Western Norway University of Applied Sciences

BACHELOR'S THESIS

Graphical User Interface for Live Transcoding

Fredrik Fauskanger

Jonathan Hofslundsengen

Computer Science

Faculty of Engineering and Science

03/04/2020

I confirm that the work is self-prepared and that references/source references to all sources used in the work are provided, cf. Regulation relating to academic studies and examinations at the Western Norway University of Applied Sciences (HVL), § 10.

Western Norway University of Applied Sciences

Faculty of engineering and science

Department of Computer Science, Electrical Engineering and Mathematical Sciences

BACHELOR PROJECT TITLE PAGE

Report title: Graphical User Interface for Live Transcoding	*Date:* *01/06/2020:*
Author(s): Fredrik Fauskanger and Jonathan Hofslundsengen	*Number of pages without appendix: 42* *Number of appendix pages: 3*
Field of study: Computer Science	*Number of disks/CDs):* 0
Contact person for field of study: Bjarte Kileng	*Classification:* None
Remarks:	

Assigner: Vizrt	*Assigner reference:*
Assigner's contact person: Gisle Sælensminde	*Phone:* 926 29 259

Summary:

Development of a graphical user interface that, through REST and Websocket communication to a transcoder, allows for live transcoding.

Keywords:

Vue.js	TypeScript	Communication

Høgskulen på Vestlandet, Fakultet for ingeniør- og naturvitskap
Postadresse: Postboks 7030, 5020 BERGEN Besøksadresse: Inndalsveien 28, Bergen
Tlf. 55 58 75 00 Fax 55 58 77 90 E-post: post@hvl.no Hjemmeside: http://www.hvl.no

PREFACE

The report “Graphical User Interface for Live Transcoding” is written by Fredrik Fauskanger and Jonathan Hofslundsengen.

We would like to thank Vizrt and everyone at Vizrt who have assisted and guided us for the past months during the development and research of this project. Above all, we would like to thank Raymond Hilseth and Gisle Sælensminde for their useful comments and help during the project period. Additionally, we would also like to thank our teachers at HVL and give a special thanks to our supervisor Bjarte Kileng for being accommodating and available during the Covid-19 pandemic. This project would have proved to be more difficult without their guidance.

TABLE OF CONTENT

1 INTRODUCTION

1.1 Motivation and Goal

The goal for this project was to develop a web application that would display and control live transcoding jobs with ease. To make this possible the application was to utilize REST and WebSocket communication towards a transcoder. Transcoding is the act of converting a format to another, more specifically sending a converted file to an output. Regarding the task at hand, a subordinate goal was to implement a template system that would auto-fill input fields when initiating a new transcoding job. Another subordinate goal was to make the user interface clear and understandable for the user.

For us, the motivation behind choosing this task was that it seemed interesting. Regarding Vizrt, their motivation for this project was to shorten down the time needed to start up a transcoding job, as this was a request Vizrt had received from one of their customers. A job, in our case, is a transcoding job for a live stream that a user can create and manipulate as seen represented in Figure 1 below. Ideally our solution would allow a user to initiate a job in close to one click. Additionally, our implementation would allow the user to easily change and update the details of a job via templates, instead of being forced to create an entirely new job. A template is a representation of a job that can be stored to then later be realized into an actual job. What becomes stored are essentially the three topmost boxes in Figure 1. The purpose of this solution was to create a good user experience, where the user could easily access current jobs, and store templates with their respective details. It would also make it easier to access actions related to each job, such as start, stop, clone, update, and delete.

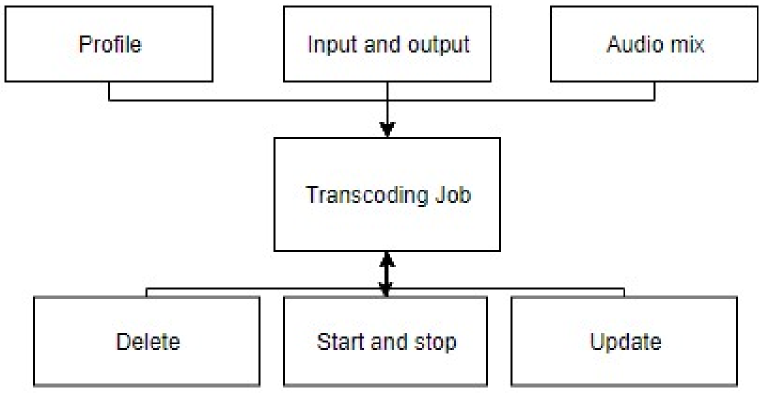


Figure 1: Transcoding job

1.2 Context

Vizrt, our task-provider, is a company that creates content production, management, and distribution tools for the digital media industry (Vizrt (1), n.d). Their interest in this project was based on using their transcoder API called Coder to prototype a possible solution for making live streaming more accessible for their customers.

Vizrt has one similar implementation, called Coder UI, however it is an in-house product for development purposes and is not available to the general user. A key problem is that the implementation of Coder UI does not have features like a template system. The relevance here was to produce a new application that would utilize the same transcoder, but with focus on making a better user experience with a more modern design. Coder UI utilizes Coder, which is the same API used in our project.

Regarding our project, we worked towards a solution that would allow the user to spend less time setting up a new job, and additionally perform related actions quicker. It would also open the possibility for businesses to save live transcoding templates for future use, which could make the process of live streaming to multiple platforms easier.

1.3 Limitations

There were limitations to our project mainly due to the Covid-19 pandemic. This impacted us in such a way that it became difficult to showcase solutions and cooperate at the Vizrt offices. Additionally, it required us and Vizrt to work divided, which again required us to plan more efficiently and become more patient as response to questions could take hours rather than mere minutes.

Another limitation that was put upon our project had been from our task-provider, where we had to take use of their STOMP broker. STOMP is a TCP like messaging protocol for WebSocket that allows communication between our website and Coder. A WebSocket is used to open a “two-way interactive communication session between the user's browser and a server” (Mozilla (1), n.d). The use of these technologies led us to spend time researching and observing how the backend functioned without the access to any documentation. However, this was considered a rather minor limitation as it made us acquire additional insight as to how their systems worked.

The last limitation that we had was us not developing everything that were in their pre-existing solution, that being Coder UI. This led us to not having access to creating new workers in our implementation. A worker is locally installed software that allows Coder to access video and audio devices. This means that we had to, and any future user must, set up a worker externally before being able to properly use our website. If Vizrt were to continue the development of this project they may address this issue.

1.4 Resources

For each practical issue we encountered using Coder, the communication with our task-provider went through Microsoft Teams or e-mail. We were also able to ask employees at Vizrt software related questions regarding the tools and programming languages we used, which are described in section 3.3.

Vizrt supplied us with licenses for Kaspersky Antivirus and Gitlab. Furthermore, we were granted access to Vizrt's internal software and intranet.

For theoretical or general questions our supervisor at HVL has been available for us.

1.5 Organization of the Report

The structure of our report is organized by chapters 1 through 10.

Chapter 1 details the motivation and goals. It also provides context regarding this project. We also detail limitations that were set to us and the resources we had available at hand.

Chapter 2 goes into detail describing the task-provider and the motivation they had for this project. It also provides information about the task-provider's initial requirements and ideas for the project, and how they use Coder today.

Chapter 3 describes how our project is designed and the different communication and design approaches. It also specifies tools we used, risks, and the evaluation methods.

Chapter 4 details our design choices and how the user interface works. This chapter also showcases differences between Coder UI and our solution.

Chapter 5 contains evaluations and methods of testing we utilized during development of our web application.

Chapter 6 discusses our project and the choices we made. It also talks about consequences due to the limitations stated earlier in section 1.3.

Chapter 7 concludes the project, summarises goals, and discusses possible future work.

Chapter 8 has the literature list.

Chapter 9 references figures in the project report.

Chapter 10 is the appendix, which contains the risk list, vocabulary, and GANTT diagram.

2 PROJECT DESCRIPTION

2.1 Practical Background

2.1.1 Project Owner

Vizrt was established in 1997 in Norway. Today, year 2020, Vizrt is one of world's leading providers of visual storytelling tools for media content creators and their customer base consist of media providers such as CNN and BBC. They operate in 30 different offices worldwide (Vizrt (1), n.d). Our office was in Media City Bergen.

First, the project was of interest to Vizrt as they wished for a more modern user interface related to live transcoding. Secondly, they wanted to upgrade their current implementation, ensuring further longevity of their API with a better solution for future support. Finally, they yearned for a product that could replace the graphical user interface their customers use for live transcoding.

2.1.2 Previous Work

Coder is a transcoder made by Vizrt. One of their products called Viz One uses this API. Viz One is an older user interface for Coder that is more complex than its successor Coder UI. In comparison to our solution, Viz One supports Coder functionality like distribution of VODs, Proxy, and Live transcoding jobs (Vizrt (2), n.d). Coder is also used in other products such as Viz Engine and Media Service and is also partly integrated in the transcoding backend of Viz Story.

They also utilize Coder internally for testing purposes. As mentioned, Coder UI uses Coder for the transcoding backend. This development-utility user interface was helpful to us towards understanding the kind of product Vizrt wanted.

During development, much of the inspiration regarding our solution was gained from how Viz One and Coder UI was set up. Coder had implemented a STOMP broker that Coder UI communicated with, to monitor transcoding jobs, and since we were to communicate using identical technology additional inspiration was gained through comparing our solution with theirs.

2.1.3 Initial Requirements Specification

We held a start-up meeting early March, where we discussed and went through details of the project. Following this we had two more meetings discussing further information and we were explained how Coder functioned along with deciding details of workplace and other practical details.

The requirements that were agreed upon during these meetings were:

- Monitoring of jobs.
- Create, update, and delete jobs.
- Possibility to look at details of a job.
- Actions such as stop and start for a job.
- Audio mixing with graphical tables for input.
- Storing of templates for later or instant use.
- Jobs should be able to use several outputs.

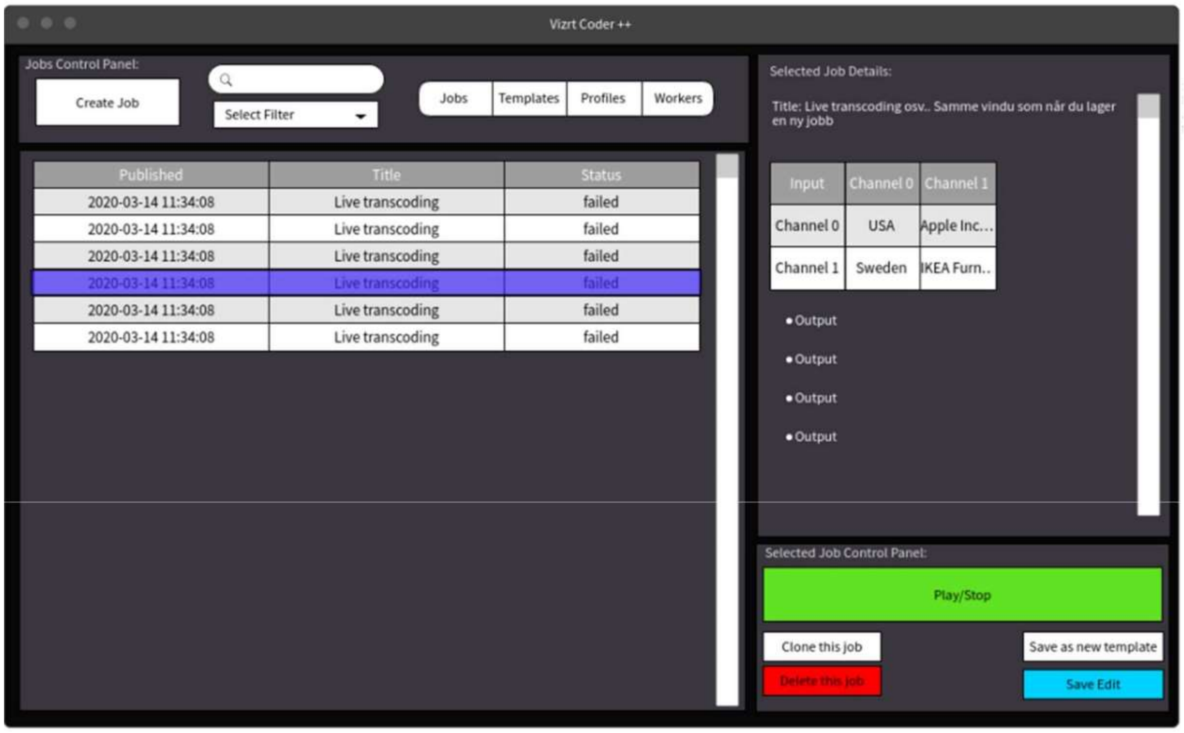
2.1.4 Initial Solution Idea

The initial solution idea that was discussed was a web application that would be hosted in Coder, with Coder acting as a backend transcoder. The communication with Coder would be done through REST API and a STOMP WebSocket connection.

