University of Tartu
Institute of Computer Science
Computer Sciences curricula

Markus Kikkatalo

The migration of an administrative application's user interface from Thymeleaf to Angular

Bachelor's thesis (9 EAP)

Supervisor: Vimal Kumar Dwivedi, PhD

Tartu 2023

The migration of an administrative application's user interface from Thymeleaf to Angular

Abstract: This thesis explores the process of migrating a legacy system to a modern architecture using Angular and Spring Boot frameworks, with the aim of enhancing maintainability, scalability, and developer experience. The migration involved analysing the previous system, refactoring the code, and employing the Cypress framework to ensure quality testing. Although the migration successfully improved code quality and developer experience, it resulted in longer initial load times for the user interface due to the complexity of the Angular application. In conclusion, this paper serves as a successful case study of migrating legacy systems to modern architectures and highlights both the challenges and benefits of such an endeavour. The results demonstrate that contemporary frameworks like Angular, Cypress and Spring Boot can significantly improve legacy systems' quality and maintainability, but careful planning during the development process is essential.

Keywords: Front end migration, Angular, Thymeleaf, Spring Boot

CERCS: P175 Informatics, systems theory

Administreerimiskeskkonna migratsioon Thymeleaf raamistikult Angulari veebiraamistikule

Kokkuvõte: Selle bakalaureusetöö eesmärgiks on pärandsüsteemi migreerimine kaasaegsele arhitektuurile, kasutades Angular ja Spring Boot raamistikke. Migratsiooni eesmärk oli parandada süsteemi hooldatavust, skaleeritavust ja arendaja kogemust. Tehnoloogia üleviimise protsessideks oli olemasoleva süsteemi analüüsimine, koodi ümbertöötamine ja süsteemi testimine selle kvaliteedi tagamiseks. Eessüsteemis võeti kasutusele uus kasutajaliidese testimise raamistik Cypress, mis parandas testimisprotsesse ning migratsioon oli koodi kvaliteedi ja arendaja kogemuse osas edukas. Siiski põhjustas migratsioon uue eesliidese raamistiku suuruse tõttu mõne jõudlusnäitaja halvenemist nagu esmase lehe laadimine ning esimese sisu kuvamine. Tegu on eduka juhtumiuuringuga pärandsüsteemide üleminekust kaasaegsetele arhitektuuridele, mis annab ülevaate sellise migratsiooni väljakutsetest ja eelistest. Tulemused näitavad, et kaasaegsed raamistikud, nagu Angular, Cypress ja Spring Boot võivad oluliselt parandada pärandsüsteemide kvaliteeti ja hooldatavust, kuid nõuavad hoolikat planeerimist süsteemi arendamise ajal.

Võtmesõnad: Eessüsteemi migreerimine, Angular, Thymeleaf, Spring Boot

CERCS: P175 Informaatika, süsteemiteooria

Table of Contents

1. Introduction

Due to the growing demand for highly interactive and user-friendly administrative programs, organisations have recently been pushed to reconsider their technical options, particularly concerning front-end frameworks. Due to the increasing complexity and demands placed on these applications, it is now being investigated whether it is possible to convert existing applications using server-side rendering frameworks like Thymeleaf to Angular and other modern client-side rendering frameworks. The main goal of this thesis is to examine in-depth the difficulties and complexities involved in such a migration process, highlighting the benefits, disadvantages, and potential effects on the creation of administrative applications and user experience. The key contributions to this work are:

- An analysis of the current system and potential solutions.
- A migration of the legacy application.
- Performing quality assurance on the migrated application.

During the migration, the system is decoupled, where the front-end and back-end components are created, maintained, and deployed separately after switching from Thymeleaf to Angular. Numerous advantages come with this decoupling, including better maintainability, scalability, and the potential for parallel development. The migrating process is not without difficulties, as it necessitates a complete understanding of Thymeleaf and Angular and the organisational and technological aspects that can affect the migration's success.

UI testing, which guarantees that the program works as planned and offers a seamless user experience, is a crucial component of the development of any online application. The effects of switching from server-side to client-side frameworks on UI testing methodologies must be considered as part of the migration process. This thesis will examine several testing tools, contrasting the strengths and weaknesses of well-known testing libraries like Cypress for Angular applications and Selenide for Thymeleaf applications. The study will also look at how migration can affect performance and user experience, highlighting the advantages and disadvantages of each framework.

With a focus on the difficulties, advantages, and potential effects of adopting a decoupled system, this thesis intends to give a thorough and informative examination of the migration process from Thymeleaf to Angular in the context of administrative applications. The thesis

looks into various migration-related topics, such as UI testing, performance, and user experience, to develop a thorough understanding of the elements that can lead to a successful migration.

2. Background

This paragraph will explain and define the thesis's related terminology, concepts and technology. It will explain the differences between server-side-rendered (SSR) and single-page applications (SPA).

2.1 Payment acquiring

Payment acquiring (ACQ) solution offers companies to quickly and securely complete transactions in physical and virtual stores. ACQ checks the payment confirmation through automatic communication with the bank that the payee originates from [1].

2.2 ACQ Admin application

The application has a Java Spring Boot version 2.7.10 as a back end solution with data management done through the MSQL database. The application is over nine years old and lately has seen few improvements and mostly bug fixes. The application follows the Model-View-Controller principle.

The current user interface is done using a server-side-rendering framework Thymeleaf. Since the amount of data that needs to be processed through the system comes from different countries, banks and issuers who can have different requirements for their transactions and claims, some of the settlements have to be completed manually. Furthermore, registering and configuring new merchants and their branches with new payment options is also done in the application. In addition, the invoices are generated and sent out through the administrative system. Such an application is necessary for cases when automatic processes might fail due to faulty or missing data and provide an easy option for the bank operations team to fix the issues quickly.

2.3. Model-View-Controller

Model-View-Controller (MVC) is a software design pattern that separates an application into three interconnected components: the model, the view, and the controller. The model

represents the data and the business logic, the view represents the user interface, and the controller acts as an intermediary between the two. This separation of concerns makes the code easier to maintain and extend, as changes to one component do not affect the others [2].

2.4. Model-View-ViewModel

The Model-View-ViewModel (MVVM) is an extension of the MVC pattern. It separates the user interface from the business logic. The model represents the data and the business logic, the view represents the user interface and the view-model acts as an intermediary between the view and the model. The view model also handles user interactions and updates the model and view accordingly. The latter also exposes data and commands that the view can bind to, which allows for easy updating of the view [3].

2.5. Spring Boot

Spring Boot is an open-source framework for building Java-based web applications. It is built on top of the Spring framework and designed to simplify the development process by configuring the application based on the dependencies and libraries included in the project. It can automatically configure the application based on the dependencies and libraries included in the project [4].

2.6. Thymeleaf

Thymeleaf is a server-side Java template engine for web and standalone applications [5]. It allows developers to create dynamic web pages using natural templates. It has a powerful expression language, supports creating HTML, XML and plain text templates and provides integration with Spring Boot.

2.7. RESTful API

REpresentational State Transfer (REST) application programming interface (API) is a set of rules that define how applications can connect to and communicate with each other [6]. REST is founded on six key constraints: client-server architecture, statelessness, cacheability, layered system, code on demand and uniform interface. These constraints provide a structured approach to the design of web services that enables them to be simple, modular and highly scalable.

2.8. Angular

Angular is a front-end JavaScript framework developed by Google, which is used to build complex and dynamic web applications [7]. It uses a component-based architecture combining MVVM, dependency injection and observables design patterns. Its powerful template engine and modular architecture enable developers to build complex and scalable applications. Angular is written in TypeScript, which ensures type safety when developing the application (Google, https://angular.io/guide/what-is-angular 2023). It focuses on performance, and using a change detection mechanism and ahead-of-time compiler leads to faster loading times and improved user experience. It must be noted that compared to the other most used web frameworks like Vue.js and React, Angular has a steeper learning curve and a larger codebase, leading to a slower initial load time for the application [8].

2.9. Server-side rendering

Server-side rendering (SSR) refers to generating HTML on the server side, sending it to the client, and then adding interactivity to the page with JavaScript. Its benefits are that SSR improves the performance of web applications by reducing the amount of time required to render the initial page, which can be especially noticeable on slower devices.

2.10. Client-side rendering

Client-side rendering (CSR) is a web development approach where the HTML, CSS and JavaScript are loaded and executed on the client side. It provides faster navigation within the application, better performance when dealing with large datasets and a better user experience with smoother transitions and fewer page reloads. When the client requests data from the server, a compact response is received, consisting of the data rendered to the page.

2.11. Decoupled system

A decoupled system is an architectural approach where an application's front-end and back-end components are developed and maintained independently. In such systems, the parts are not directly dependent on one another [9]. They mostly communicate over an API, and the technologies are easily interchangeable if the communication constraints remain.

2.12. Selenide

Selenide is a concise and powerful open-source framework designed for simplifying the testing of web applications through automation [10]. It is powered by the Selenium WebDriver that provides a straightforward API, enabling developers to write tests with less boilerplate code.

2.13. Cypress

Cypress is a JavaScript-based testing framework that provides a fast, reliable and efficient way of testing web applications by running tests in a real browser [11]. It provides real-time feedback during testing, ensuring quick problem identification and reducing debugging time. Cypress has an intuitive API and built-in features like automatic waiting and testing across multiple browsers and devices. Cypress requires a fair amount of setup and cannot be used for mobile or desktop application testing.

2.14. Migration in software development

In software development, migration refers to the process of transferring a system from one environment or technology to another. The reasons may vary from technology updates to better performance and security or compliance requirements. Migrating from a legacy system can improve scalability and reduce maintenance costs, as some older technologies may not be widely used anymore.

Regarding this thesis, the number of active users for Thymeleaf users who have starred the repository on Github is 2531 [12]. For Angular users, the number of stargazers is more than 87 thousand [13]. For the average amount of downloads for a framework, Thymeleaf has around 2 million downloads per month [14], whereas Angular has around 12 million during the same period [15]. The results for job listings also differ drastically- in LinkedIn, 0 results show up when the keyword "Thymeleaf" [16] is inserted for jobs in Estonia. However, 56 job results were found for Angular [17].

3. Related work

This section introduces and analyses similar migration tasks and completed admin interface applications. To find related works, theses with the keywords “Angular”, “Thymeleaf” and “migration” were used and a search on what technologies other domains at LHV use for their applications was conducted. The following were the most relatable.

3.1 Crosskey.io migration

In 2022, Jennie Eriksson presented her work on a similar migration task of moving from Thymeleaf to Angular application. [18]. The author describes the processes and technologies used during the migration and how different development parts were done. It is a good study case on how to create an Angular application and how to implement technologies like Swagger and REST API to it, but the migration aspect could have had better coverage. The only comparison between the legacy and migrated versions was when, after the first few pages, Google Chrome's Lighthouse tool was used to test the performance and accessibility of the developed pages. It was noted that the initial load time dropped significantly, but Eriksson managed to improve the performance by 20 per cent by compressing the files about to be bundled.

3.2 LHV Customer Admin Application

The Customer domain is responsible for storing and gathering the correct information about LHV's customers. Their administrative application uses Angular for its front-end framework, and their pages are mostly simple search forms and results lists, see Figure 1.

The Customer's front-end application was used as a template to create the initial application for the Acquiring domain as the layouts should be the same cross-domain, and they had many components that were taken to use after minor modifications.

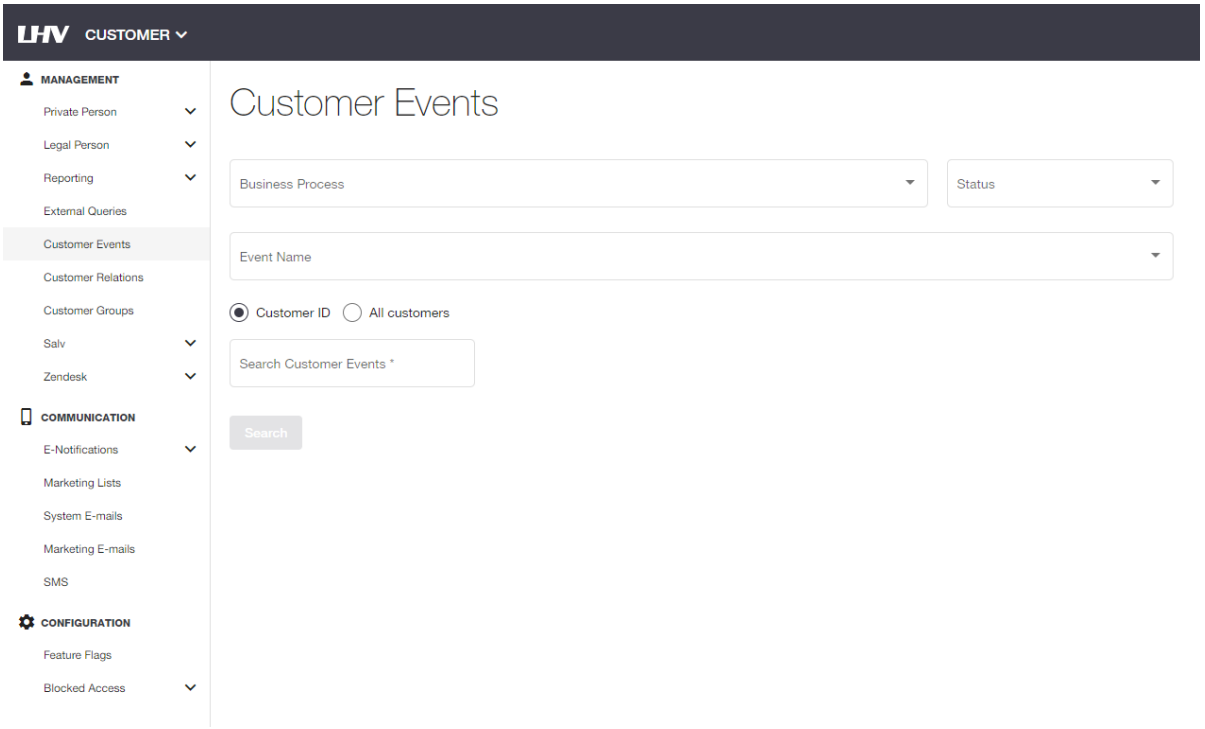

Figure 1. LHV Customer administrative user interface.

