like many of the most popular email-service providers, such as Gmail or Outlook. The server side of Alphamail is build using NodeJS, with many of the middleware packages that comes with it. Firstly, this middleware allows us to design the client side using HTML for the layout, and use embedded JavaScript to handle all request response interactions between the client and the server. Secondly, we use the middleware to make communication between the server and the database possible, such that we can save and store emails, users, drafts, etc.

The server is build using two main methods; `post` and `get`. These are used for handling a request/response interaction between the server and the client. The `get` method is used whenever a user clicks a link on the interface, which will redirect to another page, with no input information required. An example of this is when the user is at the home page and clicks the link leading to the inbox page. `post`, on the other hand, is used when the user specifies information in input fields and clicks a submit button. The information is handled by the corresponding functions, and the user is redirected to the appropriate page, once the process is finished. An example of this is when a user is on the login page and types in an email, password and submits. The server receives the information, and verifies that it matches what's stored in the database. If conditions are met, the server will redirect to the homepage of the specific email address. However, if the information was incorrect, the server will redirect back to the login page, for the user to try again. All the technical functionalities, such as sending an email or checking the database, are located in functions called by the `post` and `get` methods.

We have designed all the accessible web pages individually, using HTML. However, some of the pages are templates and will be changed according to the user logged in. An example of this is the inbox, which will only show emails belonging to the current user. Navigating through different pages is done by clicking link-assigned buttons, controlled by the `get`/`post` methods.

To ensure that these pages are not accessible without being logged, in we use login sessions and cookies. This ensures that if someone tries to manually type in an address, for instance "/homepage", the request will get rejected. Also, if a user is inactive for a set period of time, the user will automatically be logged out.

A. Decisions

Initially, we decided to use NodeJS as the main programming language. We chose this language due to the fact that it was simple to implement a client-server architecture, and we wanted our app to run in the browser making it easier to develop a user interface. However, we had little to

no experience with NodeJS and web-development beforehand. During our research process, we started to get more and more frustrated, since we lacked the knowledge to understand the sources we found on the subject.
At one point it seemed unfeasible for us to continue in the chosen direction. We wanted to make a server that could be communicated to from external devices, and found it to be impossible with NodeJS, without purchasing a domain. This prompted us to investigate resources on the topic written in other programming languages, which we were more familiar with. In this process, we came across a source that implemented a SMTP server in Python[1]. Some team members explored how we could modify the implementation to our own conditions, and even set up a login system in Python. However, the issue was, that it seemed very difficult to couple this up on the client-server based architecture we originally intended, making a GUI client, which was the whole reason we chose NodeJS in the first place. Additionally, though it was a SMTP server, it would also only work on local implementations.
We found ourselves at a crossroad: should we go with the programming language we had more experience with, and had an already implemented login system? Or should we go with the language that seemed to be better suited for the tasks at hand?

In the end we decided to go with NodeJS since we found a good source [2] that could help us get started learning the language and its concepts. However, we found that we could use the knowledge gained implementing the python login system, and transfer it to implement our own login system using NodeJS and MySQL. As we progressed with the development we quickly became more confident with NodeJS and its middlewares. Thus, we found a decent amount of success implementing some of the features and concepts we discussed in the requirements document[3] and architecture and design document[4].

Another decision we had to make was how we wanted to proceed with a SMTP server. SMTP allows multiple devices to send data via emails from one server to another. In this process, both parts agree and accept the exchange of data, thus the email can be sent. However, since we didn't have a domain that could be communicated to, we could only receive from users logged into the same server. Thus, we could skip all the preceding steps implemented in SMTP, and only focus on the transfer of data. SMTP is only necessary when we wanted to send mails externally, where we use a NodeJS function that does all the needed communication. With this, when we send emails to e.g. a Gmail account, the receiver of the email cannot respond to the email, since the email address does not exist within a real SMTP server. However, when we send a mail from one @alphamail to another, we have the means of storing this email in the database, thus we can "simulate" the behaviour of a real email-service system.