Regarding the development of our website, it was decided that we would make use of technologies like Vue.js along with Vuetify. Vue.js builds upon the regular way of developing websites by introducing components. A component can be a focused part of a single website, for instance a menu. Vuetify builds upon Vue.js by providing additional elements like buttons and text fields to speed up the time needed to create a nice-looking website. Vizrt's designer had also interest in using TypeScript instead of native JavaScript. This was due to TypeScript introducing type interference in JavaScript, which may be an upside when programming large scale applications as the program will not run if there is a type interference.

When we planned the idea for the user interface, we had interest in making it easily understandable for the general user. With long term usage in mind, we sought out the idea of using filters to sort out jobs, in a way search, to make it easier to find the required job.

The user interface was designed in the early stages to give us and the task-provider an idea as to how the user interface could look. The first digital drawing is shown in Figure 2.



Published	Title	Status
2020-03-14 11:34:08	Live transcoding	failed
2020-03-14 11:34:08	Live transcoding	failed
2020-03-14 11:34:08	Live transcoding	failed
2020-03-14 11:34:08	Live transcoding	failed
2020-03-14 11:34:08	Live transcoding	failed
2020-03-14 11:34:08	Live transcoding	failed

Title: Live transcoding osv.. Samme vindu som når du lager en ny jobb

Input	Channel 0	Channel 1
Channel 0	USA	Apple Inc…
Channel 1	Sweden	IKEA Furn..

Figure 2: Early UI Design

2.2 Literature Background

As mentioned in section 2.1.4, our preliminary task was to primarily work with and combine Vue.js with TypeScript and connect this to Coder, then subsequently create a website using these modern languages and technologies.

This would prove to be challenging as the need to experiment and learn these languages and systems became a necessity before starting the development of the final product. Both of us had little to no prior experience with neither of the languages we were to use, and therefore a lot of time was spent training and acquiring these languages to deliver a satisfying product to the task-provider.

In addition to self-learning these languages we also had to embark onto the world of audio mixing and streaming as our task was, as mentioned previously, to create an interface to manipulate their API that provided services within these fields.

3 PROJECT DESIGN

The project goal that was set by our task-provider was, as mentioned, to create and improve upon their UI to live transcode video and audio feeds to a single or multiple designated output destinations, examples to such destinations are YouTube or Facebook. First, they wanted to have it more user friendly, or more specifically, they wanted to be able to seemingly halt and resume any ongoing stream on demand. Secondly, they wanted to be able to create and save templates for easier long-term usage. Originally, you would have to either clone an existing job or input all required field from scratch to recreate a job.

3.1 Possible Approaches

In our discussion on how we were to approach our task, it was important to know how to handle the data we received as it required rapid updates regarding concurrency of jobs. As such, we researched different solutions in handling the communication and UI design, as well as comparing their pros and cons.

3.1.1 Communication

When exploring our options towards receiving and transmitting data with Coder, we first started testing with the REST API. While testing the API calls, we realized that even though we could do a GET request to get all the current jobs, it would not be ideal to keep the application updated in real time. This moved us into looking at other options.

Another option we could have used was short polling. Short polling is the process of getting data from a data source, as it requests data periodically, and it would update and replace when there is new data available. This is an interesting option; however, by itself it would not become a real-time application as that was of our interest (Wikipedia Contributors (1), 2020).

We also investigated long polling, which is like polling, where the main difference is that instead of getting new requests continuously, “it holds the request open until new data is available” (Hanson, 2014). However, by itself this solution would wind up being not ideal for our use.

The fourth option we explored was STOMP WebSocket as it is a good alternative to update in real time and is less straining for the server as it sends less expensive data, such as the headers (Kilbride-Singh, 2019).

3.1.2 User Interface

When we first discussed the design of the user interface, the task-providers wished for similar design to the pre-existing design. However, we found this design hard to read for new users. This led us to designing the user interface shown in Figure 2 in section 2.1.4. We worked on that idea with the mindset that only minor changes could be made, as we had to work on the terms we were given. When it came to programming in Vue.js, we had to carefully consider what components we could use to accomplish what we thought of as an ideal interface.

The first option regarding the layout of the webpage would be to divide it into three statically placed parts, each containing information only regarding itself. Then properly differentiate the parts so that they become distinctive and more easily recognizable. For instance, having menus and a filter at top, list of jobs or templates to the left, and creation and information about a job or template at the right.

The second option would be to replicate the user interface of Coder UI but in a slightly altered state. This would mean having menus and filters fixed at the top, list of all jobs centrally, information of a selected job at the bottom of the window, and the creation of jobs via a designated modal. This modal could then contain a drop-down list of all possible templates to use.

The third option would be to have designated parts like the first option, but let the parts be dynamic, meaning scrolling would happen within a component rather on the page itself. In Figure 2, in section 2.1.4, you can see our representation of this option. This would mean that the job control buttons, and general inputs would always be visible for the user and grant simpler use of the webpage.

When it came down to style and colours, the only requirement was that it had to be a 'dark-mode', which is where the font is brighter than the background (Wikipedia Contributors (2), 2020).

3.2 Specification

3.2.1 Communication

We considered the different options and finally decided upon taking use of both REST calls, polling and of the STOMP broker.

Considering we had live jobs that needed to be monitored for real time updates, the WebSocket was the best option available for us at hand. Since Vizrt were already using a STOMP broker we agreed to go with this option. However, since we could not edit the current implementation of the broker, we could not directly build upon their WebSocket for our use.

When creating, deleting, and editing jobs, the best option to use was in this case the REST API calls, as they were the most efficient and easy option.

Lastly, we were required to retrieve inputs from our local machines which were stored as information on the transcoder. This information would have to update occasionally, and with this considered it would be most ideal for us to utilize short polling for updating when a new worker appears.

3.2.2 User Interface

We ended up with the first option of the three options regarding the UI alternatives. The reason for this was because it addressed issues in Coder UI and was much simpler to develop than the third option. Initially our thought was to go with option three, as shown in Figure 2 in section 2.1.4, but it proved too time consuming.

Our design of the user interface was used as a guideline for implementing front end elements within Vue.js. Our focus when we designed this new user interface was to make the system more intuitive and easier to grasp for the end user. We decided on having visualization of the same type of components, such as elements containing similar information, to be placed in the same location on the screen. Not necessarily on top of each other, but so that the interface would switch between these views whenever it would be expected and reasonable. This way the user would instinctively know where to look for certain functions or statistics based on its type of information, for instance creation panel as opposed to control panel.

Another part we decided on changing was the sizes and colours of buttons. We could have kept the same exact colours as in Coder UI, but we felt the grey buttons on a grey background were not readable enough. It was important that key functions were easy to notice as well as engage with. Having bright coloured buttons, and a good colour palette that contrasts well with background colours further improved the usability of the user interface.

Designing a state machine was something we quickly put together to use as a guideline for implementing interaction of the user interface within Vue.js, while also using TypeScript and STOMP. Our focus was to have the system functioning with fewer actions performed by the user. An example would be having the user press one button and the rest of the logic would happen backstage. However, this would prove difficult as the user would still have to choose details for each job created. We ended up with a design where the user could technically create a job using only four mouse button presses through the template system.

The part of the user interface called the control panel was made responsible for handling views, search, and filtering, as well as operating key functions of any selected job. Our given task did not involve changing views to profiles and workers, but merely to manipulate and observe templates and jobs. The control panel does not work with templates concretely but does allow other components to do so. Finally, as the control panel is a general component it was placed at the top of the window.

Our implementation of job-listing, as seen in Figure 8 and 9 in section 4.2.2, listed all available jobs in a selectable manner regardless of each job's content. We decided on simplifying this aspect of the user interface in comparison to the original design of Coder UI by removing some fields like Summary and Status Message, and instead only keep these three fields: Published, Title, and Status.

The job creation panel component was made to represent information about a selected job, to create a new job, and to create a new template. These three different situations are similar enough so that we decided to designate them to the same position and component of the overall user interface.

We did consider having the control panel fixed in place so that it would not hide even if you were to scroll. However, we figured it was best to place all the components of the webpage statically to minimize complexity.

Another part we changed was the layout in which audio mixing were to be done. Instead of having the user type in 1's and 0's in a text field, the user can now visually press more comprehensible buttons to add or remove audio input rows and to switch between 1's and 0's. This would also induce, for the user, a better understanding of what the creation panel actually expects the user to provide. The only issue that could arise here was the fact that you can only select 1's and 0's, and nothing in between.

3.3 Selection of Tools and Programming Languages

We were specified by our task-provider what languages to use. The tools used were therefore influenced by these specifications, though the choices we made were not all directly compulsory.

3.3.1 Tools

Vue UI or npm (Node Packet Manager) is what we used to run our website while developing. We also used them to create new project-folders, of which we would later use for testing. These are also tools needed to support the dependencies of the website.

Coder runs as a single-network-service and is controlled through a REST-based HTTP API and was what we based our website around. As mentioned previously, this was the backend of our website and it provided the functionality for creating jobs and manipulating jobs. Additionally, it also stored all jobs and templates.

The editor of our choice was Visual Studio Code which “is a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux." (Visual Studio (1), n.d). We used this editor to develop our website, mainly utilizing extensions to use it with Vue.js and TypeScript.

Git and GitLab is what we used to push code onto a database to synchronize code and to store it. Vizrt can therefore, if they wish, proceed to develop this project as the code is theirs.

Jira was provided to us by Vizrt to be used for creating sprints and documenting goals we were to do during this period. However, the sprints were quite ubiquitous, and they ended up staying virtually unchanged. Therefore, our unofficial and main source of updating and understanding goals were through oral communication and messaging services like Discord and Microsoft Teams. Jira instead stayed as a backbone for project progress.

NDI Monitor Studio is a program to view NDI video sources that is owned by Vizrt. We used it mainly to test our system, and to double check if jobs connected and sent the information necessary to stream the video source the jobs would have as input (NDI.tv, n.d).

cURL and Postman were used to observe what was being requested or responded to from the API in Coder UI, and to test whether our creation worked as intended. These tools allow the use of HTTP methods through REST calls, such as GET, POST, DELETE and PUT (REST API Tutorial, n.d).

3.3.2 Programming Languages and Technologies

In our project, we used eight programming languages and technologies:

Vue.js “is a progressive framework for building user interfaces” (Vue.js (1), n.d) to create user interfaces to be used against front end applications as an alternative to standard HTML5, CSS3 and JavaScript.

Vuetify is an extension to Vue.js that adds components to simplify the development of Vue.js websites (Vuetify, n.d).

TypeScript is a JavaScript extension that includes types into the language. This is to improve and prevent error-proneness in web applications. It has a much stricter syntax and “adds optional static typing to the language” (Wikipedia Contributors (3), 2020).

JavaScript is a scripting language supported by virtually all browsers and allows for more complex websites than what regular HTML can support (Mozilla (2), n.d).

HTML5 is the newest version of HTML and is the foundation of virtually all websites, and “HTML is the standard mark-up language for creating Web pages" (w3schools (1), n.d).