3.3 LHV Investments Admin Application

The Investments domain handles customers' actions when making investment decisions. Their in-house application uses Angular as its front-end framework, and they have overview pages to analyse different financial assets and complex dynamic forms, see Figure 2. They are developing the next user interface design that would look more similar to LHV's public website. Previously they also had the same style as Customer Admin.

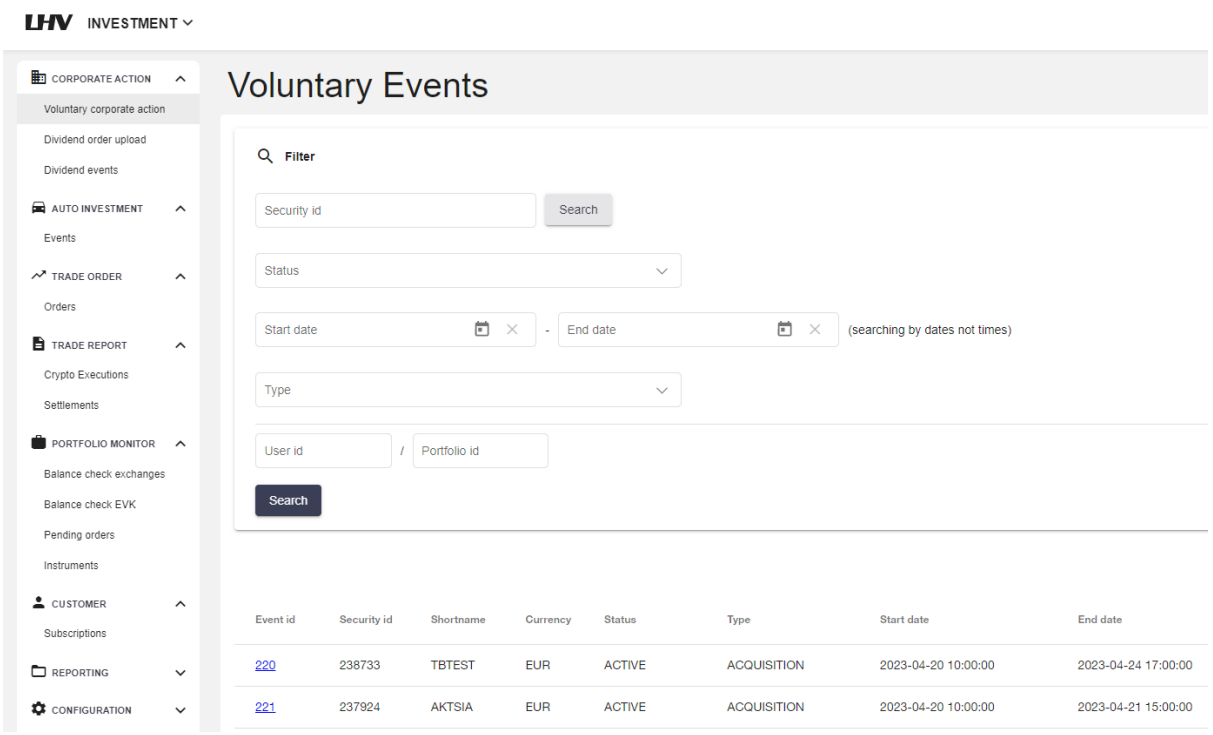

Figure 2. LHV Investment administrative applications' user interface.

4. Methodology

Before starting the migration, core problems of the current server-side rendered application must be detected and solutions from the client-side rendered application must be found for the migration to be meaningful. When the issues and possibilities for both systems have been mapped out, tasks to migrate the system will be created and the development will be started.

The development process will be separated into tasks documented in JIRA, where each task is a separate page. After the initial version of a page is complete, the front-end and back-end code will be reviewed by the author's team members. The code will be improved based on the suggested changes and comments discovered in the review. This process will continue until the reviewer(s) are satisfied with the code and approve it for testing. Then quality assurance (QA) specialists will ensure that the created change did not cause any bugs and that the functionality and safety of the application are at least as good as before. The QAs will add the missing end-to-end tests created in the Cypress framework to ensure the application's client and server sides work harmoniously. The development process is displayed on Figure 3.

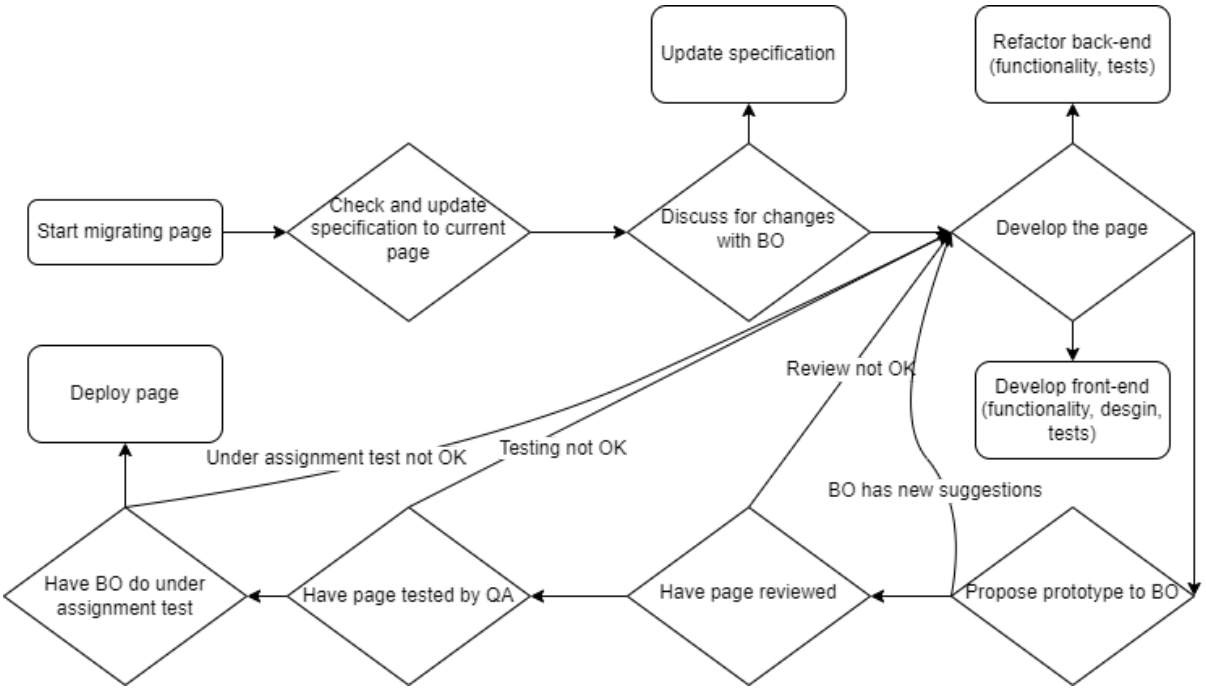

Figure 3. Diagram of the development process to migrate a page.

The code for the back-end application will be written in Java 11 using the Spring Boot 2.7.10 framework. The front-end application will be written in TypeScript using the Angular 14 framework in the beginning and version 15 later. Both sides of the applications use Gradle to build the files for deployment. The IDEs used for the development are the suggested editors

at LHV: IntelliJ for the back-end environment and Microsoft Visual Studio Code for the front-end environment.

4.1. Application requirements

Listed below are the pages that must use Angular UI by the end of the thesis:

1. The management page, where key attributes and processes can be altered.
2. The mass notification page, where key messages can be sent to all the customers in Estonian, English and Russian.
3. The files page, where essential information about the files sent to the application can be seen and the files can be downloaded.
4. The refund page, where refunded transactions can be viewed based on the described filters.
5. The claims search page, where customers' claims can be viewed based on the applied form.

The application must have the following functional requirements:

1. Redirecting to pages that still need to be migrated will open the legacy applications page.
2. Forms in the pages must be validated right after input is inserted.
3. Invalid forms must not be possible to be submitted.
4. A submitted form must receive its response before it can be submitted again.

The application must have the following non-functional requirements:

1. When the session has expired or the user is redirected to the legacy application, they must be authenticated.
2. Unauthenticated users must be redirected to the company’s SSO login page.
3. Users of the application must be authorised before navigating to any page to provide them access only to content meant for their role.

5. Analysis of the systems

In the following chapter, an analysis of the legacy system will be conducted to find the most significant issues regarding the application. We will be looking for answers to the following questions:

- What problems do Thymeleaf and other server-side rendered applications have?
- What is the impact of the problems?
- What options are there to replace the Thymeleaf framework?
- What alternatives has the company considered?
- How does an Angular application solve these issues?
- What are the validation processes to ensure that the problems have been solved?

After these questions are answered, the migration can be started purposefully.

5.1. Legacy system

In this section, we will discuss the positive and negative things of the current Thymeleaf system and address the issues that affect the company, the users of the application and the development team.

As the used technology is outside the company's strategy roadmap, the look differs from most other domains, and there is no in-house space to share experiences with other developers. As the company's main page is developed in Angular and the heads of the department require that the administrative user interfaces also be in Angular, maintaining a Thymeleaf application is not supported. A couple of other domains are also ongoing the migration from Thymeleaf to Angular. When all the company's domains use the same front-end technology, setting enterprise-wide coding practices and style guides for the UI is easier. With these documents, the web applications will have a similar look. For the company, it is also easier to recruit new personnel to the development teams when the technology stack is more similar across the teams. Maintaining a legacy application that uses Thymeleaf makes it harder to spread knowledge on the frameworks used and conflicts with the company's desire to use SPA applications.

The application users have had it since their first day in the company and are familiar with the system, but they have proposed to have a more interactive and better-looking application for years. As mentioned in the introduction, the ACQ Admin application has been in

development for almost ten years, and only a single worker in the department has been working in the company for longer than this system has existed. Therefore, the workers know where different details are located and what inputs are needed for their desired results. However, many issues have been raised over the years, some due to development mistakes or technology problems. Some of the bugs created are that when searching for some items or generating multiple large invoices, the data retrieved causes the back-end application to crash. These can be avoided by setting limitations to the requested amounts or disabling some functions that are already ongoing.

The users have also raised concerns about form validation and form interactivity. As many of the applications' pages are based on a search form followed by a displayed elements list, the actions done to create a successful query are critical for an efficient workflow. With server-side rendering, the user has to submit all the data to the form and submit it to the server, where they can receive a message that the input is in incorrect format or that a field was required. Furthermore, as the bank operations team workers often need to find specific information for a particular customer, then the legacy application, most of the forms use different IDs to create the query, making it an extra step to first find the ID of the customer and then complete the search. The workers have raised an issue where they would like to get the same results but could get the necessary information from a dropdown list that their input would filter. Such dynamic change can not be done using only Thymeleaf and the development team has no knowledge and resources to provide solutions to these issues.

To conclude, the users of the application desire they could do their work seamlessly in a user-friendly environment that points out their errors as soon as possible and prevent them from doing actions that could be critical to the uptime of the application.

Last but not least, the development team wants to create modern and performant applications requiring minimal maintenance and customer support to develop new domain features. The development team responsible for the payment-acquiring domain does not have a designated front-end developer. So far, the contributions and desire to make the user interface more customer-friendly have yet to be on a high-priority list. As the team mainly works with APIs, they are unfamiliar with the current trends and best front-end development practices. They want that the development cycle would be as seamless for them, meaning that there would be extensive documentation and resources on the framework and subject and that they have

access to other views and components previously created so they could modify them to their need and complete the initial testing of the added features with tools that are intuitive and fast. With Thymeleaf, the ecosystem to gather information is relatively small compared to other front-end frameworks. Testing the UI with the Selenide framework could be faster and more stable. It leads to long deployment times, where around 50% of the time is spent on UI testing, and frequently the pipelines stop as the tests fail to load a page before a timeout error is called with the Selenide WebDriver.

5.2. Alternatives to Thymeleaf

Over the years, the popularity of JavaScript and its frameworks [19, 20] has steadily risen, and more web applications are created using different front-end frameworks, mostly in React[1], Angular and Vue.js[2] [21]. The ones listed before are all client-side rendered out of the box but paired with back-end frameworks like Next.js[3] or libraries like Angular Universal[4], these frameworks can also work in server-side rendered applications.

For many of the popular programming languages, such as Java and C#, there exist several SSR front-end frameworks for each language- for Java, there is Thymeleaf and Freemarker, and C# has Razor[5]. However, their constraint on the interactivity in the browser and their programming language dependence can make them less attractive in general.