B. Reuse

The following are different, small implementations we found online, and incorporated into our system:

- A function that allows us to send emails, which we imported from the internet. This means that all our external email traffic is handled by a function we imported and reused on our server [5].
- The 'good source' we mentioned earlier also provided the basics of setting up the server on the socket, and setting up communication with mySQL, which we used in the final implementation [2].
- As mentioned earlier, we also used multiple middlewares we downloaded and installed, which helped us handle a variety of different tasks. Since we didn't develop these from scratch, we can say that we reused those as well. To see the full list, see the top of the 'app.js' file.

- All the CSS we used for the layout of our HTML files is taken directly from at template that we found on the internet as well, with minimal modifications done by us[6].
- A source on how we could display data from our mySQL database in a table, which we took and modified [7].

C. Difficulties we encountered

One of the biggest issues we encountered during our implementation wasa lack of networking knowledge, when we first approached the application. We believe that we had an adequate understanding on a theoretical level of how a server and client were structured. This meant that in the design and architecture document where we discussed the design, features and general structure we could easily imagine these on a centralized system with programming languages Python or C++ which we were more comfortable with. This, however, had the drawback that once we decided to tackle these design decisions we realized that some of them were easier said than done.

The first difficulty we encountered, was how to implement sending and receiving emails. In the documentation we said that we wanted to use SMTP for sending, and either POP3 or IMAP for receiving. However, this proved to be quite the challenge for us, since we didn't know how to get started with this. In the end, we decided to take a different approach for the server, which we talked about in the decisions section. For receiving email, we chose to rely on the same properties, as IMAP builds upon; only saving emails on the server side.

However, the solution to this problem presented another problem. We wanted to use a database to store an email that is sent/received and for this we used MySQL. As we had no experience with it, we had to learn the native MySQL language, and how to use it with our NodeJS server implementation. After we figured out how to tie these components together, we uncovered a new problem, which was how to take a user input to these functions. To do this we had to figure out how HTML and NodeJS interact and work together. After some research, we finally understood how to use these three components together, and we could now really start to make some progress on the implementation of the email-service.

As we mentioned earlier, you could only connect to the server on localhost. With this, we had to be on the same network, in order to test it on different machines. Due to the lockdown of the school, this was no longer a possibility, meaning we had to continuously develop and test the application on just one computer. However, we could use a visual studio package, called live-share[8], to develop simultaneously in the same files (similar to google drive or overleaf), which helped us a lot in the development process. Additionally, we made use of Github, for everyone to have the same version of the code.

D. Work distribution

Throughout the semester, we would spend the majority of the exercise lectures working on the design and what tools we would use to implement the code. Thus, prior to the implementation, we would discuss together and unanimously decide on what we were to do for the implementation process. When it came to the final implementation, we decided to work in a sprint on the days leading up to Christmas. Initially we met up physically and decided on what we were going to reuse and how the implementation should be completed. After a few days we all went home to our families for Christmas, which meant that we had to work online. We did this by sitting together in a voice call working on the code where we had distributed different work tasks. The way we would still work together was by using Visual Studio Code's 'live share' feature which allowed for remote editing of the code. So for most of the online work, Malthe would be the one who would run 'live share' with the code – as we had everything set up for his computer so he could test it – and the rest of the group would then work on the application remotely. With this in place, Daniel and Malthe focused pri-

marily on the server, while Lukas and Jens worked more on getting the front-end client side working. This is not to say we completely disbanded each other and split into two separate units. Everyone was still aware of most of what the other group was developing, as it was also necessary to be informed about the other groups process and development.

III. The features of Alphamail

When implementing the features of our application we had to primarily rely on the intended use and requirements set forth by our stakeholder. This resulted in an application which consisted of most of the fundamental characteristics with a few additional attributes which made the application more convenient to use. However, even though we did meet the minimum requirements, there were some secondary features – which we stated that we would implement – that we decided to abandon. The reason for us to discard some elements varied from feature to feature. Some of them turned out to be way more complicated than we initially thought, and relied on a greater understanding of subjects that we had no experience with. While other features simply turned out to be way more time consuming than we first anticipated, and we decided to use our time to prioritize different aspects of the application.