CSS3 is the newest version of Cascading Style Sheets and "CSS is a language that describes the style of an HTML document" (wc3schools (2), n.d).

STOMP (Streaming Text Oriented Messaging Protocol) and WebSocket is what we used to communicate between Coder and our front end. STOMP can be used to host a broker for clients. We used STOMP to subscribe to the API’s message broker to update our client of the broker’s messages whenever they occurred. The format of these messages was converted to XML (Mesnil, 2012).

The Atom format is an XML based format containing feeds (IETF (1), 2005). These feeds have entries that contain metadata. We also took use of the Atom Publishing format that opens up for use of HTTP to get collections of entries, and create, update and delete these entries (IETF (2), 2007).

3.4 Project Development Method

The methodology used in this project was Scrum through Jira. This choice is further explained in section 3.4.1.

3.4.1 Development Method

Our choice of using Jira can be explained simply by saying that it was the preferred method of which our task-provider ran their projects, and as our project laid under their domain, we were to use their practices. We were provided accounts and access to boards within their system to allow our task-provider to both provide sprints as well observe our progress throughout this project.

3.4.2 Project Plan

The project plan had one main objective, of which was to improve and add functionality to the web application Coder UI by creating a new and similar web application. First, we had to understand the structure and functionality of Coder UI to realize the improved version of it. This was put into action through a sprint within Jira.

Another part of the plan, and the underlying reason of creating a new project, was to move the web application over to more modern technology like Vue.js.

3.4.3 Risk Management

3.4.3.1 Global Pandemic

The Covid-19 pandemic of 2020 genuinely interfered with our goal to be able to use the product with a physical SDI camera at the location of Vizrt in Bergen. This, however, did not halt our progress but instead merely transformed our goal into just creating a generally functioning website.

3.4.3.2 Not Reaching Our Goals

Failing is something we attempted to avoid at all cost. However, this depended more on the importance of the goal than it being a goal by itself. Some goals had more importance than others, and we prioritised function over form. For instance, having a functioning web application was more important than having a magnificent style.

3.4.3.3 Lack of Competence

We had certain challenges to overcome when developing our solution, including:

- Storing of templates.
- REST requests updates to jobs, which may include a WebSocket or polling.
- REST POST requests to the feed collections in Coder.
- Hosting on intranet.
- Input and output strings could be very long. Occasionally strings became too wide in comparison to the window or their designated field.

3.4.3.4 Lack of Motivation

It was important for us not to lose any hope and desire regarding this project, as it could have incurred us falling behind. We therefore made sure to encourage and provide responsibilities, not only to oneself, but also to the other member. With a clear goal in mind, in our opinion, motivation is easier to acquire.

3.4.3.5 Misunderstanding Requirements and Specifications

The chance of misunderstanding the requirements were relatively high as our competence, or initial knowledge, was limited. Nonetheless, our solution to this problem was to firstly notate all encounters with the task-provider, and secondly ask and research issues we were unsure of.

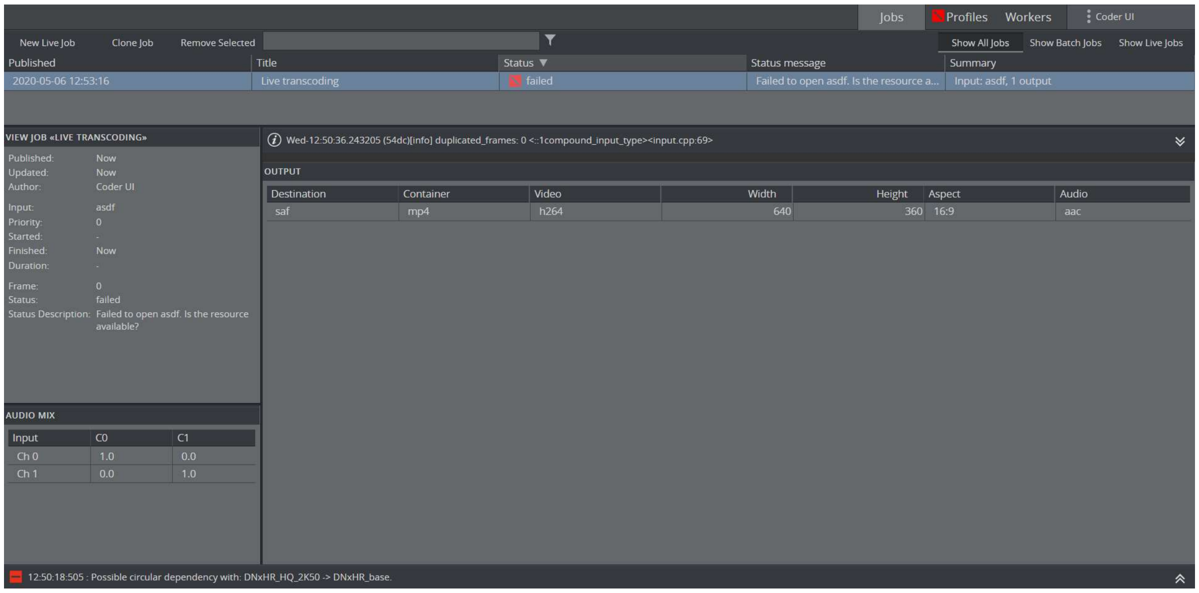
3.5 Evaluation Method

Mainly, the way we evaluated this project was by testing functionality against Coder and by comparing our solution to the requirements made by the task-provider. This would require us to test via both tools and feedback. Ultimately, the tools to test against was NDI Studio Monitor, Postman, and cURL and feedback was through our task-provider. Our solution would become pertinent solely based on these evaluations resulting in success.

4 DETAILED DESIGN

4.1 Initial Phase

In the beginning Vizrt showed us their own Coder UI as seen below in Figure 3.



Published	Title	Status	Status message	Summary
2020-05-06 12:53:16	Live transcoding	🟥 failed	Failed to open asdf. Is the resource a...	Input: asdf, 1 output

VIEW JOB «LIVE TRANSCODING»

Published:	Now
Updated:	Now
Author:	Coder UI
Input:	asdf
Priority:	0
Started:	-
Finished:	Now
Duration:	-
Frame:	0
Status:	failed
Status Description:	Failed to open asdf. Is the resource available?

```
Wed-12:50:36.243205 (54dc)[info] duplicated_frames: 0 <::1compound_input_type><input.cpp:69>
```

OUTPUT

Destination	Container	Video	Width	Height	Aspect	Audio
saf	mp4	h264	640	360	16:9	aac

AUDIO MIX

Input	C0	C1
Ch 0	1.0	0.0
Ch 1	0.0	1.0

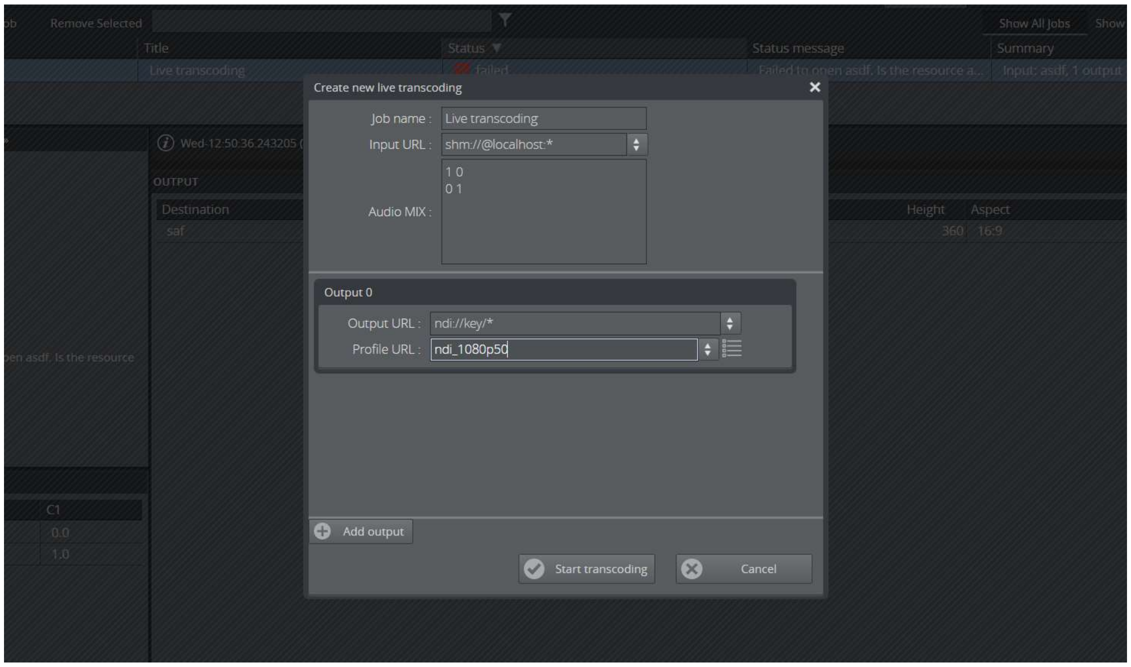
```
12:50:18:505 : Possible circular dependency with: DNxHR_HQ_2K50 -> DNxHR_base.
```

Figure 3: Vizrt own Coder UI

As previously told, the task-provider wanted to have the same product, only with more features like templates and a start and stop button. We also asked them about colour scheme, and they wished to keep a "dark mode" style. They also told us that we did not need to worry about designing profiles and workers, as seen at the top right corner of Figure 3.

In our opinion the colour scheme was dull, and it was hard to understand where to lay your eyes to get the necessary information as well as where you should press buttons to do certain tasks. We therefore told them that this would be on our list of things to address. However, the icons that show in the status column were nice and distinguishable.

Their website is very dynamic and allows for resizing of elements like the bottom panel of Figure 3 and moving of the job creation modal, titled “Create new live transcoding”, as seen in Figure 4.


Create new live transcoding

Job name : Live transcoding
Input URL : shm://@localhost:*
Audio MIX :
```
1 0
0 1
```

Output 0

Output URL : ndi://key/*
Profile URL : ndi_1080p50

Figure 4: Showcase of creating new job in Vizrt Coder UI

Even though these are neat and sometimes useful features, we did not see any major reason to reimplement these features as it would further complicate the task without providing major user benefit. In other words, we opted for a more static design in this case.

The audio mix layout in their job creation modal was a major source of discussion as well. At first, we wondered if we should keep the layout that they had. They used a multiple lined text field without any explanation for what the numbers meant. For instance, it was unclear whether the rows or columns represented the input audio. We concluded that this had to be redesigned.

4.2 Layout

As seen in Figure 2 in section 2.1.4, our initial design focused on having clear placements and colours for the user to easily comprehend. Our end design held onto this principle, but with an additionally simplified form. General controls were placed at the top, listings at the left, and creation, manipulation or viewing in the right part of the window.

Because of requirements given by the task-provider regarding keeping the colour palette dark, we had to make it as such. However, we firmly decided upon altering their original dark colour palette by adding strong colours where important elements would lay. In addition, we made sure to disable buttons that the user should not press before necessary, like 'Create' and 'Revert'. Our thought was to guide the user and clarify when these buttons genuinely could do a useful action. This would avoid spawning unnecessary errors and avoid confusing the user with buttons that might seemingly do nothing.

4.2.1 Control Panel

As mentioned, at the top part of the website we decided to have general controls. We named this component the control panel since it was used for creating new jobs, filtering jobs in the list below, and flipping between different parts of the website. The idea was to allow for flipping between seeing jobs, templates, profiles, and workers, but the views, and thereby buttons, for profiles and workers were never created as the task-provider did not require, nor wish, for us to develop these parts.

With our initial design, control buttons like play and stop were designed as a part of the control panel, however we felt that these were too specifically regarding jobs. They were instead placed into creation panel as first intended, as they fit more thematically there. This subsequentially decreased coding complexity as communication between components in our project, using Vue.js mixed with TypeScript, was relatively heavy to code without being overwhelmed with warning from TypeScript.