LHV wanted to avoid enterprise products and use modern and trendy technologies. LHVs' oldest administrative application, Core Admin, is built using Adobe ColdFusion[6]- an enterprise solution for creating web applications. At the beginning of the 2010s, migrating the Internet bank and administrative applications to newer technologies began. In the following years, multiple technologies like Velocity[7], AngularJS[8] and Dart[9] were considered and implemented for a short while. In 2016, when Angular version 2 was released [22], the company developed a proof-of-concept application using Angular version 2. Satisfied with

[1] React website - https://react.dev/
[2] Vue website - https://vuejs.org/
[3] Next.js website - https://nextjs.org/
[4] Angular Universal guide - https://angular.io/guide/universal
[5] Introduction to Razor - https://learn.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-7.0&tabs=visual-studio
[6] Adobe Coldfusion website - https://www.adobe.com/products/coldfusion-enterprise.html
[7] Apache Velocity webpage - https://velocity.apache.org/
[8] AngularJS webpage - https://angularjs.org/
[9] Dart webpage - https://dart.dev/

the result, they started migrating the internet bank from ColdFusion to Angular. However, as the company's growth was immense, in-house applications were left as they were, primarily in ColdFusion or Thymeleaf.

After many years of developing the internet bank in the Angular framework, the company has established a styled components collection from which the development teams can retrieve reusable components with what the whole internet bank application is built. It provides a unison appearance and expected functionality cross-pages and domains. Creating a similar collection is believed to be started soon for administrative applications.

In conclusion, several technologies could replace Thymeleaf, but ultimately, the knowledge, code base, and experience already present in the business must be considered when deciding which technology to migrate. The Angular framework was chosen to make the transfer in light of this.

5.3. Migrated system

This section will describe an overview of the migrated application and how it solves the negative aspects of the legacy system pointed out in the previous section. We will also note some of the drawbacks and potential effects that the decoupling could have on the system.

To begin with, the front and back end will be split into separate projects. When a decoupled system is set up, the build times of the front-end application will not affect building the back end. As the introduction notes, the user interface has had a few improvements over the last couple of years. Most of the work is held in the back end, where features are added to communicate with our partners through API calls. For the server to communicate with the user's browser, the communication has to be transferred to conduct RESTful API calls using JSON files. REST API constraints will make the data transfer between the client smaller. Integrating a new front-end technology into the system in the future will be easier, as back-end communication will be defined in a more versatile way than before. Having a decoupled system makes it possible for the development team to achieve faster results when deploying to the respective side of the application.

In addition, the user's experience will be more fluent with a SPA, as they will receive faster feedback on errors in their workflow. With the Angular framework, users' actions will be instantaneously checked due to what inputs written to the pages' forms will be validated, and the workers will have feedback on their inputs' correctness. Furthermore, the users can benefit from Angular's state management. When filling out a search form and visiting a detailed view of a result object, they can return to the previous page with the previous search results. Lastly, the website will receive an updated appearance, making it welcoming and more similar to other administrative applications in the company.

However, migrating to and serving an Angular application for the front end has some side effects. To begin with, the migration can be quite extensive, as previously, the back end was tightly coupled with the Thymeleaf framework with the MVC pattern. Initially, there will be a learning curve for the development team as they will need to learn new coding best practices for new technologies so that the development times can be longer in the beginning compared to the current situation.

A clear drawback regarding performance is the increase in the initial page load of SPAs, which affects the user experience. When the customer opens the website, all static content will be loaded into the browser before the view is shown. It can be optimised with lazy loading, where the components will be rendered only when they are supposed to be displayed on the page, but it will take longer to open than a server-side rendered application does, where the HTML is sent from the server. However, with the server-side application, the following pages will also take the same time to load, whereas SPA view shifts will be quicker as they are already loaded into the browser.

In conclusion, although the Angular application has some drawbacks and side-effects, most can be conquered with gained experience and precautious planning. Considering the framework's upsides, such as interactive forms and an updated appearance, it is reasonable to begin the migration.

5.4. Validation of the migration

In the following paragraphs, different options to validate and test that the migration was successful. The quality assurance methods will cover the application in performance, user experience and functionality.

5.4.1. Automated tests

Automated testing helps developers to catch bugs early in the development process, reducing the time and effort spent on manual testing. Automated tests are usually divided between different testing levels, where unit tests cover a component's functionality, integration tests for the components' interaction with each other and end-to-end tests check the system's communication with its dependencies. With the latter, there is a subsection with UI testing, where the front-end applications functionality is put under test. Having a testing strategy that validates the system on different layers, the code quality can be effectively reviewed by running the tests.

However, they do not replace manual tests but rather help to revalidate units under test. As Rudolf K. Keller et al. has pointed out [23], automated tests often cover only the problems discovered before, and with new features, unexpected defects may occur.

5.4.2. Performance analysis

Conducting a performance analysis of a web application is necessary to give customers a better user experience. Modern users want to use fast and responsive web applications and to ensure that their system is performant enough for the end user, they can look at some metrics. As web browsers may vary in their technologies, it is important to know the end users' most used browsers and measure each browser independently. Most of the customers of this system use Chromium-based browsers like Google Chrome and Microsoft Edge or Mozilla Firefox with a Quantum browser engine. In the company, it is forbidden to use Internet Explorer due to safety reasons. Some tools to measure the performance of a web application are built into the browser. Google Chrome provides tools like Lighthouse[10] and Performance Insights[11] which provides multiple useful metrics like Time to Interactive to get the amount of time it took for the page to become fully interactive, or First Contentful Paint, which marks the time the first text or image is painted. Using metrics in that tool, we can compare the legacy

[10] https://developer.chrome.com/docs/lighthouse/overview/
[11] https://developer.chrome.com/docs/devtools/performance-insights/

system with the migrated system, and we should get an overview of how performance affects the user experience.

5.4.3. Communication with the end user

The changes and additions must be communicated with the application's users throughout the migration process. In general, A/B testing[12] and feedback surveys are held with a smaller user base, so the added features would meet the customers' requirements. This system's only users are in-house workers with whom every business logic aspect can be discussed continuously. As a result of the migration, the application's appearance has changed, and it is easier for the customer to be in close touch with the development process, so they have a say in aspects that they would like to do some fine-tuning. Reordering navigation menu items or changing the way a value is displayed, in the end, is done so the users would be more efficient in their work.

5.4.4. Build time comparison

It takes various tasks to complete the migrated and legacy back-ends deployment pipeline, as it entails creating a jar file, testing it on various testing levels, and deploying the system to various environments. Although some external elements, such as the resources available and the network connection, may influence the outcomes, it should be possible to build an overview of the change across many pipelines.

Due to the added complexity of decoupling the system, the front-end component's build time will also be added to the comparison. However, as it is an individual application, the flexibility and the possibility that either of the projects may not need to be built in a pipeline when changes are only made in the other project.

[12] Techopedia definition on A/B testing - https://www.techopedia.com/definition/27398/ab-testing

6. Application Development

6.1 Preparation

Before the development of the client-side application could be started, the tasks needed to be done related to the migration of the system and a separate Git project, where the new Angular application would reside. The tasks were created in JIRA, and the initial number was 23. It consisted of creating the application, setting up the environments so that the communication between the applications would work and refactoring each page.

To create the project, LHV's development operations team members created the project in GitLab. They helped the author create the necessary pipelines and set up the application's deployment parts. The application was created using another in-house administrative UI application as a template. All the functionalities were removed, but the core layout, OpenId Connect authentication, authorisation and shared components that should be used in all of the company's Angular Admin UIs were kept and personalised.

Because LHV is a vital service provider [24], the systems must be safe, and the processes to set up these systems may take time. It includes setting up the physical servers and allowing different machines to communicate with each other on specific routes. As the migration of an in-house administrative application is not of the highest priority, to speed up the development process, the application will initially be published to a private library, from where it can be implemented to the back end using build tools such as Gradle. Later on, the application will be run on a separate machine, but this decision will not affect the progress of this thesis.

As a legacy server-side user interface application, ACQ Admin uses CAS to authenticate the users. However, as this method will be deprecated in the company, a new authentication method, OpenId Connect, had to be implemented. However, CAS and OpenId had to work simultaneously to have the legacy application
work simultaneously with the new, modern application. Hence, the back-end security configuration had to be modified to identify the front-end requests and convert the OpenId Connect user to a CAS user, adding special roles necessary for the CAS ticket to work correctly. The client is authenticated and authorised in every view change in the front end. If the user has not been authenticated or the session has expired, they will be redirected to the

AdminSSO login page if the authentication token has expired. To redirect to the pages that have yet to be migrated, a redirect guard was implemented to route the client to the pages rendered by the server.

For each page, specific privileges had to be configured so that people with adequate authorisation could access the pages. These privileges had to be set on both sides of the application to ensure safety. If the user tries to navigate to a page they do not have privileges to access, an access denied page with an error message will be displayed, as displayed in Figure 4.

Figure 4. An error message that is displayed when required authorisation level is missing.

6.2 Front-end development

6.2.1 Application file structure

The front-end project, as seen in Figure 5, is set up, so it would be effortless to make it into a standalone application. The Angular project is in the *frontend* folder, where the configurations of the project and the tools used are. The *cypress* folder contains e2e and UI tests created in the Cypress framework. Tests are separated into pages and page groups to resemble the Angular application's structure.

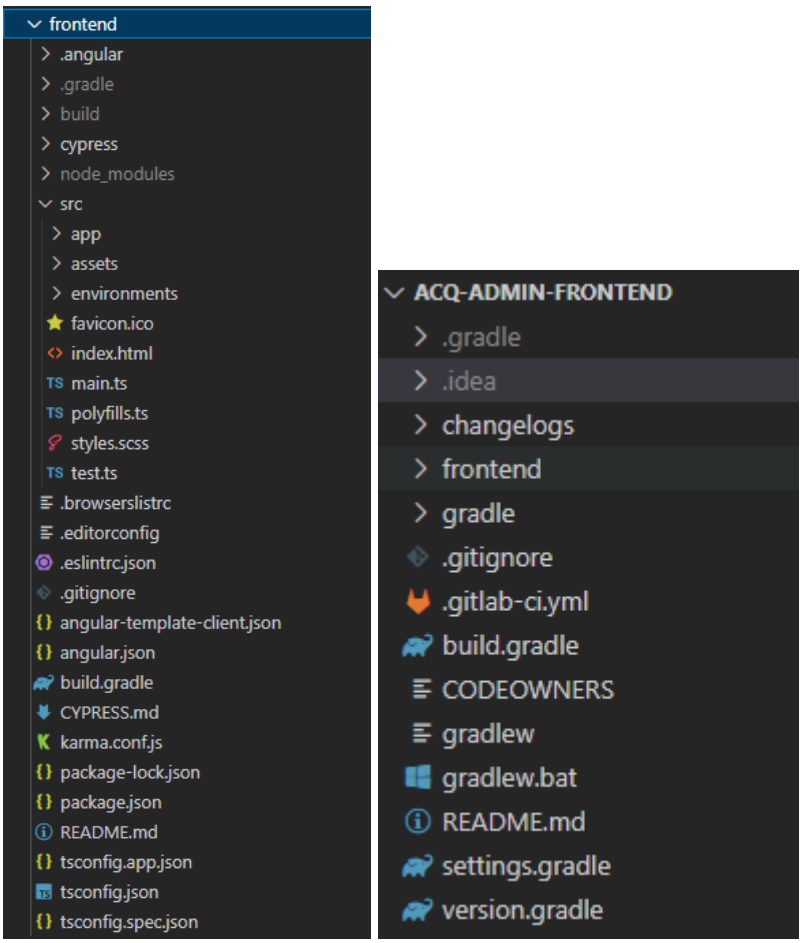

Figure 5. Application and Angular project file structure.

The *app* folder, what is displayed in Figure 6, is the application's root folder, where the root components are - *app.module.ts*, *app-routing.module.ts*, *app.component.html* and *app.component.ts* [25]. App.module.ts is used to bootstrap the application on launch. It specifies what components are declared over the application, what modules are imported to be used within the application, the service providers and what component to use as the root component.

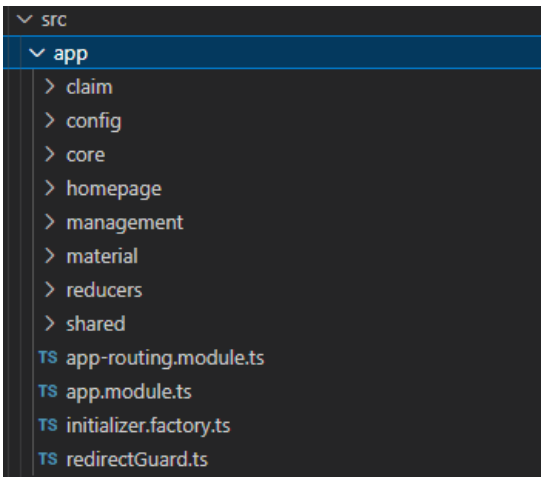

Figure 6. Angular root folder structure.

The shared folder consists of components that can be used across the application. Due to Angular's component-based architecture, the components need to be developed once and later on, they can be reused when they are declared in a module and then initialised in the new component's html with *<app-COMPONENT-NAME />* tag, and as the configurations can be added within the tag, reusable components can be pretty functional.

Other folders are feature group directories consisting of page view component modules. Feature group directories have their module and routing module, where the desired dependencies and paths are added.

6.2.2 Module and component folder structure

Each page view also has a *module.ts* and *routing-module.ts* file, where the final imports and declarations are defined. A simple component, which does not need state management, consists of a Typescript class, an HTML template and an SCSS file. The Typescript class defines the interaction of the HTML template and the rendered DOM structure, while the style sheet describes its appearance [26].

A component that has to alter data changes and keep track of user actions needs state management, and reducers are used to store it. As displayed in Figure 7, to keep the component structure clear, each step of the actions is in a separate folder:

- *actions* - Actions are used in state management to express unique events throughout an application [27]. They are the input and output of many systems in Ngrx. They help to understand how events are handled in an application.

