Design & Development of a Full-stack ERP Marketing Web App

Héctor Gabriel Suero Martínez

Haaga-Helia University of Applied Sciences

Bachelor's Thesis

2022

Business Information Technology

Author(s)
Héctor Gabriel Suero Martínez

Degree
Business Information Technology

Report/thesis title
Design & Development of a Full-stack ERP Marketing Web App

Number of pages and appendix pages
40

The software project consists of a full-stack application for an enterprise resource planning system used by car inspection companies; it is used for the creation of marketing material and for the execution of campaigns.

Full-stack development is concerned with the presentation, business logic and data layers of web applications, and the stack of technologies on which they are built. The thesis documents the design and development of the application.

UML is an object-oriented modelling language used in system analysis and design for creating different types of models that capture behavioral and structural details. Requirement analysis was done for the Web App starting with a domain analysis to understand the context of the problem to be solved. Once pertinent concepts were identified and documented, behavioral models were created based on the domain analysis and data collected.

The database design goes through the phases of conceptual, logical and physical design. In the first two phases' models were created for the data and their relations that progressed in detail. The anomalies that result from data redundancy were explored and its solution—normalization—was implemented and results discussed. It is not always possible to remove all data redundancy completely because it may be required to some degree by the business' requirements as was the case in this project.

The software development of the back end is documented by showing samples of code for services, custom built middlewares for enforcing security, error handling and logging HTTP request information to Azure Log Analytics workspaces. The front-end main components were described such as the Imagebank & Print Editor. RTK Query was shown to simplify the state management when making requests to the back-end REST API.

Unit testing grants flexibility to code since any change introduced can be easily verified to conform to expected behavior. Without testing, making changes to code is a risk of introducing bugs. The Jest testing framework and its implementation were explained with samples.

The Web App was built and deployed automatically using Azure Pipelines. A YAML pipeline definition sample was shown and described in detail. The conclusion is a reflection of the challenges faced, solutions and suggestions of how to improve system development with Test Driven Development.

Keywords
full-stack, database design, requirement analysis, express, redux toolkit, azure pipelines

Abstract

Table of contents

1 Introduction

This thesis project consists of the design and development of a full-stack marketing web application for a new enterprise resource planning system (ERP). The commissioning company Alpha in conjunction with Beta requested an application tailored for Beta's members: an association of car inspection companies. The envisioned application's functionality spans from creation of different types of marketing material to the execution of the campaigns that will allow them to manage their marketing process.

1.1 Background

The project on which this thesis is based was initially started as part of a Multidisciplinary Software Project course held in Haaga-Helia during the autumn of 2020. The ERP system of which the marketing tool is part of was not completed at the time of initial development. Subsequently, the team of students that undertook the project were tasked with the creation of a minimum viable product (MVP) that was constrained in scope. This version of the product relied on test data provided by a dummy RESTful application programming interface (API) that provide campaign data mimicking the ERP own API, and whose primary functionality was to create text-based marketing material that was assigned to campaigns.

At the time in Finland about 98% of all inspection companies used Muster, a vehicle inspection system by the software company Eta. Theta—a private equity firm—purchased Eta, and in addition it acquired Iota, the largest player in the industry and competitor of members of Beta. Theta acquiring Eta meant that Beta members relied on a system critical to their business, yet in the hands of their competitor. Since Muster's development had stalled partly due to out-of-date architecture and its monopoly of the market, it created an opportunity for an alternative to emerge.

At the request of its members who were concerned about business growth, Beta set out to develop a new ERP system whose copyright and management would remain under its control. In order to finance the project and administer the system, a new company—Alpha—was established and is jointly owned by Beta and a majority of its members.

When it comes to marketing in the context of car inspection companies, an inspection reminder is of vital importance because it represents a window of opportunity where services can be provided to new customers as well as previous ones. Car ownership frequently changes (365 000/year) resulting in inspection companies' customer registry

quickly becoming outdated. Epsilon, the Finnish transport and communication agency has up-to-date information on all Finnish vehicles and their due inspection times, and therefore, it is a key entity in the success of the project. Its contribution will allow Alpha to target the correct vehicle registrant in a continually shifting landscape.

The marketing web application was initiated because Alpha's ERP system lacks but envisions a new reliable and efficient marketing tool; one that integrates to their inspection system and that empowers its clientele to expand their customer base.