The filtering design was made so you could either type in manually or select a filter from a drop-down list. You could choose to remove a filter with either keyboard, mouse, or both. Also, the filtering, shown in Figure 5, would happen instantaneously in comparison to having to press a search button before actually filtering.

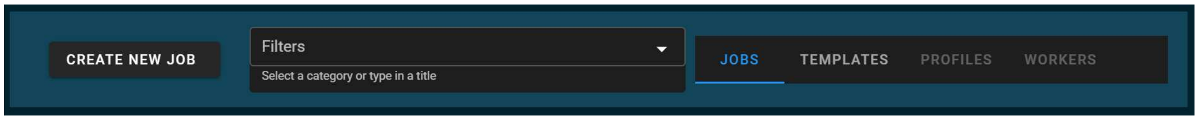
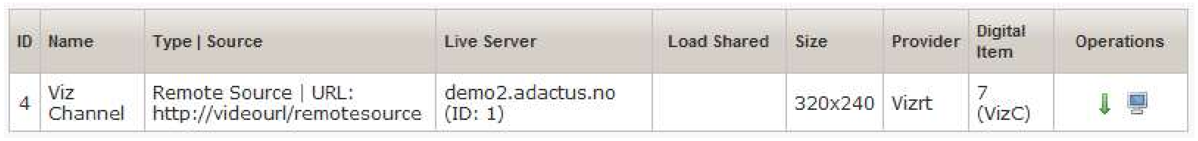


Figure 5: Control Panel in the new UI

4.2.2 Jobs and Templates List Panel

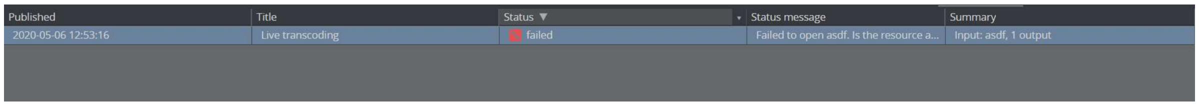
Coder UI simplified its previous version, the Viz One, of job listing significantly. The main reason was that hosting services like YouTube and Twitch did not have public streaming functionality like today. Instead you had to essentially create your own URL to live stream towards. Viz One had to support services like those mentioned above provide, but at a time before these hosting services were available. When Coder UI was developed, hosting services were established, so Vizrt could cut this complex part out of the software, greatly simplifying the website.

Coder UI, in our opinion, was still too cluttered with unnecessary information. Below you can see the progress between the styles in Figure 6, 7, and 8 by the number of columns.



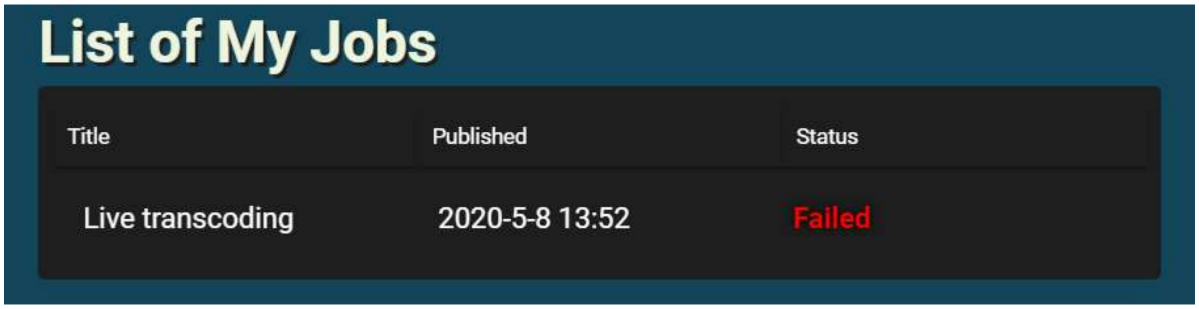
ID	Name	Type | Source	Live Server	Load Shared	Size	Provider	Digital Item	Operations
4	Viz Channel	Remote Source | URL: http://videourl/remotesource	demo2.adactus.no (ID: 1)		320x240	Vizrt	7 (VizC)	

Figure 6: Viz One solution to live jobs. This is one table of many.



Published	Title	Status ▼	Status message	Summary
2020-05-06 12:53:16	Live transcoding	🟥 failed	Failed to open asdf. Is the resource a...	Input: asdf, 1 output

Figure 7: List of jobs from the pre-existing solution

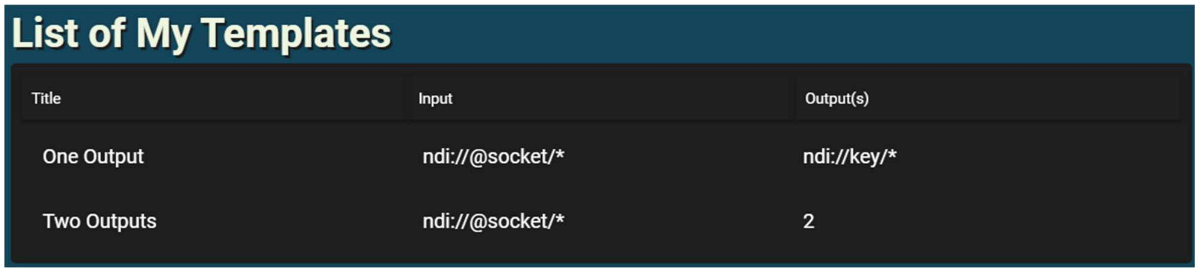


List of My Jobs

Title	Published	Status
Live transcoding	2020-5-8 13:52	Failed

Figure 8: List of jobs from the new UI

In addition to further simplifying the listing of Coder UI, seen in Figure 7, we have also added another list specifically for templates, shown in Figure 9, since it was requested. The fact that the website presents two different lists in a similar style is also the reason why the 'Title' column was moved completely to the left compared to Coder UI's placement as it is the only column that is synonymous to either list.



List of My Templates

Title	Input	Output(s)
One Output	ndi://@socket/*	ndi://key/*
Two Outputs	ndi://@socket/*	2

Figure 9: List of templates from the new UI

Since a job, and thereby a template also, can contain more than one output, we decided to have the content of the cells under 'Output(s)', in Figure 9, showing the number in case of multiple outputs and the output-URL in case of only one outputs.

Coder UI had icons to amplify its presentation of the status of a job. We too wanted to use these icons. However, we could not acquire these icons as we did not find them in Vizrt's source code nor were we able to copy these icons directly off their website.

Moreover, creating new ones would take too much time so we opted for using coloured text as a simple compensation.

The published column was also simplified and made clearer by removing the time in seconds as can be seen in Figure 7. This is a small difference that makes the reading slightly easier for the user.

The job list contains published and status, while the template list has input and output, and this is because the templates do not actually run, unlike actual jobs, rather they represent possible future jobs. Therefore, we show the possibility of a job having a specific input and outputs.

4.2.3 Creation Panel

Understanding that the original Viz One was more complicated than Coder UI, having multiple pages to create a single job. Coder UI had simplified this down to virtually only one modal.

Our creation panel design is functionally almost identical to the one Coder UI has, only ours is combined with template creation and job manipulation to make it more intuitive for the user to know where to look.

Audio mix is the part where we significantly altered Coder UI's design. Before you had to type in what you wanted manually, as seen in Figure 10, but we changed it to where you could press buttons to switch between 0's and 1's, as seen in Figure 11.

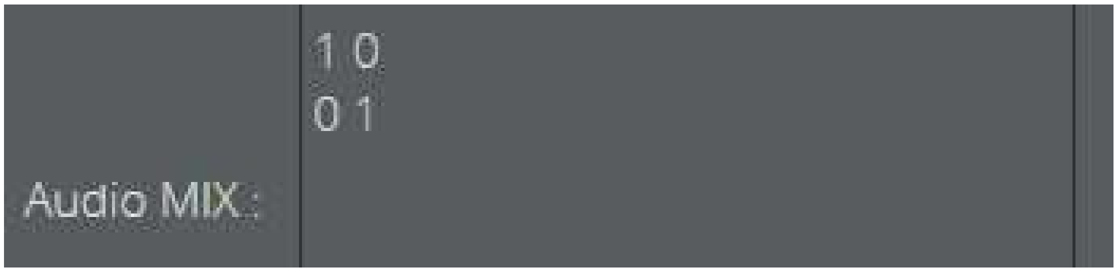


Figure 10: Audio Mix from Vizrt Coder UI

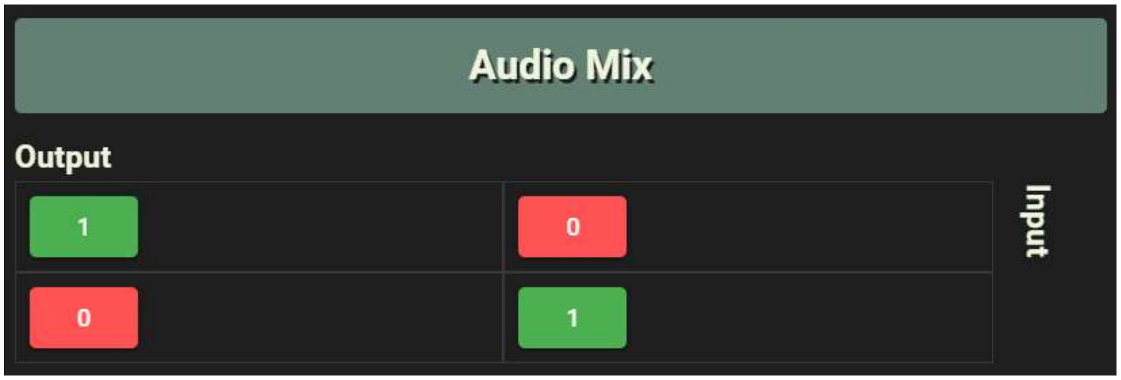


Figure 11: Audio Mix component

The input rows are defined by the user by either increasing or decreasing the amount via buttons seen in Figure 12.

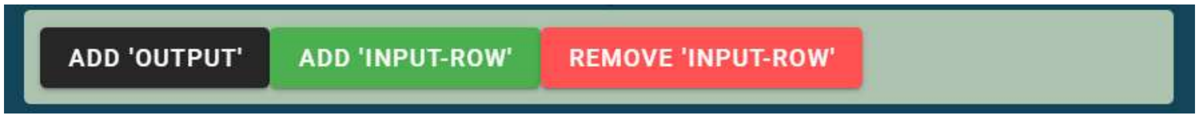
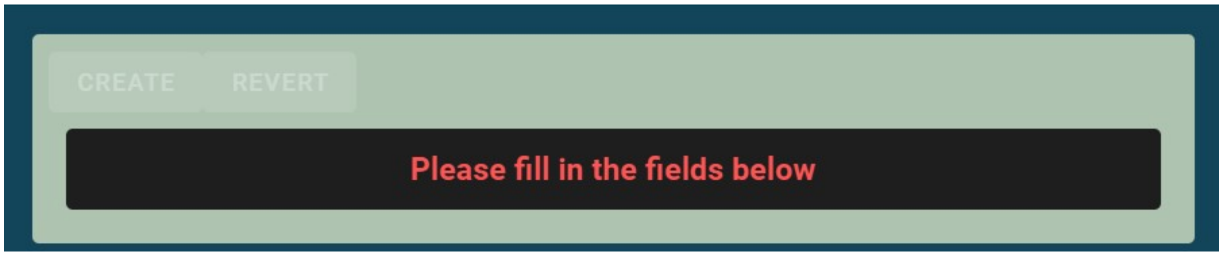


Figure 12: Component for adding new output along with changing input and output rows for audio mix.

Audio mixing was the one aspect of Coder UI we felt was too simple. Especially when considering that it was used globally with all outputs, preventing it from mixing audio based on the output channel. Meaning every output had to have the same audio in a sense. This was problematic when customers wanted to stream the same visuals using different languages for each output. Our design prevents this by forcing the user to mix individual audio for each output.