- *effects* - Effects are used to handle external interactions of a component [28].
- *reducers* - Reducers handle transitions from one state to the next state in an application. They are pure functions that take an input state and an action and return a new state [29].
- *containers* - Containers are Angular components that are responsible for managing the application's state and passing it down to other components. They are declared in the module, so the container and its child component will be rendered when the module is instantiated.
- *components* - Components are responsible for rendering data and user interfaces.
- *selectors* - Selectors are pure functions to retrieve slices of store state [30].
- *services* - Services are Angular constructs that fetch data from the server and validate user input. They can be made available to every component with dependency injection. A service must need one provider, and usually, it is registered within a specific module so that it would be available to all components in the module [31].
- *models* - Models define the data structures used in the component. They can be entity classes, enums or interfaces.

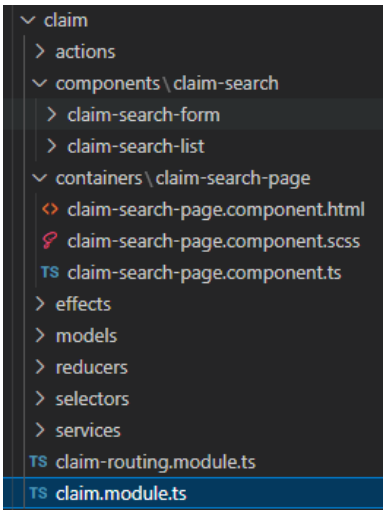

Figure 7. An overview of an Angular module's file tree.

6.3 ACQ Admin migration (ACQ-2316)

In this section, the development of the created Angular pages is covered. Each view's services, functionality and other aspects are covered. In the end, the total amount of time for different development parts is described to see how evenly the development time is

distributed between different parts of the development process. The pages are covered in the order that they were completed.

6.3.1. Sidebar and application setup (ACQ-2385)

Before functional pages can be migrated, a front-end application that displays the shared components, the header and the sidebar, must be implemented. The header consists of the following items:

- Company's logo.
- A dropdown list containing redirect links to other administrative UI pages.
- Link to Help page.
- Logout button.

The header functionality and design were implemented with a reusable component from the customer domain. Only the domain links had to be changed.

The sidebar's design was also implemented from the customer domain. However, the navigation groups were taken from the legacy system, as shown in Figure 8. The arrangement and grouping were discussed with the bank operation workers, so the functionalities they used the most would be the first options in the sidebar and the least used pages in the last.

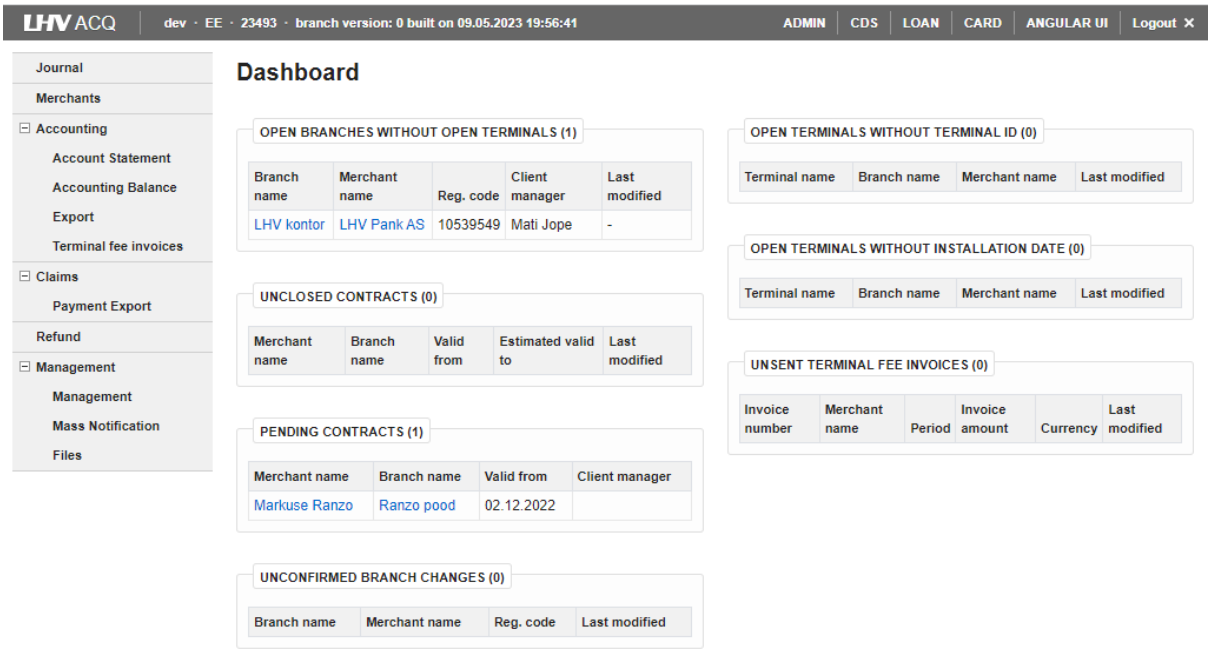

Figure 8. The view of legacy applications landing page.

Although the sidebar is filled with items that would display the site's pages, they have yet to be created. As a solution, a temporary redirect functionality was added, which will redirect an

authenticated user to the legacy page of the application. Further, these redirects will be removed incrementally when the views are migrated to Angular.

As shown in Figures 9 and 10, a blank landing page and static help page were created to ensure the application's routing works. In the future, the landing page will be a dashboard to display important information for the back-office workers to prioritise their work, as it was in the legacy version. The help page provides valuable links for the users to contact the development team in case of website problems.

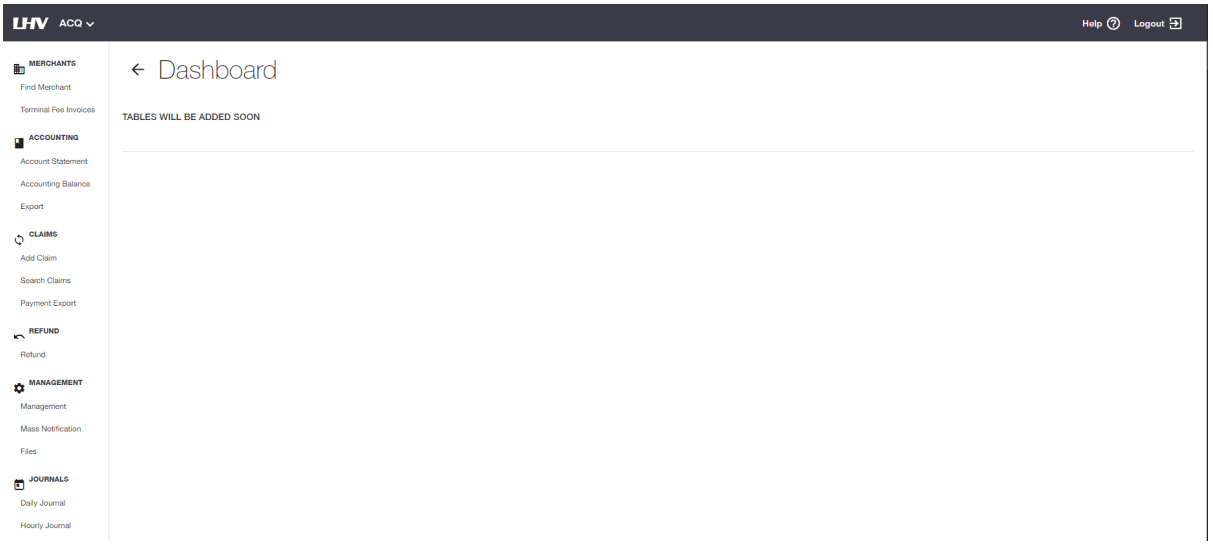

Figure 9. The landing page of the migrated application.

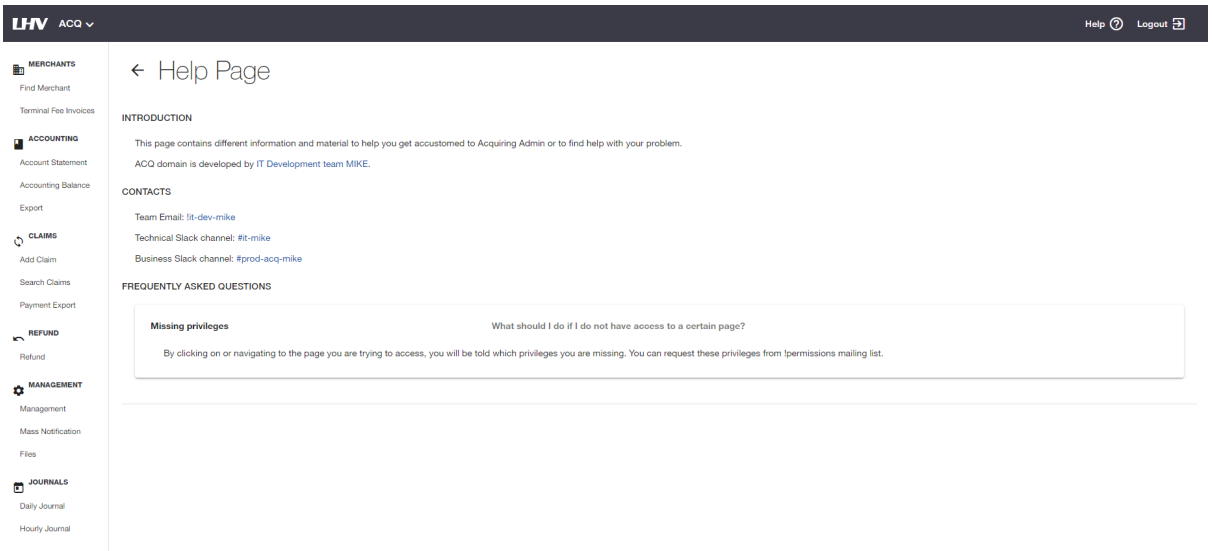

Figure 10. An overview of the Help page.

The initial pipeline in Gitlab was created so the commits could be built and deployed to an artifactory, which stores the versions of our company's projects.

The total time to complete this task was 13 hours and 30 minutes. From that, 10% was testing and deployment, 8.5% was the review, and the remainder was development, including implementing the review fixes.

6.3.2 Mass Notification page (ACQ-2333)

The Mass Notification page is used to send out emails to our customers. As the working language with different merchants may differ, the email is sent in three different languages- Estonian, English and Russian. Each language section has its own subject and body line, where prepared statements can be entered, such as problems with card payments and four other notification types, and modified. It is required to have the email written in all three languages. The user can add emails to where the message would be sent, separating them by a comma. The latter requirement is necessary to inform our partners. The user must be notified whether sending out the messages succeeded or not.

The legacy version of the page had two models described in the Thymeleaf framework. The notifications model served the default prompts and *massNotificationSendForm* to modify and send emails to merchants. The user interface had two HTTP request options to communicate with the pages controller class- GET request *getNotifications* to change the notification form's subject and body content after the user has selected the notification's type and a POST request to send the submitted form, see figure 10. The form was validated to check for empty fields and incorrect email input patterns. After successful validation, the emails are constructed and sent to companies acquiring domain clients.

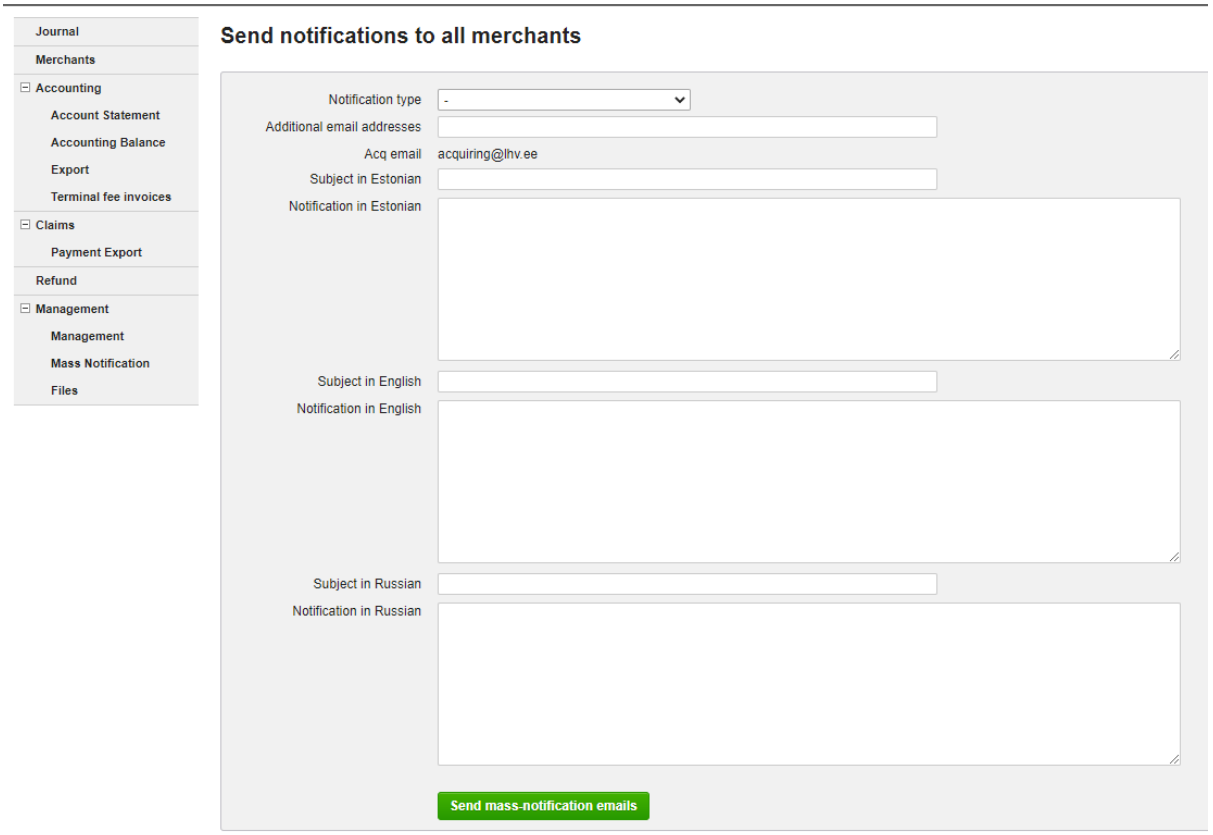

Figure 10. The view of the Mass Notification page's legacy version.