A. What Alphamail can do

- Login/register

We wanted our application to be accessible by different users, such that they have their own inbox to ensure privacy and convenience. Therefore, we have implemented the feature for users to register an account. We store the informations of the account in our database. With this, the same user can sign back in at a later time.

- Send/receive

We have implemented a way for a user to compose and send an email and, likewise, receive and view emails sent by other users. The motivation and thoughts behind this features is explored further in ”Documentation”[3].

- Send to multiple receivers

We implemented a way for a user to send an email to multiple users without having to compose a new one for each receiver. To do this, you simply separate the emails with either a comma or a comma followed by a space.

- Reply

Most of the time, when receiving an email, it is nice to be able to quickly respond to the sender. Therefore, we have implemented a ’*Reply*’ feature that fills out the necessary information for composing an email, such that this email will be sent back to the person whom the user wants to answer.

- Forward

Likewise, when receiving an email, it is common that a user would want to send the email to other users. The ’*Forward*’ functionality allows the user to essentially ’copy’ the received email with the ability to change only the recipient.

- Save draft

The draft feature allows the user to save a composed email without sending it. Thus, when an email is still ’work in progress’, but the user don’t wish to send but rather save the email for later edits, this is now possible.

- Password encryption

One of the final features we implemented was password encryption. In general, it is very unsecure to simply save passwords as text. Instead, we decided to implement encryption for an added security measure for our users.

- Sending to external sources

As previously mentioned, we made use of a NodeJS middleware function to communicate with external sources with SMTP, such as Gmail.

B. What Alphamail can not do

As previously mentioned, there were features we didn’t not manage to implement. We mentioned some, however, here is a list of every case of such.

- Sorting emails by folder

Initially, we had the idea that it would benefit the overall quality of Alphamail if it was able to

sort emails in different folder. First off, this would allow the user to sort emails, keep track of their important mails and setup folders for different purposes. Additionally, this would allow for us to more easily implement an algorithm to sort mails by junk, spam, advertising and so on, later in development. However, we did not get to implement this, as we had our priorities elsewhere. Furthermore, when reading back through our Requirements document, we realized we described 'archive email' wrongly. Archive email, instead, would be a 'default' folder to store mails in, but, as we didn't implement folders, we didn't implement archive emails.

- Automatic update of inbox

We stated in the Requirement documentation, that we would update the inbox once every timeframe. However, as we work on a webbased client, we found this unnecessary, as most browsers update whenever a user performs an action.

- Login attempts limit

Another function which, unfortunately, did not make it to the final version of Alphamail, was a limit to the amount of login attempts users had, before their account would be locked for a certain amount of time. This function would've helped against brute-force hacking, as hackers wouldn't be allowed to have a bot testing all possible passwords for a username. The reasons we didn't do this, is we simply couldn't figure out how to implement it. Specifically, we didn't know how to lock an account for a certain time frame.

- Trash folder

The trash folder was a folder with a unique property. It would temporarily store previously deleted emails, making it possible to restore emails deleted on accident. However, just like the locked account case, we could not figure out the time problem.

- Attachments

Arguably, one of the more important features that didn't make it to the final version, was the ability to attach files or pictures to an email and send it to another user. We didn't implement this, as we did not know how to store such file types in our database.

- External communication

As mentioned in the *decisions* section, we do not have our own domain. This means that external sources cannot send us information – the only mails we can receive are from other Alphamail users on the servers local network.

- Error handling

This final entrance to things that Alphamail can't do, is that it doesn't have error handling. This means, if there is an unforeseen error or bug, the Alphamail server will crash.

IV. Tests

Here we simply state which types of tests we made, rather than going into detail with results, changes etc. If you wish for this info, see appendix B. The functions we tested were:

- Testing while development/white box testing.
- General test 1: Drafts.
- General test 2: Drafts.
- General test: Register
- Limit testing: Login
- Limit testing: Register
- Limit testing: All functions with input for sending emails (send, reply, forward, etc.).
- Blackbox scenario testing: The cases described in our testing plan [9], for the login functionality.
- Blackbox scenario testing: The cases described in our testing plan, for the send/receive mail functionality.
- Blackbox scenario testing: The cases described in our testing plan, for the mark as read functionality.
- Blackbox scenario testing: The cases described in our testing plan, for the delete email functionality.