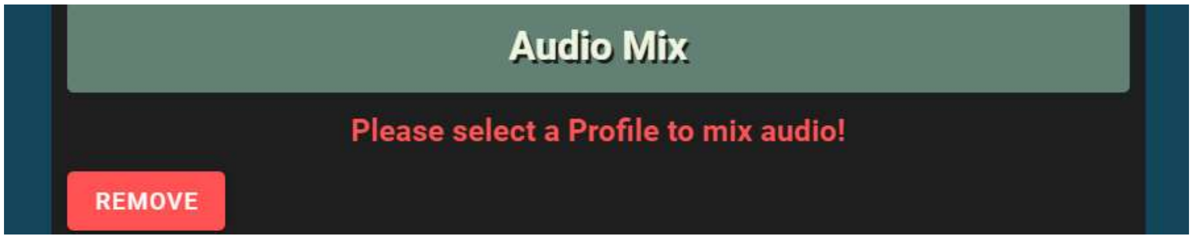
The only issue with this design, as shown in Figure 11, is not being able to choose any volume setting except for 0's and 1's. If we had more time, we could have implemented either a switch for advanced tuning or a more detailed mixing system that would offer more longevity. The problem is that if you were to mix a stereo input as mono it "will increase in loudness up to 6.02 dBSPL" (Wikipedia Contributors (4), 2020).

The error messages seen in Figure 13 and 14 were designed to provide the user with useful information that could be understood by the user, not to merely be interpreted by a developer. They were made to guide the user through the user interface.



Please fill in the fields below

Figure 13: First error when creating a new job in the UI



Please select a Profile to mix audio!

Figure 14: Second error when creating a new job in the UI

This guiding was also a reason for having the hints under the drop-down fields, like "Select one of your devices, like a web-camera", as shown in Figure 15. This is most prevalent in the creation panel, but we did implement an error modal that would display exceptions to the user, which could be considered more technical. Error modals are further explained in section 4.5.1.

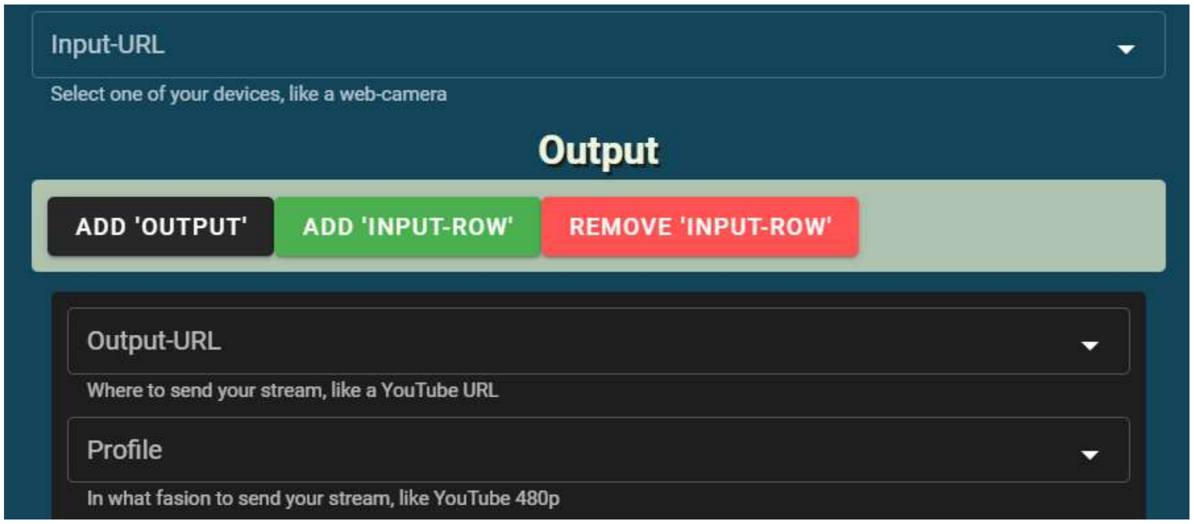

Input-URL
Select one of your devices, like a web-camera
Output
ADD 'OUTPUT'
ADD 'INPUT-ROW'
REMOVE 'INPUT-ROW'
Output-URL
Where to send your stream, like a YouTube URL
Profile
In what fasion to send your stream, like YouTube 480p

Figure 15: Hints when creating a new job in the UI

4.3 Website Architecture

The code structure of the website is hierarchical to the App.vue component, of which contains all the underlying components as leaves. The components of the site therefore have their positions determined by App.vue. However, we do use a global object to synchronize all components to the same jobs.

We call this object 'JobsList'. This object contains information about all available jobs, selected job, and filtering. Ideally this object would also contain information about templates to avoid repeating code, but we did not have time to refactor this. The reason for this being ideal is that templates require very similar methods to jobs as templates are essentially jobs.

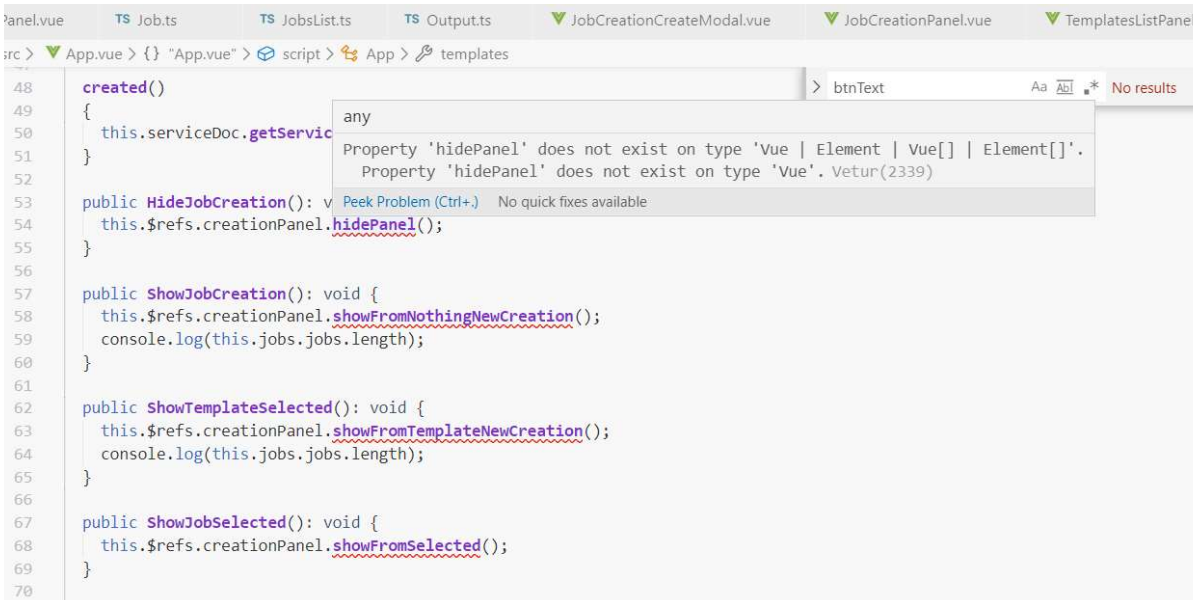
Another way we communicated between components was through Vue.js' callback; '$emit'. '$emit' is a Vue.js event that can be triggered from a child to a parent and thereby pass data upwards in the hierarchy. Yet using '$emit' deemed not perfectly suited as it was not well liked by the syntax of TypeScript. Even though it worked well enough, TypeScript sent warnings about the use of references of which we needed to further pass on the method call to another component sent by the emit, this is detailed in section 4.4 (Sundhu, 2018).

4.4 Vue.js Design Choices and Future Improvements

We did not have any design or code philosophy when it came to how to connect the visual elements, within the Vue.js template tags. A solution we could have done would have been to only use middle-ware methods and getters instead of directly accessing fickle variables within these tags. For instance, avoiding accessing a variable contained in a class fetched from the local script in the template tags. This would ensure longer lasting stability in the code, since variables referenced under template tags do not throw errors if they are wrongly written or do no longer exist unlike those referenced within the script tags. In other words, there is no IntelliSense with template tag connecting to the script tag, and the HTML-engine ignores and always attempts to fix human errors by allowing you to “omit certain tags (which are then added implicitly), or sometimes omit start or end tags, and so on. On the whole it's a "soft" syntax, as opposed to XML's stiff and demanding syntax” (Tali Garsiel, 2011).

In Figure 16 you can see we used Vue.js’ reference attribute to connect components together in App.vue. This was a design choice mistake as TypeScript falsely perceives these ‘ref’ attributes as a general type instead of its true type. It spewed out errors and warnings because it did not properly recognize the components class extension of Vue.

Indeed, we could have searched for a different solution that not only would have made TypeScript not throw warning at us, but also minimize the amount of hardcoding required by this solution. Nevertheless, the reason why we stayed with this solution was that it worked and due to lack of time.



```
created()
{
  this.serviceDoc.getServic
}

public HideJobCreation(): v
  this.$refs.creationPanel.hidePanel();
}

public ShowJobCreation(): void {
  this.$refs.creationPanel.showFromNothingNewCreation();
  console.log(this.jobs.jobs.length);
}

public ShowTemplateSelected(): void {
  this.$refs.creationPanel.showFromTemplateNewCreation();
  console.log(this.jobs.jobs.length);
}

public ShowJobSelected(): void {
  this.$refs.creationPanel.showFromSelected();
}
```

```
any
Property 'hidePanel' does not exist on type 'Vue | Element | Vue[] | Element[]'.
  Property 'hidePanel' does not exist on type 'Vue'. Vetur(2339)
```

Figure 16: False error in Vue.JS due to TypeScript

4.5 Modals

When the user would press the create job button, a modal would pop up, or technically unhide itself. Our thought process when deciding on this design was that we wanted to avoid altering any of the existing panels to make way for job or template creation confirmation buttons, and we also did not want to clutter the creation panel with these buttons either.

The layout of the creation modal was originally inspired by Vue UI's "Save as new preset" modal, shown in Figure 17, that comes up when you create a new project as it perfectly encapsulated part of our requirement regarding templates.

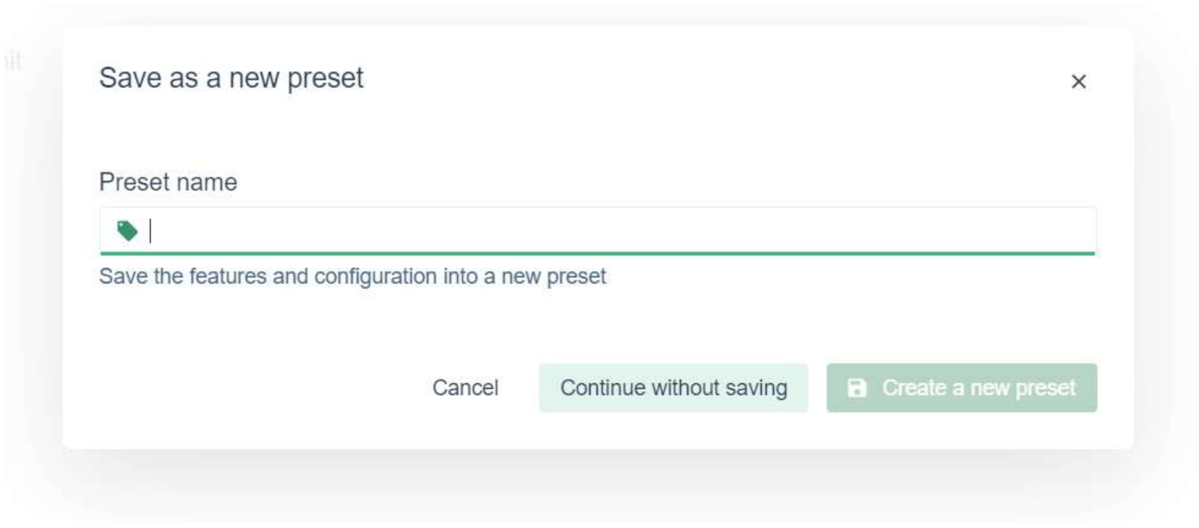


Figure 17: Creation Modal from Vue UI

The first implementation, seen in Figure 18 below, was therefore similar to the modal in Figure 17. It turned out less sleek than expected, but it worked just fine.