The upgraded view of the page is created as a *Mass-Notification* submodule into the *Management* module. The container of the webpage uses two components:

- Shared *page-title* component to display the name of the page and notifications regarding the requests done to the back end;
- *mass-notification-form* to create the email.

The page-title component requires the title of the page for input. The component has additional inputs like *description* to add a subtitle to the page and *back* to enable the in-site back button to navigate to the previous page. This component will be used in every view.

Mass-notification-form has two input properties, *notificationTypes* and *loading* to get the default prompt types and whether a request to get the notification types is currently in progress. The component has an output property *submitted* to send the event to its parent, which will initiate the HTTP request to the back end. The page can be seen in Figure 11.

When the page has been added to the DOM, a request to the back end is made to get the possible notification types and their prompts. The response is stored in the module's reducer

and accessed with a selector. Using a reducer in this instance is not necessary, as the state of this input data is irrelevant. The development was still done to get a better understanding of Angular's state management action pipeline. After selecting the notification type, the email fields are filled, and they can be edited. After the user has ended their interaction on a *form* field, it is validated against the rules set on the text areas using a Validator from the Angular's forms library. All the required fields must have a value to validate the form. Otherwise, the submit button is disabled; see Figure 12 for the invalid form view. If the form is in a valid state, see Figure 11, the button becomes interactable, and when it is pressed to submit the email request, which is followed by a dialogue modal to ask for confirmation to start the process.

The system's back-end functionality was modified to meet the requirements to transfer data over a controller and validate the POST request with the same constraints on the server side. It is necessary to increase the safety level of the application so intruders who could try to send their malicious requests to our system would have restrictions regarding their input data.

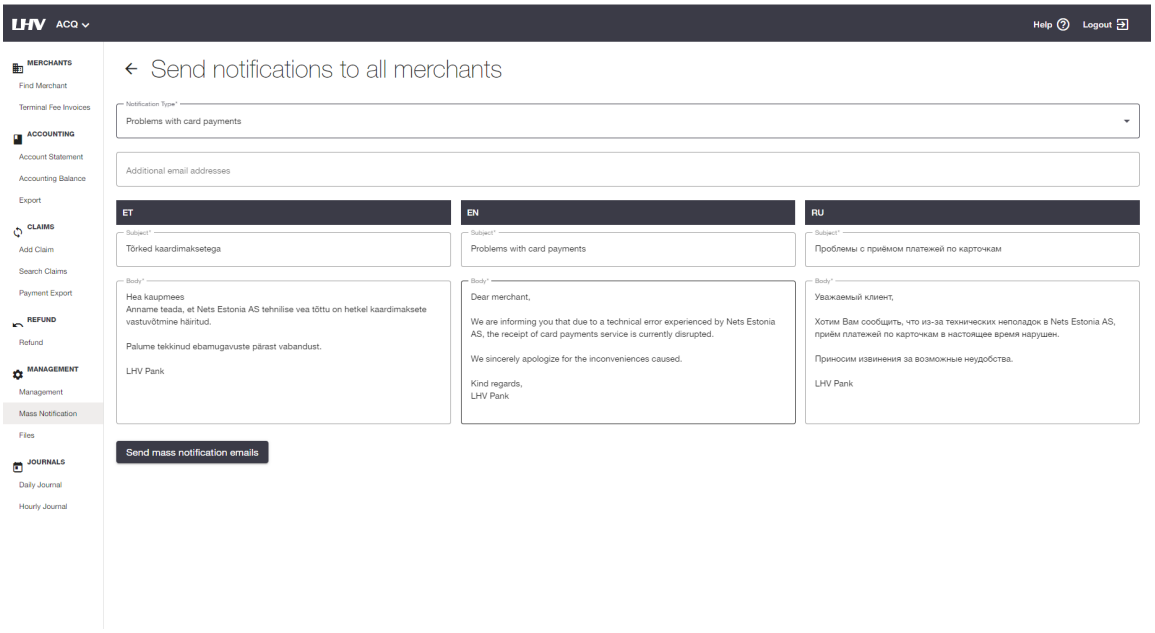

Figure 11. The updated view of Mass Notification page with "Problems with card payments" was inserted as a template.

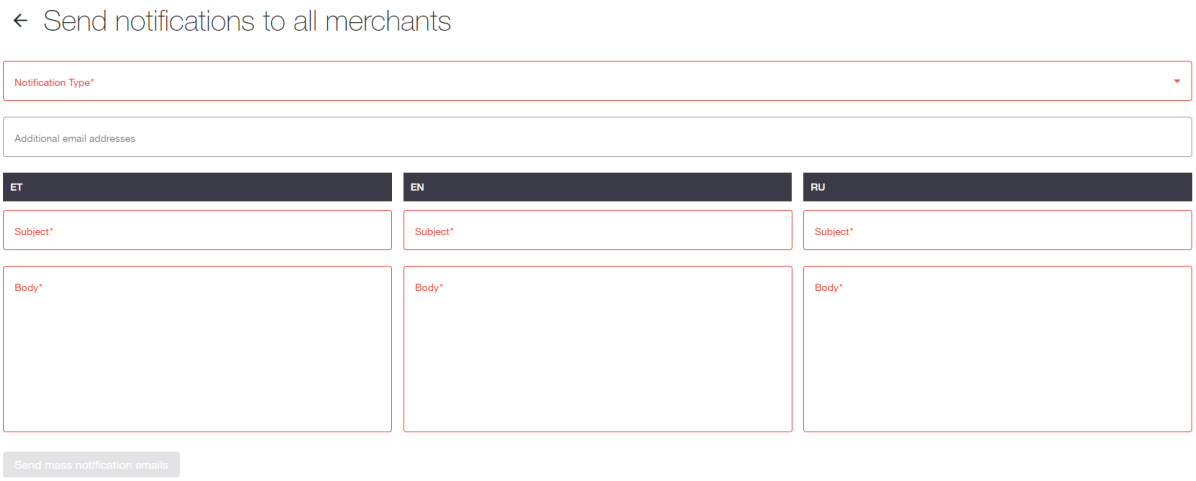

Figure 12. Invalid form fields displayed on the migrated Mass Notification page.

Many obstacles and errors had to be overcome as it was the first page with functionality and requests to the server. Thanks to these problems, such as how to add state management to a module or create dynamic forms, the developer's knowledge of developing in Angular improved vastly. As of writing, the team is still finishing testing the page as there were problems with serving the front-end application in test and prelive environments. The reviews for the task took over 10 hours, testing took around 90 hours, and the development with fixing review threads took 87 hours. The reason for testing took that long was to set up the Cypress framework. The biggest obstacles when setting up were getting the framework to function through our company's network proxy and setting up a Cypress session to make the login to the page through our systems admin single sign-on page quicker.

6.3.3 Management page (ACQ-2334)

The Management page allows the bank operations people to manually call some of the scheduled processes, clear different values in the cache and update values used in the automated jobs.

The page had 13 actions, as seen in Figure 13 for the legacy application's implementation, which made a POST request to the back end with the value of the action as a path variable and input data in the request body. Six buttons called services without a request body, and their outcomes were related to updating data of previous periods, deleting specific cache data from the system and sending data from one domain to another. The other seven options

consisted of services to request currency data for a specific date or update our system's business logic parameters to the necessary values.

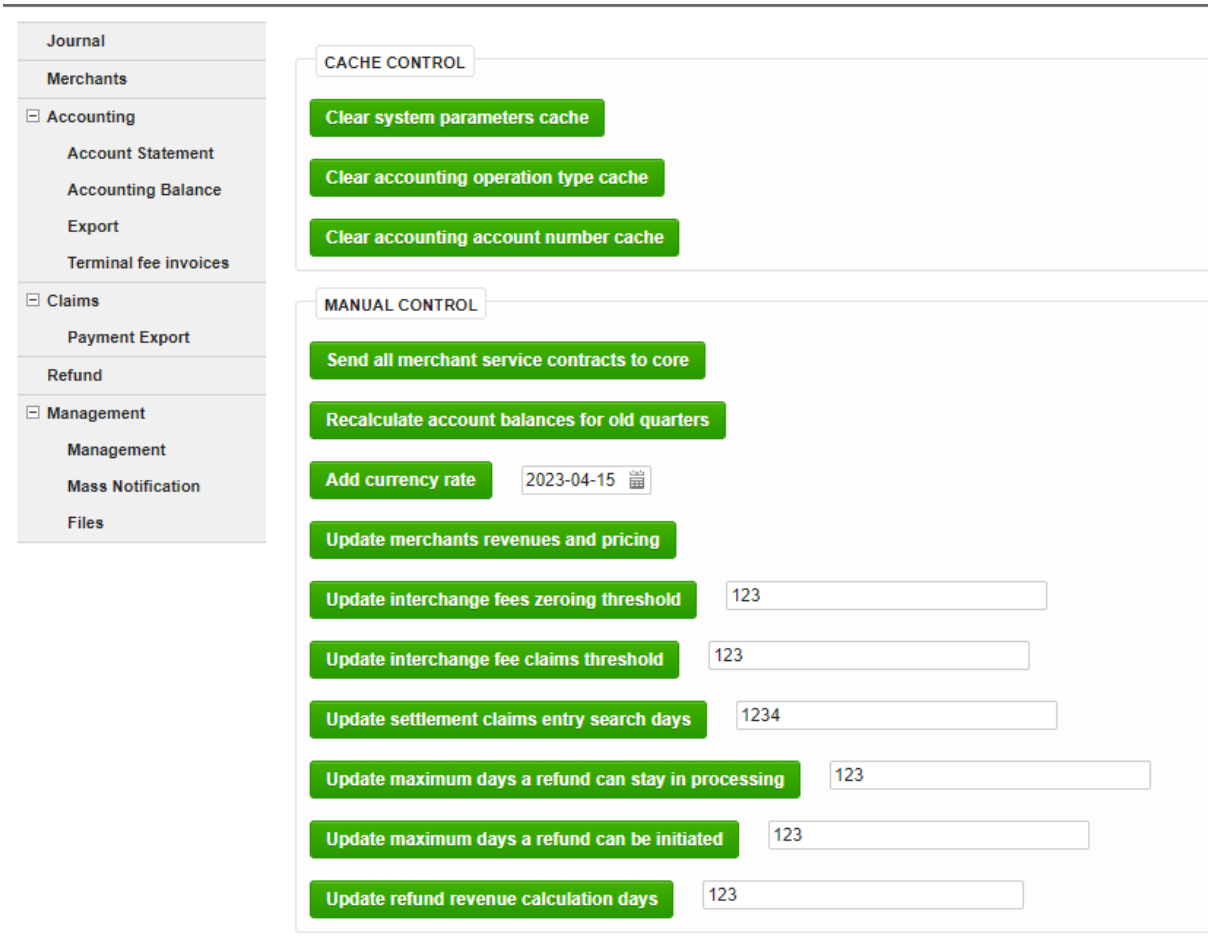

Figure 13. The screen of the legacy Management page.

During the analysis of this page, one of the options was supposed to be removed over six years ago. After consulting with the bank operations team, it was agreed that the option was redundant and would be removed.

While developing this page, the author was suggested to store pages' static data, such as labels and placeholder values, in a specific file. It also allows translating the page to other languages using the same key-value pairs to display the data. A simple translation module with dynamic JSON storage called *i18n*[13] was implemented. A JSON file had to be created, which stored the data, and the project had to be configured to use the file's data when needed. In components template files, the module was activated with the following prompt: *label.value.from.file* | *translate*. It was easy to implement *i18n* into the application, and it was

[13] Home page of i18n, https://github.com/mashpie/i18n-node

found to be quite helpful in avoiding typographical errors with label names that were used more than once.

The created page was a submodule of the *Management* module. Its container requested the module's service to get services that could be executable from the back end. The component displays the features in 2 separate blocks- cache control and manual control, see Figure 14. The blocks are divided into three columns - the feature title, the input if needed and the submit button. The forms' input values have default settings passed down from the parent component. Inputs have either a number or a date format input, and the latter has a maximum date restriction that does not allow entering a date that is in the future. Type validation is also done on the server side.

Back-end migration included extracting the business logic from the controller to a designated service, *ManagementService*, which communicates with other services to complete the request.