After some tweaking, all tests were passed, and from what we tested everything works as intended. The final testing we did for our application was carried out approaching the report deadline. Here we

met up again in order to test with multiple computers, in order to make sure that our implementation worked over LAN and not just locally. This test was done by having the server run on one of our computers and have other computers connect to the desired port on the server. We would then carry out some of the same tests as was done when the server and client was on the same computer, to confirm that everything worked as we intended.

V. Changes to previous reports

Throughout a work process, one will, almost, always end up deviating from the original plan – this was the case for us as well. We made some previous reports, where we stated our original idea of which functionalities Alphamail would have, how they would/wouldn't work and more.

Most of the changes are mentioned earlier, but here is a complete list of changes.
Changes to our 'Requirement document' [3] report:

- We briefly say that we include the opportunity to add email to folders - this is not the case.
- We do not mention which method we use for receiving emails. We use an IMAP-esque implmentation for this, as the emails are stored on the server only, and are not saved on the client side. With this, users can access their email from different systems with their login credentials.
- We mention we use SMTP to send emails - this is partly true, but only when we send to non-Alphamails. Internally, we simply save the mail to the database.

Changes to our 'Architecture and Design' [4] report:

- Our general architecture remains the same. However, something worth mentioning is that the client is running on a web browser, controlled by our server, which is what enables our architecture to be layered in the first place. This was not mentioned in the original document.
- General design remains the same, except for a few functions being either changed or removed, such as the ArchiveEmail function being removed.
- We stated that we would like to be able to select multiple emails – this is not something we ended up implementing, and thus changed. With this, we also didn't use the composite pattern.

Changes to our 'Formal verification'[10] report:

- This is not changing, rather making clear that delete we used is the simple one; when a delete is requested by the client, it is deleted permanently.
- Though the login we described in this document would have been very good to implement, we didn't manage to do so, for various reasons. Instead, you have an infinite amount of login attempts, until a correct username and password is entered.

Changes to our 'Testing plan' [9] report:

- Most of the report haven't changed. However, the load testing did change, as we chose not to containerize our system. As the mail ended up running on local networks only, we do not see a reason for scaling up for more, as we reduced the amount of users at a time significantly by only being local. Thus, we also didn't perform a load test.

These were the changes made to earlier reports.

VI. Discussion of results

Even though we had to find some alternative methods and make a few changes to our design and architecture along the way, the final product is actually quite close to how we pictured it in our minds. We always wanted to create Alphamail as a web-application with a UI, that allows the user to easily navigate our website, and we believe that we have achieved this goal.
We managed to implement most of the requirements discussed in the requirements document[3], e.g. send/receive, login and delete. However, we had a few bumps on the road, which required us to change our implementation ideas and methods

slightly. The most significant change we had to make, was regarding the send/receive functions, which we intended to implement via SMTP and POP3/IMAP respectively. We discussed previously what decisions we made to engineer some alternative solutions for this problem. However, we just want to emphasize that the solution for the receive method has more similarities to the IMAP method, rather than POP3. Our solution shares some of the ideas, such as keeping the emails on the server side, so that they only can be accessed via the website. Since our solution only allows us to receive emails from other Alphamail users, our solution is not completely adequate, in context of the email services we see today.

We know that the decision we made means that Alphamail is not a particularly adequate email-service in practise. However, we are still quite satisfied with the final product, since it contains most of the functionalities we wished to implement, thus we still believe that it meets most of the requirements that makes it an email-service. Where it might prove more useful, however, would be for internal use. An example of this could be in a organization, that doesn't want outside communication, but still a way to communicate internally over a network.

VII. Conclusion

In this project, we have developed an email-service software as a web-application. We have used some of the well established methodologies of software development to dictate our workflow and processes. In particular, we have been greatly inspired by SCRUM and spiral model in this regard. With these in mind, we worked incrementally on the Alphamail product. We specified the requirements and crafted a design, architecture and implementation plan. We produced multiple prototypes[1], arranged and conducted a wide range of tests in an effort to validate and verify our product, to the best of our ability. The process has been challenging and there was many things that we had to learn from scratch. However, the learning outcome has been very significant. We have learned much about software development processes, as well as many technicalities regarding communication and networking. After hours of determined work we, the Alphamail development team, are proud to release Alphamail 1.0.