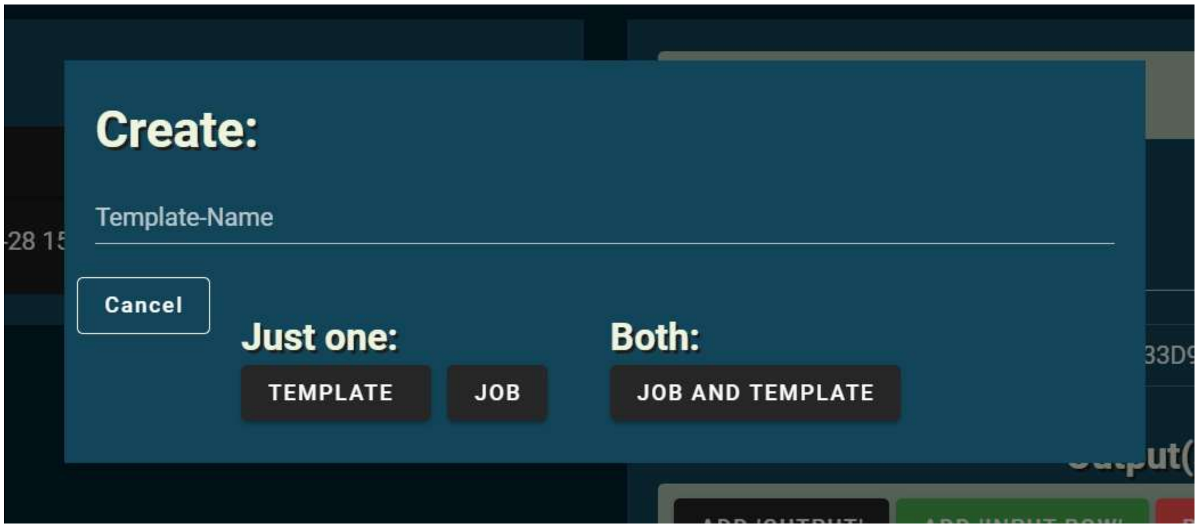


Figure 18: First creation modal

Later during development, we realized that zooming could not be done without ruining its style. This was fixed in the new solution, as seen in Figure 19, by locking the size of the modal, not to the percentage of the window, but dynamically to the width needed by its contained elements. Additionally, we agreed that there was an unnecessary amount of clutter in this implementation. Therefore, we combined template title with title input from the creation panel and changed the buttons with two simple switches, as seen in Figure 19. These switched would also remember the last changes made within a session.

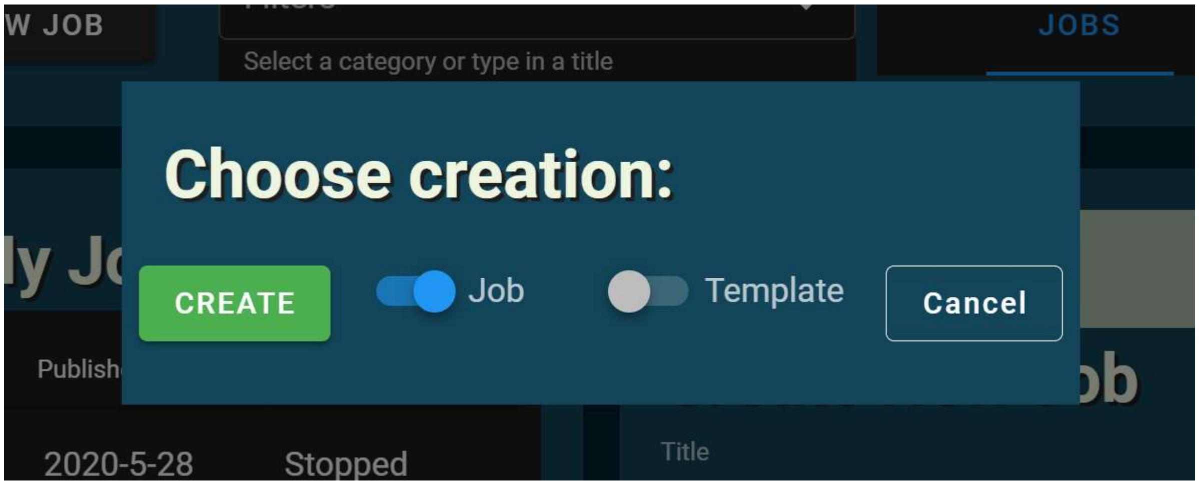


Figure 19: Final creation modal

We also decided to help the user understand this title change via a tooltip that pops up when hovering over the template switch, as seen in Figure 20. Another minor feature was that the ‘create’ button would now be disabled in case of both switches being turned off, which is also shown in Figure 20.

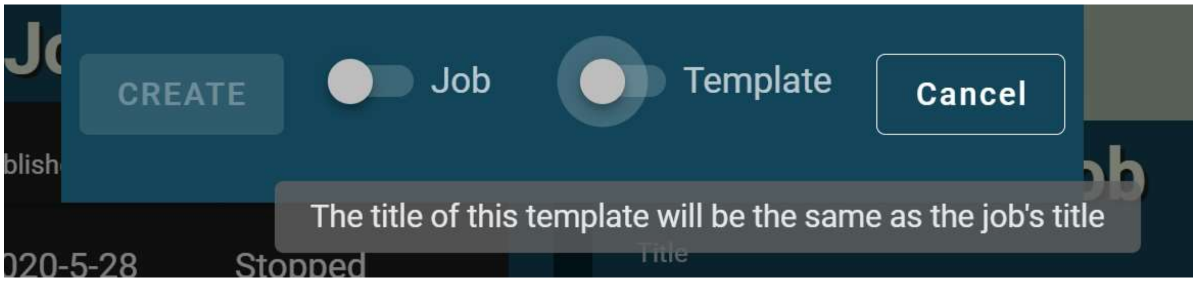
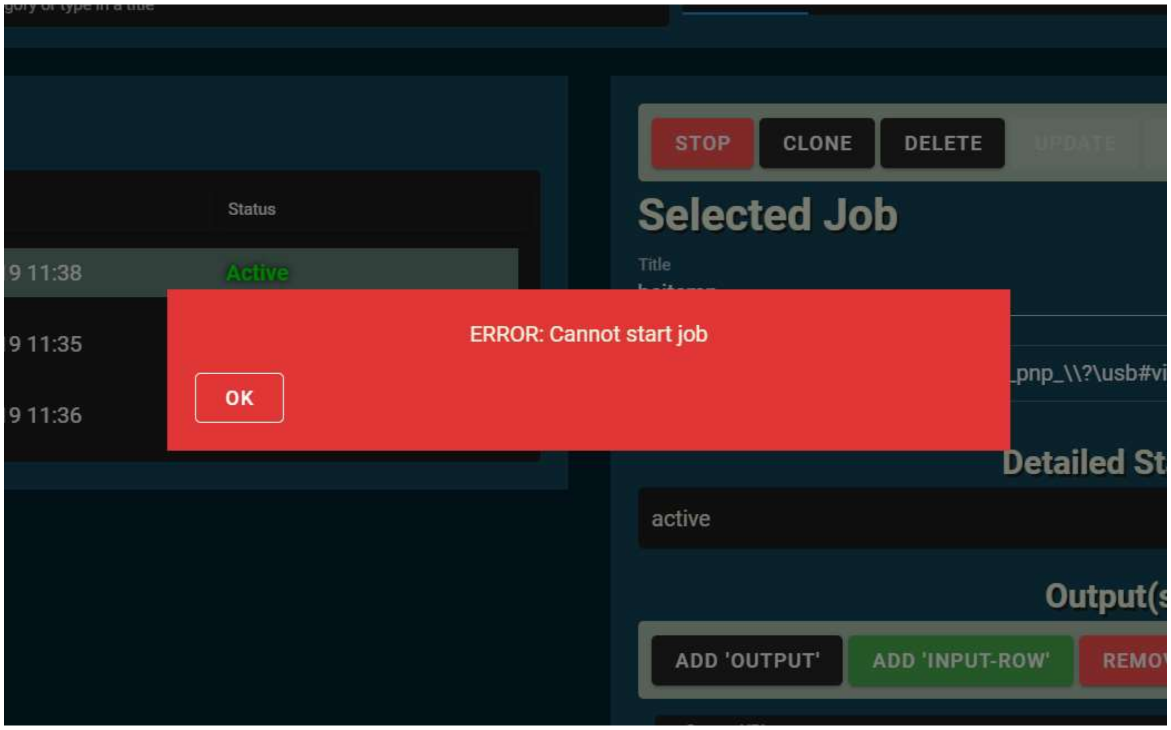

CREATE
Job
Template
Cancel
The title of this template will be the same as the job's title
2020-5-28
Stopped
Title

Figure 20: Additional creation modal features

4.5.1 Error Messages

As mentioned, some errors were displayed through modals. The action the user could then take would then be to click OK as seen in Figure 21. This would close the modal. There is no fancy interaction between the user and this modal, rather it can be interpreted as a mere messenger.

To simplify throwing error messages from anywhere in the code we decided to make the modal display the errors statically. This meant that the modal could be called upon by any file in the code easily.



ERROR: Cannot start job

Figure 21: Error Modal that would appear if an error had occurred.

4.6 Connection to the API

Our website has a client/server architecture where Coder is the server and our website acts as a client. Coder contains several Atom collections with information about jobs, templates, profiles, and workers. These collections work similarly to a database, where you can create, delete, and update entries to these collections. This communication is done through HTTP REST API for all respective collections. However, since the job collection would update quite frequently, our website would function as a client to the STOMP broker that is hosted on Coder. This allowed us to monitor these jobs without the need of a REST call for each update which put less strain on the web application. However, it still required handling of the messages from the broker.

As these collections work in XML, we adopted DOM (Document Object Model) for parsing of XML both to and from Coder. We came to this conclusion because we were most familiar with DOM, and it fit well for our project. We also looked at other choices such as XPATH, however it seemed too cluttered and confusing to work with together with Vue.js and TypeScript for our liking.

When working towards creating objects out from these parsed XML files mentioned earlier, methods were created that would focus on parsing Atom objects. Therefore, a general class 'AtomObject' was created to allow inheritance to classes such as jobs, templates, and profiles. This allowed us to easier access variables within these objects, as variable names and functions inherited would be equal for all Atom objects.

A 'Job' class was used for both jobs and templates due to their similarity of variables. This allowed for general REST functions to start, stop, update, and delete, with the disabling of start and stop for templates. These functions were contained within the object itself, to ensure easy access to its own variables, most importantly the HTML link to itself in the Atom collection.

While these choices allowed us to create and access data, there was also a need for a way to access external URLs for Coder. In this case, a class 'ServiceDocument' was created to keep control over links of the collections across our entire application. This allowed us to retrieve the URLs needed for the collections when the user opened the application, and also opening up for external URLs, as the application would search for URL parameters when first starting. Essentially, this means that the web program could access other machines that contain Coder and control their workers for live transcoding.

In the end, there was also a need for polling of workers and profiles. The reason for this was that you could not add workers or profiles in our implementation. Since workers and profiles must be added externally, polling was put in place to check and update the worker and profile objects, to match the Coder Atom collections.

5 EVALUATION(S)

5.1 Evaluation Method

5.1.1 Evaluation by Trial and Error

During development instead of evaluating through unit testing and double-checking code before compiling, we wrote the necessary code and immediately tested if it were correct or if the Vue.js compiler would throw an error message. Unit testing was unnecessary as testing would require visual confirmation via NDI Monitor Studio, and since unit testing “is not as easy […] when a major function of the method is to interact with something external to the application” (Wikipedia Contributors (5), 2020).

If the code seemed to work as intended and no error message were to show up, then we would inform the others in the group about an incoming push, if then the coast was clear then we would commit the changes to our GitLab repository.

5.1.2 Evaluation by User-Testing Through the Task-Provider

Through Teams meetings we would show the task-provider how far we had come with development, ensuring that we were on to the right tracks.