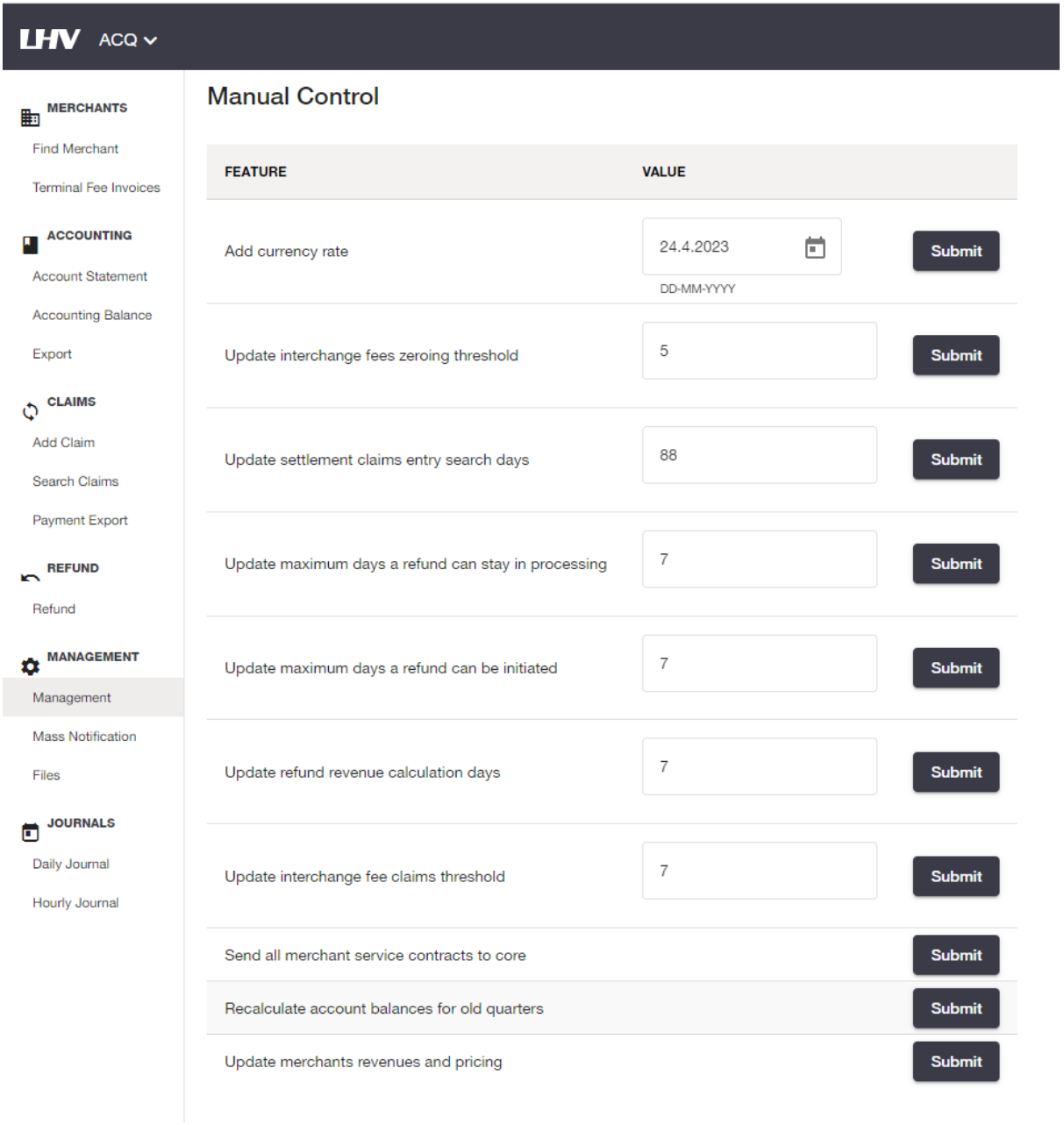

Figure 14. A view of the migrated Management page.

The Management page task still needs to pass the testing phase. However, it required 57 hours for development and for reviews, 7 hours. Around 20 hours of the development was for the front end, and the rest was for refactoring the back end.

6.3.4. Files page (ACQ-2332)

The Files page is used to have an overview of sent files that our partners have sent to us that we need to process. As seen in Figure 15, it consists of a form to make a desired query to receive the desired files by their type, name, the time period they were sent to the system or from what daily job they were stored in the system. A button, *Import files*, is displayed on the

page to request the files sent to the domains' server to be extracted to the application's database.

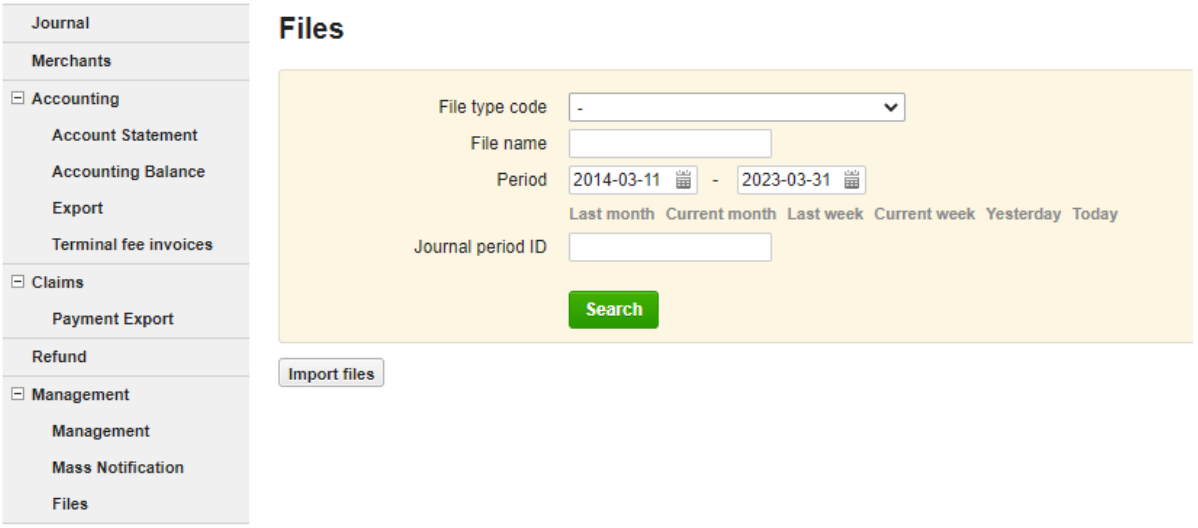

Figure 15. Form of legacy Files page.

Within the task, an Angular upgrade from version 14 to 15 was made to have the application in a newer setting as soon as possible to migrate the framework's version as quickly as possible. To complete the process, the frameworks maintainers have created a well-structured guide to complete the upgrade[14].

As many forms will require a date range selection, a shared component, date-period-with-presets, was created. The component also had default selections to get the following values: previous and current month; last and current week; yesterday; today. When injecting it into another component, attributes to modify the injectable can be added. Its attributes were showClear to add a possibility to empty date input fields when they are optional for the form, and the initial date range can be set with the defaultStart and defaultEnd attributes.

During the development of this page, it was learned how to make dynamic tables using Angular Material tables[15]. As displayed in Figure 16, the Angular version update did not affect the end result in appearance.

[14] Update Angular to v15, Google, 2022, https://angular.io/guide/update-to-version-15.
[15] Overview of Angular Material tables - https://material.angular.io/components/table/overview

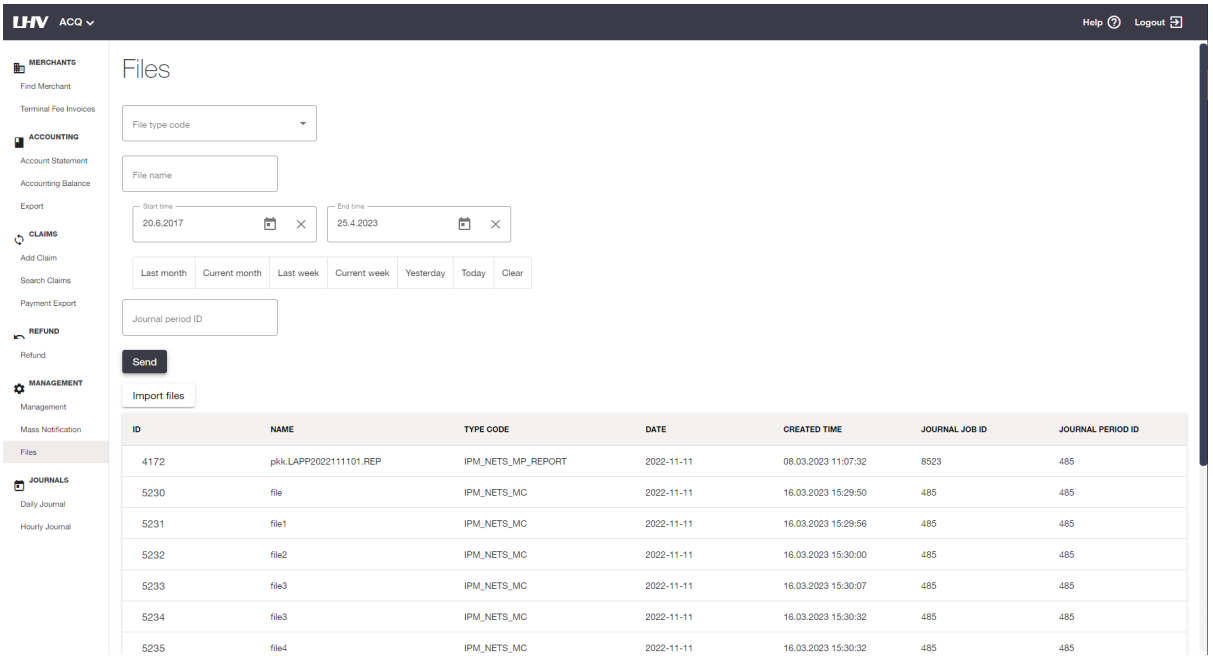

Figure 16. Updated look of migrated Files page.

ACQ-2332 still needs to pass the testing phase. So far, the development has taken 43.5 hours, and reviews have lasted for 3 hours. Of those 43.5 hours, ten were dedicated to migrating to Angular 15. Creating the page and back-end refactoring required an equal amount of time.

6.3.5. Refund page (ACQ-2336)

When any claim has been refunded, it will be displayed on the refund page. The view has a form consisting of fields for alphanumeric input field for merchant's id, number input fields for refund message id and original transaction id and required date input fields for start and end date to select refunds in the specific date range, see Figure 17 for the legacy version of the page.

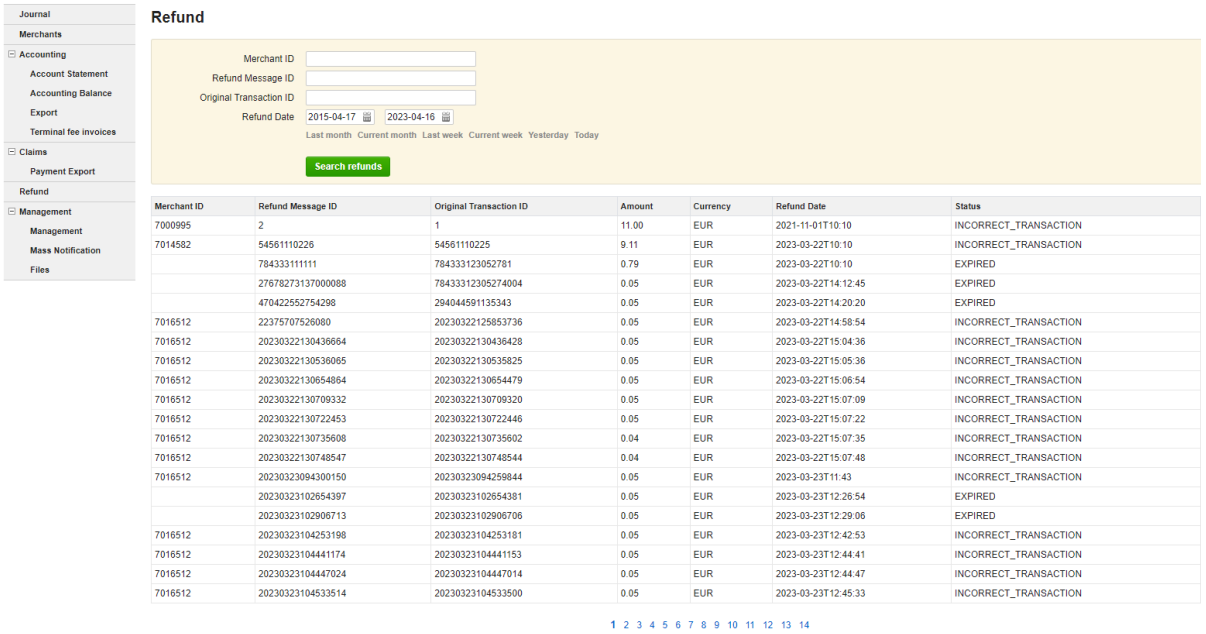

Figure 17. A display of the legacy Refund page.

Because the amount of refunds done in a given period is not limited, the amount displayed on the page and requested from the server must be constrained. On the front end, a paginator displays 10, 20 or 50 items at a time, 20 being the default option as it was the displayed amount in the legacy version; see Figure 18 for the created page with the paginator. When requesting the refunds from the back end, the size and number of the page are added to the request header to get the necessary response. Besides the refunds, the response also provides the total amount related to the form result so that the amount can be displayed in the paginator.

For some views, it is required to show specific results on view load. The Refund page is required to display refunds of the current day.