[1]See the Github repository for more information.

REFERENCES

[1] The Alphamail team. *Github containing Alphamail*. URL: https://github.com/Biorrith/Software-Teknologi.git.

[2] LetsBuildThatApp. *NodeJS REST AP youtube playlist*. URL: https : / / www . youtube . com / watch ? v = F7NVpxxmmgM & list = PL0dzCUj1L5JE4w_OctDGyZOhML6OtJSqR & ab_channel = LetsBuildThatApp. (accessed: 20.12.2020).

[3] The Alphamail team. *Requirement document*.

[4] The Alphamail team. *Architecture and design report*.

[5] Leen Gui. *node-sendmail*. URL: https://www.npmjs.com/package/sendmail. (accessed: 20.12.2020).

[6] TEMPLATED. *Industrious: Responsive HTML5 template*. URL: https://templated.co/industrious. (accessed: 07.01.2021).

[7] Noor khan. *How to display Data from MySQL database table in Node.js*. URL: https://codingstatus.com/how-to-display-data-from-mysql-database-table-in-node-js/. (accessed: 27.12.2020).

[8] Amanda Silver. *Introducing Visual Studio Live Share*. URL: https://code.visualstudio.com/blogs/2017/11/15/live-share. (accessed: 20.12.2020).

[9] The Alphamail team. *Testing plan report*.

[10] The Alphamail team. *Formal verification report*.

APPENDIX A
HISTORY AND EVOLUTION OF ALPHAMAIL

Here we state which changes in our sprint for specific dates, starting around the 18th of December. The first few days, specifically 18-20th, went with us figuring out exactly how we were going to implement the system. We had a lot of discussion about the designs, about the compromises we had to make with our implementation, and splitting up to research different possibilities. We ended up sticking with our original plan, implementing in JavaScript, with a different goal. Instead of implementing a server that operates online, we only operated through LAN. With this, we started on the real development the 21'st of December.

Progress made on the 20/12-2020 and the 21/12-2020:

- Set up a server to connect to on the given port, specifically port 3000. Additionally, set up a connection to mySQL.
- Made a login and register functionality (without code encryption). The register function makes creates tables in mySQL, in which we plan to store the emails send/recieved.
- Made it possible to send an email, which called both 'saveMail_recieved' and 'saveMail_send' functions. These store the mail in the mySQL tables created in the register function.
- Made mark as read/unread, by storing a boolean value in mySQL. When switching between the two, simply invert the bool.
- Set up a basic cookie session. This blocks access for non-logged in users, and redirects them to the '/' address. Additionally, we use the session to keep track of the logged in user, and give them the proper emails.
- Set up .ejs files for the following addresses: '/', '/homepage', '/send', '/inbox', '/outbox', '/deletein/:id', '/deleteout/:id', '/viewin/:id', '/viewout/:id', '/markasread/:id', '/markasunread/:id', '/register' and '/login'.

Progress made on the 21/12-2020:

- Made it possible to send to multiple emails at a time, by chopping up the string.
- Set up the posibily of making a draft. To do so, we made a draft table in mySQL, which is created in the register.
- Made a .ejs file for the following addresses: '/drafts', '/deletedraft/:id', /viewdraft:id/ and '/savedraft'.
- Improvements upon existing .ejs files.

Progress made on the 22/12-2020 and 23/12-2020:

- General cleanup in the code.
- Implemented CCS style to make the mail better looking. Implemented pictures.
- Added options to reply and forward emails.
- General testing and correction of code. As an example, you now can't register the same user twice, and simply get redirected to the register address if you try to.

We took the day off 24-12/2020. Progress made on the 25-12-2020:

- Made a installation guide/README.txt file.
- General code cleanup

This marked the end of our sprint, and we stepped away from the project for a while. We came back to test on the 13-01/2021. For info, see appendix B.

Finally, on the 21-01/2021, we made one last implementation; encryption of the password. We hash it

with sha256, and store the hashed code, instead of the normal code. When we log in, we hash the input and compare it to the saved hash.