We did this by screen sharing in Teams changes we made whenever we had a meeting and getting and asking questions about the site as well. Such as what the site should contain. We would then take notes and attempt to address these issues at hand till the next meeting.

Additionally, the code was available for download so that anyone with access to the repository could test the site locally. One of the task-providers periodically did this and gave feedback in terms of approval and disproval of different parts of the site.

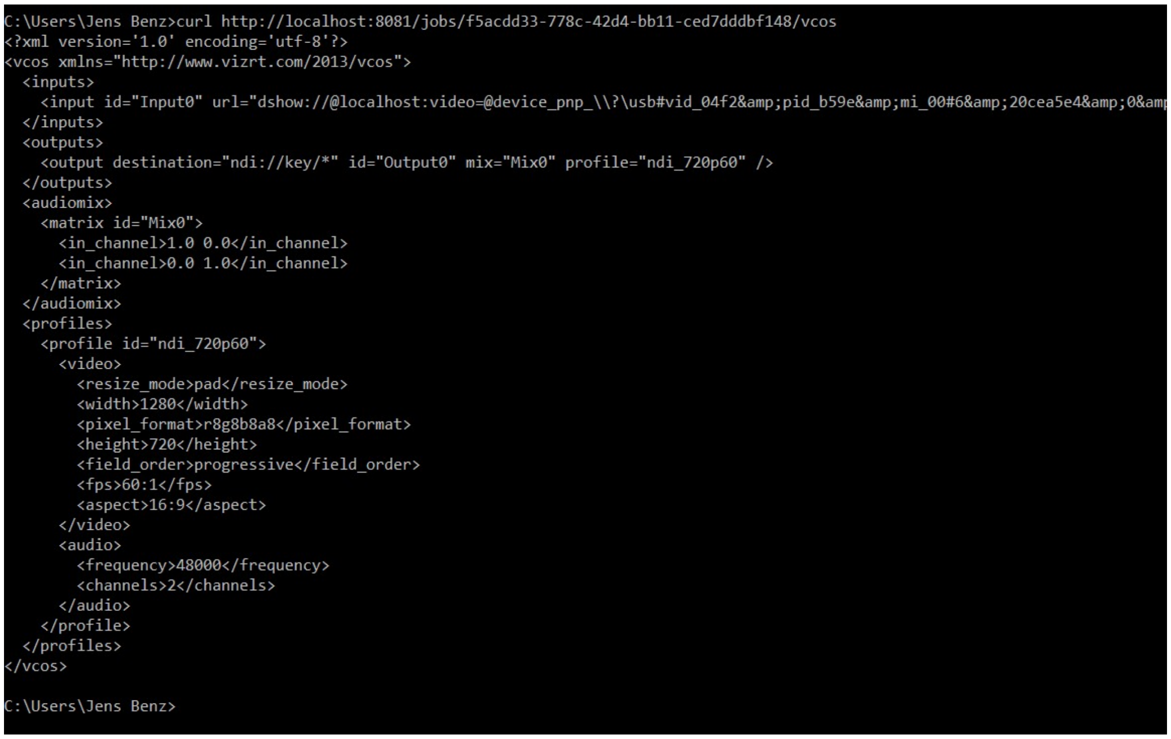
The task-providers would, however, not give any feedback directly relating to the code itself. Feedback would instead be, for example, suggestions on how communication with the API should be done.

This type of evaluation also opened for a two-way feedback communication. That meant that, in addition to them giving feedback to us, we could also give feedback on Coder, report bugs, or suggest necessary changes needed for Coder.

The evaluation was practiced through having the task-provider compare our solution with their requirements for the project. A failed evaluation would be where one or more requirements were not met.

5.1.3 Evaluation by Testing of Robustness

Testing by cURL-ing, as shown in Figure 22, the responses and requests between the site and the API, seeing if they correspond properly with the site and if the requests are correctly communicating with Coder. Also, testing the site by using it as a user would, creating streams and watching them, seeing if they coincide with what is expected. For instance, audio plays in both the right and left audio channel, the video streams with right resolution to all the right outputs, and that this fits with what the UI of the site tells.



```
C:\Users\Jens Benz>curl http://localhost:8081/jobs/f5acdd33-778c-42d4-bb11-ced7dddbf148/vcos
<?xml version='1.0' encoding='utf-8'?>
<vcos xmlns="http://www.vizrt.com/2013/vcos">
  <inputs>
    <input id="Input0" url="dshow://@localhost:video=@device_pnp_\\?\usb#vid_04f2&amp;pid_b59e&amp;mi_00#6&amp;20cea5e4&amp;0&am
  </inputs>
  <outputs>
    <output destination="ndi://key/*" id="Output0" mix="Mix0" profile="ndi_720p60" />
  </outputs>
  <audiomix>
    <matrix id="Mix0">
      <in_channel>1.0 0.0</in_channel>
      <in_channel>0.0 1.0</in_channel>
    </matrix>
  </audiomix>
  <profiles>
    <profile id="ndi_720p60">
      <video>
        <resize_mode>pad</resize_mode>
        <width>1280</width>
        <pixel_format>r8g8b8a8</pixel_format>
        <height>720</height>
        <field_order>progressive</field_order>
        <fps>60:1</fps>
        <aspect>16:9</aspect>
      </video>
      <audio>
        <frequency>48000</frequency>
        <channels>2</channels>
      </audio>
    </profile>
  </profiles>
</vcos>

C:\Users\Jens Benz>
```

Figure 22: Using cURL to detail a job entry

5.2 Evaluation Results

5.2.1 Result from Trial and Error

We managed to quickly improve our website using this technique iteratively. It did help that our group was small and communication between the group members was easy. We would not recommend our method in a group much larger than ours as it would greatly complicate development. It would at least be recommended to utilize branches if you were to use this evaluation.

One major part of this evaluation was that we learnt that one should not push code containing bugs to the repository. Once you do this you tend to affect the others negatively. However, of course this depends highly on how large the bugs are. Small and insignificant bugs may be ignored when pushing. This might be seen upon as hypocritical, but there is too often the case that bugs go by unnoticed without harming much.

This type of evaluation is virtually unavoidable in the realm of programming, so it becomes difficult avoiding one of its major drawbacks; creating unnecessary bugs due to fast iterations. However, fast iteration could also be considered an upside as it makes it easier to notice errors quicker. Therefore, though unavoidable, this evaluation was helpful for us, especially since the technologies used were new to us when we started off.

5.2.2 Result from User-Testing Through the Task-Provider

The task-provider rarely complained at design decisions but gave tips and ideas to successfully create what the task required. Our understanding and choices were, therefore, often approved.

Feedback received by this evaluation was highly rewarding as it set us on a definite course to the result our task-providers wished for, and because they noticed faults and quirks that we would otherwise overlook.

Eventually they orally evaluated that the solution had successfully met their requirements.

5.2.3 Result from Testing of Robustness

Most importantly, testing the website in ways that does not directly involve developing the site unveiled bugs that otherwise might have gone unnoticed. After addressing these bugs and confirming that they were truly fixed we had made the website less error-prone than it would have otherwise been.

The reason why some of these bugs might have gone unnoticed is because Coder could accept requests that were conclusively incorrect. For instance, putting an XML-document containing a new job without our code attaching the outputs' audio matrix. Ideally, these faults would not be made in the first place, but realistically the only way we could properly notice faults like this would be by watching a job stream or cURL-ing network requests and responses.

Though testing can be difficult and might not give any help, but when it does it is worth the effort. It is recommended to test your software at least before release to ensure that your product will not be a flop.

For our purposes, this evaluation ended with us fixing or reporting all related bugs until we no longer could spot any new ones. If we had a designated team or more group members, we may had noticed more bugs and subsequently made the website more robust.

6 DISCUSSION

Our task had limitations due to the project being based on a pre-existing implementation that again took use of existing solutions. When we look at the approaches we took regarding communication, it would have been more difficult to find a solution using other technology than those chosen related to communication with the API. The reasoning behind this statement is that the pre-existing implementation already utilized a STOMP broker. Since we could not change the backend, the only options we had was to either only use REST communication or use the existing messaging broker. If we chose to not use the existing broker, it would force our website to send REST requests every time something updated. This would drastically increase load times due to the repeated requests.

This led us to investigate Coder's STOMP broker and how its communication functioned, to then translate and handle the messages received in our own way. This could have been avoided if there had been a possibility to set up the WebSocket to handle all messages from the transcoder, instead of just the transcoding jobs.

When looking at our UI approach, the consequences are second to none for the general user. This is because we were virtually free to design our website from the requirements. Eventually, this led us to request changes in the API from our task-provider, like template storing, and opened for general discussion between the group members. A consequence in our UI design is that having many outputs in a job will force the user to scroll down. This might not be ideal; however, it is just a small inconvenience for the user. Another is lack of lower level information, like a job log, which becomes a downgrade from Coder UI for more advanced users.

Because part of our requirements was to use Vue.js, TypeScript and the STOMP broker we had to continually adapt our approach on solving this project. An example of an obstacle is the combination of TypeScript and Vue.js. The combination gave us unexpected issues in the code and there was an absence of help and discussion regarding the combination of these two technologies. Often, this led us to issues as shown in Figure 16 in section 4.4. We solved many these issues by finding workarounds and accepting that even if we had errors, like in Figure 16, it would function as expected. The combination of these technologies has proven to be a hinderance, though not necessarily detrimental for the project.

We had initially planned to add in Vuex for reactivity and state-storing of our objects. This were not implemented due to lack of time and research. Instead our solution became to pass objects down from the main component App.vue to the other underlying components. This allowed these components or objects to access and update data similarly well to what Vuex would provide.

Assuming we did implement Vuex, it could have improved our code, though there would not be any visual improvement at runtime. It would however make it easier for future programmers. If we had more time and we were to refactor our code, we would profoundly consider replacing this part of our solution with Vuex.

7 CONCLUSIONS AND FURTHER WORK

The goal of this project was to develop a new web application based on a pre-existing solution. The web application was to mainly use the Vue.js framework and allow for users to create, manipulate, and edit jobs by communication with a transcoder. The new functionality our implementation focused on was templates, meaning any user could store jobs for future use. Additionally, we focused on establishing pleasant and modern user experience regarding our UI solution.

We recognize that our project can potentially be useful for customers of Vizrt in addition to developers internally in Vizrt. The project might be both useful for Vizrt and new groups as Vue.js is increasingly more common to be used as a framework for web applications. Our project will also allow Vizrt to showcase a product utilizing new functions like templates to their customers, and perhaps test to see if such a functionality satisfies the market.

Supposing that we were to have continued our work, it would have been of high interest to add other functionalities that we dropped due to time constraints. Though not being a priority nor a requirement, implementing additional functionality like job log or giving the user full control over the transcoder would have been interesting. We could have solved the issue where the user may have to, before using our application, use pre-existing software to properly set up their transcoder.

We could have solved this project more efficiently if we were not constrained into using their existing WebSocket technology. The reasoning behind this statement is that more time than necessary was spent researching their WebSocket. Assuming that we were told early on which technology to adopt, we could have gained more time to focus on other aspects of this project. Instead of having to research and test all alternatives regarding communication with the API, we could have actualized better solutions using technologies we originally wanted to implement, like Vuex.

Another issue we would address would be syncing number of output audio columns, as we ended up hard coding two columns in every audio mix matrix instead of basing the number on a profile’s referenced number of output channels. This would have provider a more accurate audio mixing experience for the user.

Additionally, regarding the audio mixing, we would want to address the issue concerning the Pan law as mentioned in section 4.2.3. Our solution to this issue would be to allow the user to choose other volume values than 0’s and 1’s. This could be, for instance, through a slider or an audio knob. Possible fault regarding this law would then no longer lay upon the implementation.

To sum up, based on feedback and testing we can conclusively say the goal has been met. The web application functions well and meets the task-giver’s requirements.