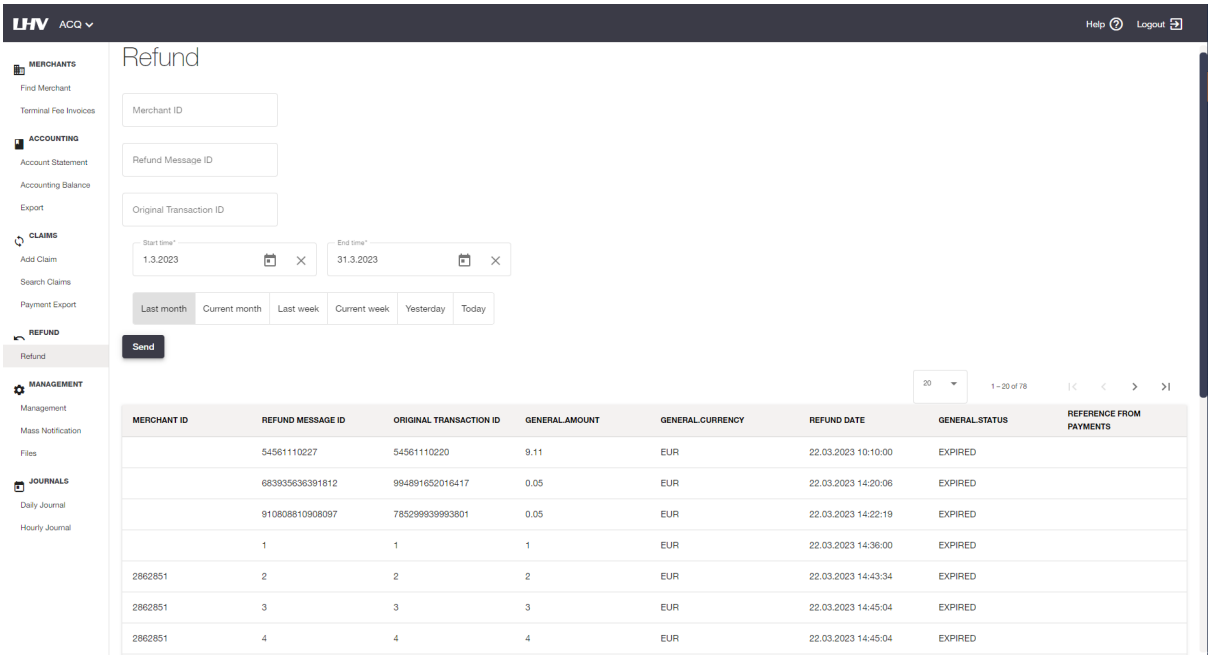

Figure 18. The view of the migrated Refund page.

The task still needs to pass the testing phase. It took 17.5 hours for development and 1.5 hours for review. The time between front-end and back-end development was equal.

6.3.6. Claim search page (ACQ-2337)

A claim search page is required for the bank operations team to see what claims have been made against our merchants and what are still unsettled. Unsettled claims are of high importance to the bank's workers because individuals have incorrectly distributed money until they have settled them. A search form can be filled, as shown in Figure 19, that consists of different identification options, currencies and claim types to find the desired claims. The results are displayed in a list that shows the most essential information about the claim. Each claim has a link tag that redirects the user to a more detailed view of the selected claim and a link to the merchant related to the claim. These views still need to be migrated to Angular. The user can also be redirected to create a new claim, to fix or settle different situations.

The back end had set a limit on the maximum amount of results to render to the document with 1000 results. When a search result exceeded the amount, the first 1000 results would be displayed, and a warning message would be shown. There were cases where the front end would crash with too many results that could have been displayed with a higher load. As this implementation impacts data availability, a change in the migrated system must be made to

divide the results between pages and display a set amount on each page, as implemented with the Refund page. The page users also requested adding a new field to filter currencies.

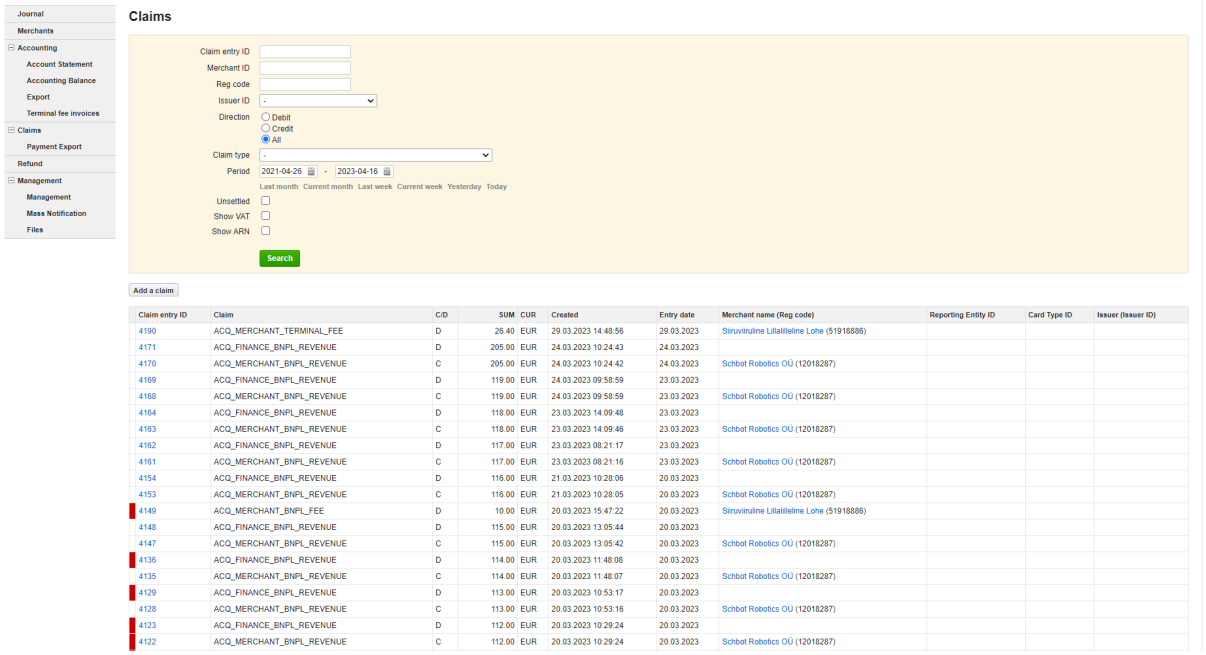

Figure 19. A display of legacy Claims search page.

As Figure 20 presents, the migrated search form to search claims now consists of 13 different options: two text inputs, one number input, one radio button group, three individual checkbox options, a start and end date, a dropdown list, and three input fields with dropdown filter option. The filter was implemented to make finding the correct values easier, as these input properties have many options. It was added to choosing the claim type, the issuer and the merchant related to the claim. When the user does not enter the exact value for these three fields, the inputs will not be used in the request.

When the view is opened, a request to the back end is made to receive the unsettled claims for the current day. This request will be taken under discussion by the development team on whether to initiate this request again when there are already claims in state management. Leaving the previous claims would mean the visitor would have this session's last search result, improving the user experience because they would not need to fill out their search form again to get the result. This feature would come in handy when the workers visit the claim details page and return to the search page.

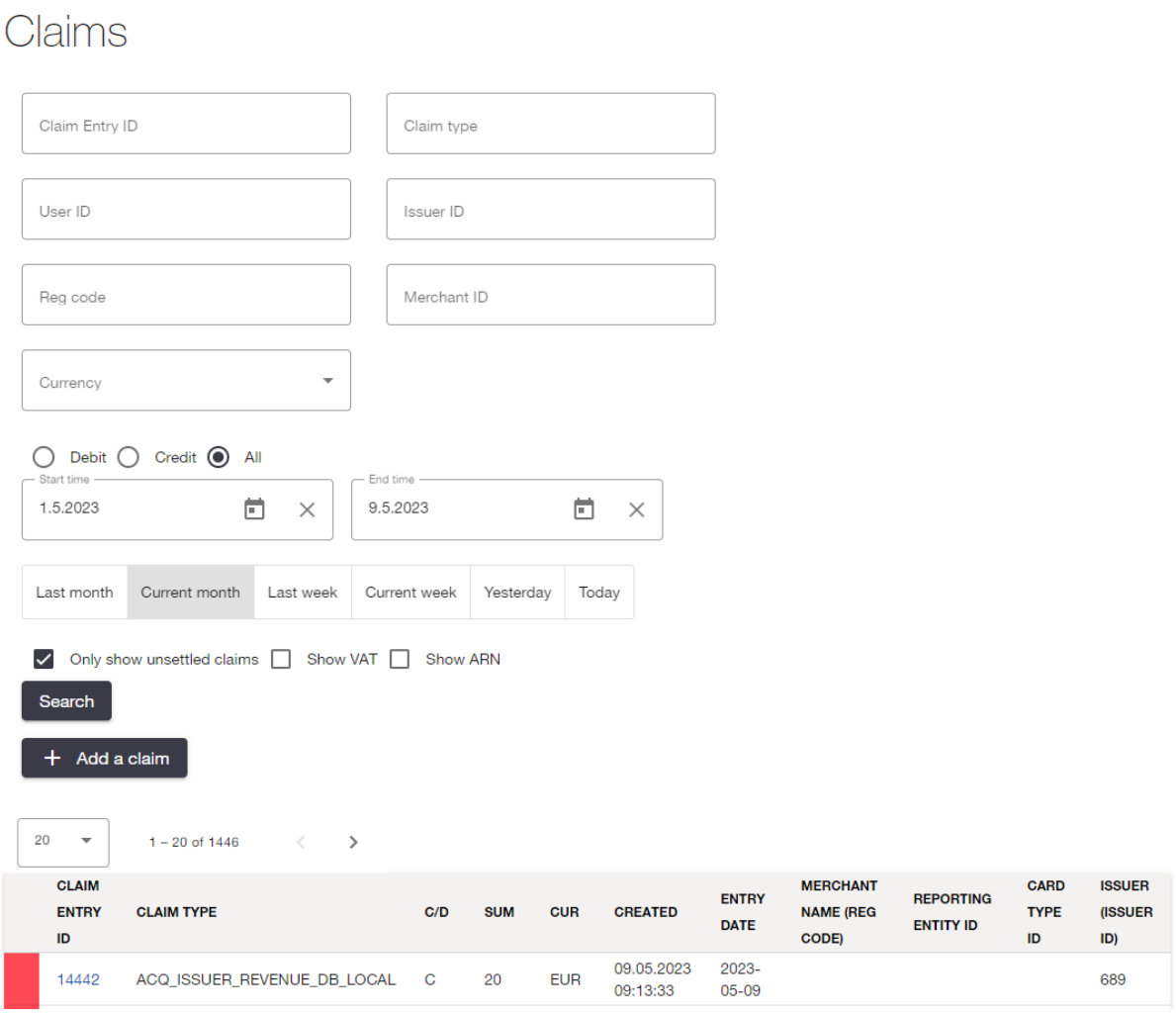

Figure 20. An overview of the migrated Claims search page.

The task still needs to pass the review phase. It has taken 39 hours for development, 14 hours for front-end development and 26 for back-end refactoring.

To conclude, there were five pages developed during this thesis. Four out of five pages had a search form and query results list, and one had multiple individual input forms to change the values of several features. The program has been under development for 255 hours; an extra 22 hours have been used to review the tasks, and 91 hours have been spent testing. Since these pages still need to be put into production, the time required to examine and test them will continue to increase.

7. Quality Assurance

This section discusses and compares the testing approaches and results between the legacy and migrated applications. The testing will be conducted on the author's local device. The specifications of the machine:

- Device: Dell Latitude 5421;
- Processor (CPU): 11th Gen Intel(R) Core(™) i7-11850H @ 2.50Ghz;
- RAM: 32 GB;
- Operation System: Windows 10 Enterprise (version 22H2).

The build times of the pages are checked from the projects' Gitlab pipelines and the local machine.

To run the legacy system locally, Gradle's *bootRun*[16] command was used to start the system. By visiting *localhost:8080/acq/admin*, the landing page was loaded to the browser.

There were two options to run the front-end migrated system locally:

1. run *ng serve* to run the front end in a development live server. Its drawbacks were that it does not minify some of the Javascript files, so the initial page loads, around six seconds, are significantly longer than usual.
2. Create a JAR file with Gradle scripts. They created the JAR of the application into the *frontend* module in the front-end project. The following command was added to the *back end's build.gradle* file to run the JAR when running the back end: *implementation files("C:\\path\\to\\jar\\frontend-1.0.7-local.JAR*. For the server to start the application, a redirect service was implemented so that when a user entered a URL of *localhost:8080/acq/admin/index.html*, the Angular application would load up. The page's loading speed is faster as the JAR is bundled and minifies the application's size down. However, when running some performance tests, the browser had problems staying in the Angular application. Instead of the migrated pages' view, the legacy dashboard will open with the migrated pages' URL.

In order to run Cypress tests locally, the application must also be live, so option one was used for it. The second option was implemented during the performance analysis.

[16] Spring Boot Gradle plugin interface guide -https://docs.spring.io/spring-boot/docs/current/gradle-plugin/reference/htmlsingle/#running-your-application

7.2. Back-end functionality

This section will discuss the main changes done to the back-end communication with the client and server. An overview of the validation methods applied to the communication will be given.

In the legacy system, Spring Controllers that receive the requests and respond to them fill different sections of the models that would be rendered to the page with the response. In contrast, the migrated application responded with a JSON request body. Every variable was validated using constraints from the *javax.validation.constraints* and *org.hibernate.validator.constraints* libraries to ensure that the requests would have the correct input. It is needed to validate the inputs in both the browser and server, as the request from the browser can be intercepted. If there are no validations, then it is possible to create denial-of-service attacks without much effort.

The move from using a model to transferring JSON objects made the communication between the client and server much cheaper. As the entire HTML was sent as a response to the client, the amount of data sent was significantly larger and the time it took to respond to different actions was also longer. As seen in Appendix I, the time and data were usually five times bigger for the legacy system than the migrated system. With larger requests, the data size differed by less than two times, but the time taken to complete the request was still five times longer for the legacy system.