APPENDIX B
TESTING OF ALPHAMAIL

The following is a list of all the test we made, with an in depth explanation of results, changes and when they were made.

Testing while developing in our sprint:

- **Test made:** We, of course, tested our program while making it, which we did not document thoroughly, as we would fill literal pages with bugfixes and tests done. This is also an excuse for us not thinking about the documentation while developing, which is very stupid of us not to. Additionally, the white box testing of this program was done during the development, testing for dead code, states etc.
- **Results:** Loads of results, not many documented - possible changes may be seen in appendix A.
- **What changed:** Just about every function, leading to our final product.
- **When:** Changes made between 19-12/2021 and 26-12/2021.

General test - drafts:

- **Test made:** Saving an existing draft when viewing it.
- **Results:** A new draft gets saved, rather than update the draft we're viewing. This, for us, is unpreferable, which made us fix it.
- **What changed:** Made a 'savedraft_view' function, which updates the existing draft based on its ID, rather than save a new draft.
- **When:** Changes made 13/01-2021.

General test - drafts:

- **Test made:** When you view a specific draft, what happens when you send it?
- **Results:** When you send the draft, it does not get deleted from drafts.
- **What changed:** Made a 'senddraft' function, made a 'sendMail' function containing what previously was in the 'save' post call. Now both 'send' and 'sendraft' call the 'sendMail' function, and 'senddraft' deleted the draft afterwards.
- **When:** Changes made 13/01-2021.

General test - Register:

- **Test made:** Special characters in the email.
- **Results:** Since we're making new SQL tables, and those dont allow special characters in their name, this would result in a crash.
- **What changed:** To fix this, we made it so you can only input letters, numbers and '.' as username when registering.
- **When:** Changes made 13/01-2021.

Limit testing - login:

- **Test made:** In SQL we set a limit for the length of the username (255 chars). What happens when you exceed the limit? Important to notice, limit testing for password are not performed, as they are always transformed to 64 digits.

- **Results:** When the size exceeds the limit, the program crashes. With no input, nothing happens, as HTML makes input required to proceed.
- **What changed:** Set a max limit on the character inputs in the username/email (255).
- **When:** Changes made 13/01-2021.

Blackbox scenario testing - login functionality:

- **Test made:** We tested the 5 cases described in our Test Plan.
- **Results for the cases:**
  1) Correct username and password gives access. Test passed.
  2) A wrong password simply redirects you to the login page again, without logging you in. Test passed.
  3) A wrong username simply redirects you to the login page again, without logging you in. Test passed.
  4) A wrong password and username simply redirects you to the login page again, without logging you in. Test passed.
  5) With the limit testing, we made this scenario doens't exists no. Tests were passed, no changes made. Test passed.
- **When:** Tested 13-01/2021.

Blackbox scenario testing - login functionality:

- **Test made:** We tested the 6 cases described in our Test Plan.
- **Results for the cases:**
  1) Mail send successfully, test passed.
  2) The program does not crash nor send an email - however the mail system does not give an error to the user. Test passed.
  3) You can't send without specifying a recipient, test passed.
  4) As long as you separate the users/emails by either ',' or ', ' this works. If multiple spaces, the program will see the rest of the string as one big email, thus not sending properly (see case 2). Additionally, the mail will not appear in the outbox - this was an easy fix, however, and works properly now. Test passed.
  5) We decided not to include files, making this case unreachable.
  6) Mails can be seen in the inbox as intended. Test passed.
- **When:** Tested 13-01/2021.

Blackbox scenario testing - delete mail functionality:

- **Test made:** We test the 2 cases described in our Test Plan.
- **Results for case 1 and 2:** These two cases merged into one, as we did not implement the feature to allow restoring a deleted email for 30 days. Instead, when you delete, you delete permanently. However, this works as intended.
- **When:** Tested 13-01/2021.

Stress/load and performance - testing with containers (Docker):

- **Test made:** We didnt actually do this testing, as we decided against using Docker in the end. Without this, we were not sure how to stress test.
- **Results:** N/A.
- **What changed:** N/A.
- **When:** N/A.