8 LITERATURE

Hanson, J., 2014. *Http Long Polling.* [Online]
Available at: https://www.pubnub.com/blog/http-long-polling/
[Accessed 1 June 2020].

IETF (1), 2005. *The Atom Syndication Format.* [Online]
Available at: https://tools.ietf.org/html/rfc4287
[Accessed 1 June 2020].

IETF (2), 2007. *The Atom Publishing Protocol.* [Online]
Available at: https://tools.ietf.org/html/rfc5023
[Accessed 1 June 2020].

Kilbride-Singh, K., 2019. *Websockets vs Long Polling.* [Online]
Available at: https://www.ably.io/blog/websockets-vs-long-polling/
[Accessed 1 June 2020].

Mesnil, J., 2012. *STOMP Over Websocket.* [Online]
Available at: http://jmesnil.net/stomp-websocket/doc/
[Accessed 1 June 2020].

Mozilla (1), n.d. *WebSockets API.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[Accessed 31 May 2020].

Mozilla (2), n.d. *What is Javascript.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript
[Accessed 1 June 2020].

NDI.tv, n.d. *Tools.* [Online]
Available at: https://ndi.tv/tools/
[Accessed 1 June 2020].

REST API Tutorial, n.d. *Using HTTP Methods for RESTful Services.* [Online]
Available at: https://www.restapitutorial.com/lessons/httpmethods.html
[Accessed 1 June 2020].

Sundhu, S., 2018. *How to emit data in Vue: Beyond the Vue.js Documentation.* [Online]
Available at: https://www.telerik.com/blogs/how-to-emit-data-in-vue-beyond-the-vuejs-documentation
[Accessed 1 June 2020].

Tali Garsiel, P. I., 2011. *How Browsers Work - Context Free Grammar.* [Online]
Available at:
https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/#context_free_grammar
[Accessed 1 June 2020].

Visual Studio (1), n.d. *Getting Started.* [Online]
Available at: https://code.visualstudio.com/docs
[Accessed 1 June 2020].

Visual Studio (2), n.d. *IntelliSense.* [Online]
Available at: https://code.visualstudio.com/docs/editor/intellisense
[Accessed 1 June 2020].

Vizrt (1), n.d. *About Us.* [Online]
Available at: https://www.vizrt.com/vizrt
[Accessed 1 June 2020].

Vizrt (2), n.d. *Coder Local Documentation.* [Online]
Available at: Not Accessible to general users
[Accessed 1 June 2020].

Vue.js (1), n.d. *Introduction.* [Online]
Available at: https://vuejs.org/v2/guide/#What-is-Vue-js
[Accessed 1 June 2020].

Vuetify, n.d. *Vuetify.* [Online]
Available at: https://vuetifyjs.com/en/
[Accessed 1 June 2020].

w3schools (1), n.d. *HTML intro.* [Online]
Available at: https://www.w3schools.com/html/html_intro.asp
[Accessed 1 June 2020].

wc3schools (2), n.d. *CSS.* [Online]
Available at: https://www.w3schools.com/css/
[Accessed 1 June 2020].

Wikipedia Contributors (1), 2020. *Polling (Computer science).* [Online]
Available at: https://en.wikipedia.org/wiki/Polling_(computer_science)
[Accessed 1 June 2020].

Wikipedia Contributors (2), 2020. *Light on dark color scheme.* [Online]
Available at: https://en.wikipedia.org/wiki/Light-on-dark_color_scheme
[Accessed 1 June 2020].

Wikipedia Contributors (3), 2020. *TypeScript.* [Online]
Available at: https://en.wikipedia.org/wiki/TypeScript
[Accessed 1 June 2020].

Wikipedia Contributors (4), 2020. *Pan Law.* [Online]
Available at: https://en.wikipedia.org/wiki/Pan_law
[Accessed 1 June 2020].

Wikipedia Contributors (5), 2020. *Unit Testing.* [Online]
Available at: https://en.wikipedia.org/wiki/Unit_testing
[Accessed 1 June 2020].

9 REFERENCES

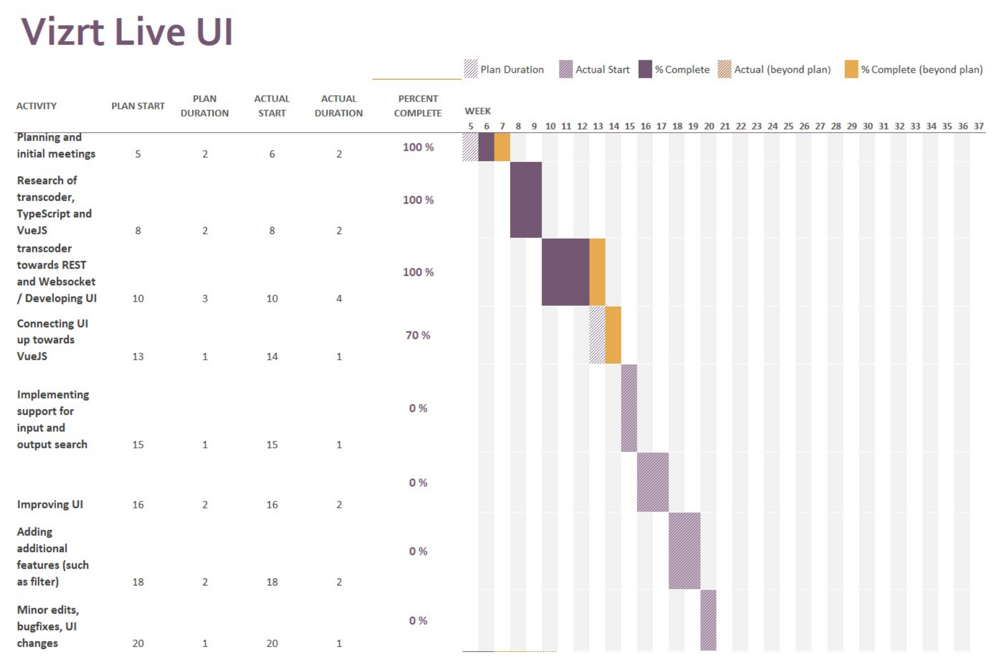
10 APPENDIX

10.1 Risk list

Situation	P	C	R	Measures
Global pandemic	5	3	15	Communicate equally often with chatting or meeting applications on our individual computers. Share screen, files and pictures whenever necessary.
Not reaching our goals	2	5	10	Plan out and do tasks based upon the goals we ourselves set forth or goals we are given by the task-provider.
Lack of competence	3	3	9	Take time and focus on learning technologies necessary to achieve requirements set for the project.
Lack of motivation	2	4	8	Distribute responsibility so that every member of the project group has at least one concrete and understandable mission continuously.
Misunderstanding requirements and specifications	2	4	8	Write out, check and ask questions thoroughly to ensure that we are on the right tracks.

10.2 GANTT diagram

Early on in the process of planning we decided on a GANTT diagram shown in Figure 23. This was edited into a second GANTT diagram due to unforeseeable issues, such as the Covid-19 pandemic and minor delays in updating Coder for our project. We made the choice to make a new GANTT diagram better reflecting our status, which is shown in Figure 24.



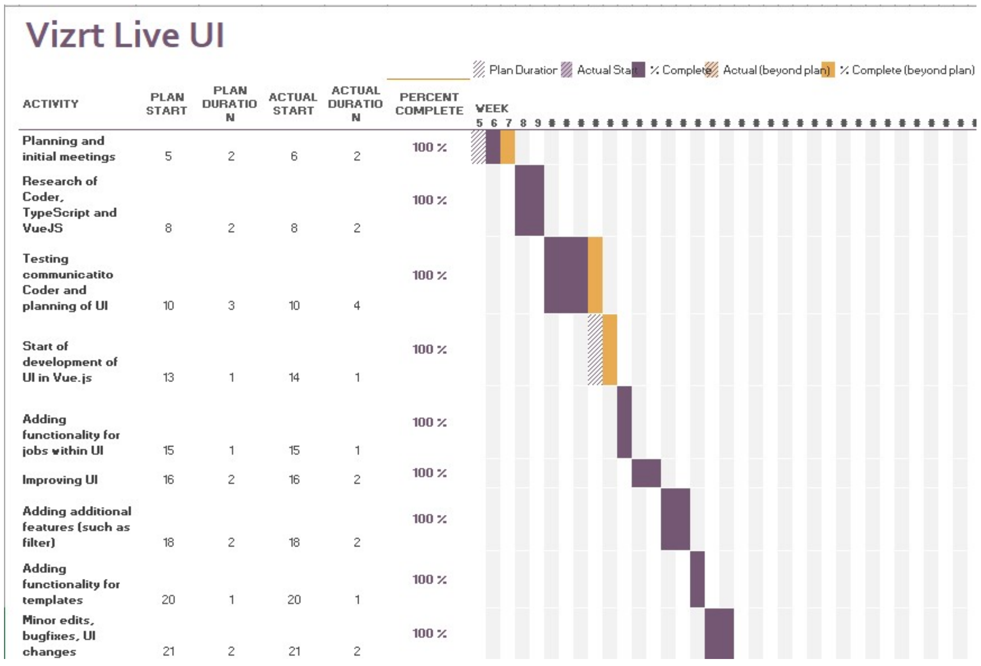
Vizrt Live UI

▨ Plan Duration ▨ Actual Start ■ % Complete ▨ Actual (beyond plan) ■ % Complete (beyond plan)

ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Planning and initial meetings	5	2	6	2	100 %
Research of transcoder, TypeScript and VueJS	8	2	8	2	100 %
transcoder towards REST and Websocket / Developing UI	10	3	10	4	100 %
Connecting UI up towards VueJS	13	1	14	1	70 %
Implementing support for input and output search	15	1	15	1	0 %
Improving UI	16	2	16	2	0 %
Adding additional features (such as filter)	18	2	18	2	0 %
Minor edits, bugfixes, UI changes	20	1	20	1	0 %

WEEK 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37

Figure 23: Early Gantt Diagram



Vizrt Live UI

▨ Plan Duration ▨ Actual Start ■ % Complete ▨ Actual (beyond plan) ■ % Complete (beyond plan)

ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Planning and initial meetings	5	2	6	2	100 %
Research of Coder, TypeScript and VueJS	8	2	8	2	100 %
Testing communicatito Coder and planning of UI	10	3	10	4	100 %
Start of development of UI in Vue.js	13	1	14	1	100 %
Adding functionality for jobs within UI	15	1	15	1	100 %
Improving UI	16	2	16	2	100 %
Adding additional features (such as filter)	18	2	18	2	100 %
Adding functionality for templates	20	1	20	1	100 %
Minor edits, bugfixes, UI changes	21	2	21	2	100 %

WEEK 5 6 7 8 9 #

Figure 24: Final Gantt Diagram

10.3 Vocabulary List

Job	A live transcoding transmitted through Coder to outputs.
Input / Output	Input provides a data stream to be sent through the API to the output. Output is a video streaming platform. For instance, a web camera and Twitch.
Coder UI	In house and older version of the website we have made. In other words, another live transcoding website based on Vizrt's API.
Vizrt	Task-provider and a company that creates content production, management, and distribution tools for the digital media industry
API	"Application Programming Interface"
VCOS	"VCOS is a specification format for video transcoding" (Vizrt (2), n.d)
Coder	API supporting production of VODs (Vizrt (2), n.d).
VOD	"Video on Demand"
STOMP	"Simple text-orientated messaging protocol" Over WebSocket. Used to communicate with Coder.
UI	"User Interface"
Template	A representation of a Job that can be utilized to create a new job in the future without having to repeatedly type the necessary information of a job each time.
Atom Collection	An Atom collection is a feed document that lists URLs and information surrounding Atom entries.
Vuex	Vuex is a centralized storage for all components in an application, that allows for reactivity.
Modal	Elements that are placed on top of everything else in an application.
Proxy	A middleware computer system that reads and forwards requests.
Transcoding	To convert a format to another.
IntelliSense	"IntelliSense is a general term for a variety of code editing features including: code completion, parameter info, quick info, and member lists" (Visual Studio (2), n.d).