Implementing a RESTful API has made the back end independent from the front end and made the data transfer quicker and payloads smaller.

7.3. Automated tests

The following paragraphs discuss the changes made in the automated tests for the front and back ends. Furthermore, the outcome regarding coverage and time taken to complete the tests will be compared with the legacy version. Only the main packages and modified pages will be used for quality assurance in the thesis, as the overall codebase is so big that the effect of the changes would not be seen in the coverage report.

The migration to Angular also led to transfer tests from Selenide to the Cypress framework. When transferring from a front end written in Java to one written in Typescript, the change had to be made to a framework that supported creating tests in the same language and was supported by the Angular framework.

The following command was executed to open the Cypress framework testing browser for the front end: *npm run cypress open –config-file ./cypress/local-config.js*. The tests were run on Google Chrome v112. The terminal's command to run the tests was: *npm run cypress run –config-file -/cypress/local-config.js*. The Selenide tests were run by selecting the packages and running them in IntelliJ IDE.

The user interface tests consisted of visiting the page, filling the form in the set way and ensuring that the correct result was received. Cases where the form is invalid, were also tested, and for situations where the server would return an error, an error message was expected to be displayed on the page.

As the expected results of the tests were the same, the key factor in the front-end migration was the time performance of UI tests, as they take much time in the development pipeline. When testing with Cypress and Angular, the current biggest bottleneck is the initial load time of the application. Before each test, Cypress resets the state of the application and makes a new initial load, but when the load is around 6 seconds locally, the time starts to pile up. As seen in Appendix II, the test suites usually run between 8-27 seconds, and most of the suites consist of several tests. The problem is more prominent with simpler pages like the ones created in this thesis. However, as the application gets fully migrated to Angular, the test flows are more extensive and the ratio between time-to-test and time-to-load should get to a better rate. Nevertheless, even due to the load time bottleneck, the results are already better than testing with Selenide. With these five pages, the time to complete the test run was 52% better with Cypress. The whole user interface test environment on Cypress could make the testing process 15 minutes faster in our pipelines and 4 minutes faster when running locally.

Secondly, back-end tests needed extensive refactoring as the integration tests do not follow many of the code practices we use today. The unit and integration testing levels had to be refactored for the back end so the code would be high quality. Many integration testing classes used stubbing to get results from a method, which might cause any side effects to the

system, such as querying from the database. In contrast, we now follow the statement written about integration testing in Spring Boot by Baeldung [32] that, as the testing levels suggest, the tests should integrate different application layers. That also means no mocking is involved. With this in mind, multiple instances of subjects had to be added to the database when setting up the test environment. Removing the stubs and mocks makes the system's workflow more thoroughly tested, meaning the code is of higher quality as the expected results are still received.

In addition, the tests were validated with Java Code Coverage (JaCoCo) plugin. It checks for the coverage of the complexity of the code and the missed pieces of code. As seen in Appendix III, most of the attributes got better for the majority of the pages. There are instances that the coverage remained the same or got slightly worse. For example, the files page did not have as good coverage as the legacy version. However, removing the mocks made the system more covered, which makes an argument that code coverage should not be the only thing when code quality is assessed. As seen from the report, there was only one line not checked with the tests, and as it was an exception call, when there are no files found to be imported, it is almost impossible to recreate the situation in a remote test environment as we do not store files that could be imported into the system.

7.4. Performance analysis

The performance analysis of the legacy and Angular application will be done in Google Chrome version 112 for Chromium-based and Mozilla Firefox Developer edition version 113 browsers. The built-in developer tool Lighthouse will assess the performance in Google Chrome. For Mozilla Firefox, the tool used is the built-in performance analysis tool.

As discussed in the migration analysis, it was believed that the initial load times would increase with the Angular application. The Angular application took four times longer to get its first content painted to the DOM and had it actionable compared to the Thymeleaf webpage.

7.5. Build time comparison

Part of the objectives of the migration was to make the developer experience better when they commit their features to the code repository. During the analysis, it was brought up that the complexity of the system overall will rise and

When comparing the pipeline times in the pipeline, the back end's jobs length has yet stayed the same, but it is believed to drop when more user interface tests have been migrated to the Cypress framework and front-end project. Currently, the pipelines for the back end take around 38 minutes to complete. For the front end, they are around 10 minutes. As long as the time reduces for the back-end project to finish its jobs and for the front-end project to get bigger at a similar pace, it will be okay for the development team. If both of the projects would start taking more time, then places for optimisation have to be checked.

Locally the legacy system completed its build job in 8m and 13 seconds, whereas the migrated version takes 7 minutes and 55 seconds for the back end and 42 seconds for the front end. With this, the time growth logic is the same as for the repository pipelines.

8. Conclusion

In conclusion, the switch from Thymeleaf to Angular has improved the system's overall quality, user interface, and developer experience. A more contemporary development environment, better code organisation, increased maintainability, and quicker front-end testing with the Cypress framework are all advantages of utilising Angular. A more thoroughly tested system has been produced due to the reworking of the back-end integration tests to eliminate the stubs and mocks, which is a significant increase in quality.

One of the primary objectives of the migration was to improve the system's performance, particularly in front-end load time. However, the Angular application's initial load times were four times longer than the Thymeleaf webpage, which is a notable drawback. The issue becomes more relevant when running UI tests with Cypress, as the time to load the application takes the majority of the time when running a test.

Using the JaCoCo plugin to validate the code coverage of the system showed that most of the attributes got better for most of the pages. Although, there are instances where the coverage remained the same or slightly worsened, such as the files page, where the coverage was not as good as the legacy version. This finding suggests that code coverage should not be the only thing considered when assessing code quality.

Regarding build times, the migration has led to a slight decrease in build times, which is a positive outcome. However, the time it takes for the back-end project to finish its jobs has yet to drop, and further migration of UI tests to the Cypress framework is required to optimise the pipelines fully.

In terms of further research, one area that requires attention is in the initial load time of the Angular application. This issue must be addressed to improve the user experience and to improve the testing process's efficiency. Secondly, opening different Angular application pages when launching the application as a library can increase the user experience. Thirdly, comparing the results of a separately running front-end application with the current front end as a library could bring compelling findings on the system's performance.

Finally, it should be noted that the migration from Thymeleaf to Angular is a complex process involving several steps and considerations. Despite some drawbacks and areas that require further research, the migration has significantly improved the system's overall quality, user experience, and developer experience. It is an excellent example of how modernising a legacy system can bring about significant benefits for all stakeholders involved.

9. References

1. LHV page of payment acquiring. https://www.lhv.ee/en/payment-acquiring (02.04.2023)
2. Apple Developers documentation page for Model-View-Controller. 2018. https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html (02.04.2023)
3. Britch D., Schonning N., Dunn C., Osborne J. The Model-View-ViewModel Pattern - Xamarin. Microsoft Learn. 2021. https://learn.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm (02.04.2023)
4. Webb P., Syer D, Long Josh, Nicoll S., Winch R., Wilkinson A., Overdijk M., Dupuis C., Deleuze S., Simons M., Pavić V., Bryant J, Bhave M., Meléndez E, Frederick S, Halbritter M. Spring Boot Reference Documentation. 2023. https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/ (02.04.2023)
5. Homepage of Thymeleaf. 2022. www.thymeleaf.org (02.04.2023)
6. Fielding, R.T. (2000). Architectural Styles and the Design of Network-based Software Architectures", Doctoral dissertation, University of California, Irvine.
7. Guide to Angular architecture. 2022. https://angular.io/guide/architecture (02.04.2023)
8. Saks E. (2019). JavaScript Frameworks: Angular vs React vs Vue. Bachelor's Thesis, Haaga-Helia University of Applied Sciences.
9. Decoupled definition on technopedia. 2021. https://www.techopedia.com/definition/598/decoupled (05.05.2023)
10. Documentation of Selenide. https://selenide.org/documentation.html (26.04.2023)
11. Documentation of Cypress. 2023. https://docs.cypress.io/ (26.04.2023)
12. Stargazers of Thymeleaf repository on GitHub. 2023. https://github.com/thymeleaf/thymeleaf/stargazers (26.04.2023)
13. Stargazers of Angular repository on GitHub. 2023. https://github.com/angular/angular/stargazers (26.04.2023)
14. Who is using Thymeleaf? 2023. https://www.thymeleaf.org/whoisusingthymeleaf.html (26.04.2023)
15. Npm package for Angular's main repository. 2023. https://www.npmjs.com/package/@angular/core (26.04.2023)

16. Job listings search results with the keyword Thymeleaf. https://www.linkedin.com/jobs/search?keywords=Thymeleaf&location=Estonia&geoId=102974008&trk=public_jobs_jobs-search-bar_search-submit (26.04.2023)
17. Job listings search results with the keyword Angular. https://www.linkedin.com/jobs/search/?currentJobId=3548404760&geoId=102974008&keywords=Angular&location=Estonia&refresh=true (26.04.2023)
18. Eriksson, J. (2022). Migration of the User Interface of a Web Application: from Thymeleaf to Angular. Åland University of Applied Sciences.
19. Stack Overflow Developer Survey 2022. 2022 https://survey.stackoverflow.co/2022/#most-popular-technologies-language-prof (03.05.2023)
20. Stack Overflow Developer Survey 2021. 2021. https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language-prof (03.05.2023)
21. Greif S., Burel E. State of Javascript 2022, 2023. https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/ (03.05.2023)
22. Angular version list and history. 2023. https://www.guru99.com/angularjs-1-vs-2-vs-4-vs-5-difference.html (03.05.2023)
23. Berner S, Weber R, Keller R.K (2005). Observations and Lessons Learned from Automated Testing. Zühlke Engineering AG.
24. How are customers' assets protected, https://www.lhv.ee/en/how-are-customers-assets-protected (26.04.2023)
25. Launching your app with a root module. 2022. https://angular.io/guide/bootstrapping (26.04.2023)
26. Introduction to components and templates. 2022. https://angular.io/guide/architecture-components#introduction-to-components-and-templates (09.04.2023)
27. NgRx- Actions. 2023. https://ngrx.io/guide/store/actions (26.04.2023)
28. NgRx- Effects. 2023. https://ngrx.io/guide/component-store/effect (26.04.2023)
29. NgRx- Reducers. 2023. https://ngrx.io/guide/store/reducers (26.04.2023)
30. NgRx- Selectors. 2023. https://ngrx.io/guide/store/selectors (26.04.2023)
31. Introduction to services and dependency injection. 2023. https://angular.io/guide/architecture-services (26.04.2023)

32. Testing in Spring Boot. 2023. https://www.baeldung.com/spring-boot-testing (06.05.2023)

Appendices

I. Time and data to receive responses from the server

Page	HTTP method	Request description	Time taken (ms)	Size of data
Claims (legacy)	GET	Get form selectable values	257	23.2 kB
	POST	Search query for no results	147	15.0 kB
	POST	Search query for 1000 results	1830	762 kB
Claims	GET	Get form selectable values	26	5.9 kB
	POST	Search query for no results	32	384 B
	POST	Search query for 1000 results	352	451 kB
	POST	Search query for 20 results with pagination	68	44 kB
Refunds (legacy)	POST	Search query for no results	102	11.5 kB
	POST	Search query for 20 results	72	24.5 kB
Refunds	POST	Search query for no results	19	400 B
	POST	Search query for 10 results	16	2.5 kB
	POST	Search query for 20 results	17	4.5 kB
Management (legacy)	GET	Get features	113	15 kB
	POST	Submit feature with input	54	311 B

Management	GET	Get features	20	2.4 kB
	POST	Submit feature with input	30	436 B
Mass notification (legacy)	GET	Get prompts	87	19.2 kB
Mass notification	GET	Get prompts	21	6.9 kB
	POST	Submit mass notification	61	420 B
Files (legacy)	GET	Get form selectable values	148	16 kB
	POST	Post query for 15 results	98	14.4 kB
Files (legacy	GET	Get form selectable values	9	1.1 kB
	POST	Post query for 15 results	29	2.8

II. Time comparison for user interface tests between the Selenide and Cypress frameworks

Page	Time taken in Cypress (s)	Time taken in Selenide (s)
Claims	19	55
Refunds (Cypress)	15	42.5
Management (Cypress)	19	35
Mass notification (Cypress)	8	43.5
Files (Cypress)	27	33.5
Total (Cypress)	88	209.5

III. Java Code Coverage rapport

Package	Instruction cov.[17] (%)	Branches cov. (%)	Complexity	Missed complexity paths (%)	Missed lines	Missed lines (%)	Missed methods	Missed methods (%)	Missed classes (%)
management (legacy)	87	72	49	14.29	16	12.40	2	5.26	0
management	92	82	85	8.24	14	5.81	2	2.82	0
mass-notification (legacy)	90	91	113	8.85	35	12.20	5	1.49	0
mass-notification	97	96	93	3.23	11	4.33	1	0	0
files (legacy)	100	91	15	6.67	0	0	0	0	0
files	98	88	19	15.79	1	1.92	1	10	0
refund (legacy)	93	92	189	7.94	22	4.67	6	4.69	0
refund	92	95	175	6.86	22	4.99	7	5.93	0
claim (legacy)	97	93	259	7.33	11	1.57	6	4.20	0
claim	98	93	255	7.84	10	1.45	5	3.73	0

[17] *cov*- coverage

IV. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Markus Kikkatalo**,

1. grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis

The migration of an administrative application's user interface from Thymeleaf to Angular, supervised by Vimal Kumar Dwivedi,

2. I grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in points 1 and 2.
4. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Markus Kikkatalo

09.05.2023