1.2 Aims and objectives

The aim of the project is first and foremost to document the design and development of the product as it is to date. The product is a marketing tool that enables the creation of marketing material for SMS, e-mail and print. The SMS and printed material will be compiled and sent by a marketing company which offers those services, only the e-mail will be sent from the marketing web app. Due to the time constraints and dynamics with the stakeholders involved, the print feature has been deprioritized.

The objectives are as follows:

- Conduct an analysis of the problem the product is trying to solve, to examine its context, its constraints, and to model the way the solution ought to behave in order to fulfil its purpose.
- Design models of the data and its relations so that it can follow the business rules. To implement the design and produce a database that is well structured, efficient and optimized.
- Document a visual and descriptive break down of both back-end and front-end.
- Create pipeline that builds, tests and deploys the solution to the Azure Cloud.
- Come up with suggestions for improvement of the process and product.

2 Theoretical Framework

The theoretical framework on which this thesis work is based on are topics relevant to the various stages the project goes through during the system development life cycle (SDLC), particularly system analysis, design, development, testing, and delivery of the software product. The first theme of focus is concerned with the language used for eliciting requirements and which facilitates the creation of the system's blueprint design. Secondly, the framework used for managing the project's development life cycle, how to minimize defects through test driven dev, and lastly, continuous integration and continuous delivery of production-ready code.

2.1 Unified Modelling Language

The Unified Modelling Language (UML) is a graphical modelling language used to do visual system analysis, design and deployment using an object-oriented approach. The organization behind the standard is Object Management Group (OMG) which first adopted it in 1997 (Ashrafi & Ashrafi 2014, 54). It was later published as an ISO standard by the International Organization for Standardization in 2005 (International Organization for Standardization 2005).

There are three important activities that take place when developing a system: analysis to determine the requirements, the design to translate those requirements into a system blueprint, and finally the implementation where the solution is constructed. There are also three key roles involved in the process, and each one has their own perspective of the system. UML allows for these different views (models) of a system each varying in the amount of detail or generalization that is conveyed. The business owner views the system in terms of its business function (conceptual view) and in its most abstract sense. The architect is tasked with designing the solution and so, views the system in terms of its components and their logical relationships (logical view). Lastly, the builder focuses on the implementation, and views the system at the lowest in level of abstraction: its blueprint (physical view). UML diagrams can be divided into three categories: behavioural which models interactions, structural which focuses on the components themselves, and dynamic, how the systems components interact with one another and external entities to conform to the expected behaviour of the system. (Ashrafi & Ashrafi 2014, 54-55)

UML has four capabilities (Ashrafi & Ashrafi 2014, 55-56):

- Visualization of system components along with their relationships and interactions.
- Specification through models for analysis, design and implementation
- Construction of code from models through Object-Oriented language compatibility

– Documentation through modeling

2.2 JavaScript

JavaScript is a programming language supported by all major graphical internet browsers that allows for the the programmability of web pages. After becoming widely adopted, a ECMAScript standard was created that specifies how the language works and all browsers that support it should abide by it. The language is loosely typed, meaning you do not have to specify which data type is being worked with. Being so, it accepts almost any input but its interpretation comes with its own idiosyncrasies. What is meant is not necessarily how it is understood and requires familiarization with the JavaScript way. In a way it makes programming easy and fast for beginners, yet this flexibility makes it harder to find out bugs. (Haverbeke 2015, 6)

Nowadays the JavaScript language is not limited to the web browser as covered in section 2.4.4. This opens the possibility to create server applications rather than being constrained to the client side. In addition, JavaScript has the node package manager (npm) and its software registry, the world's largest with over 800,000 packages (W3Schools s.a.).

2.3 Full Stack Development

Full stack development deals with both the front and back end development of web applications, but more specifically development that covers three layers: the presentation (user interface), application (business logic) and data (database) layers (DigiMantra Labs s.a.). In web development a common software architecture is the three-tier which organizes the application into three parts: the presentation (client), application (server) and data (database) tier. Sometimes the word tier is used interchangeably with layer but their meaning is different. While they both imply a functional division of the software, in the case of tier each one runs in its own infrastructure. (IBM 2020)

The back-end is the server-side software whose role is to implement and enforce the business rules. It is commonly written using frameworks which make development faster as more focus can be directed to writing the server instead of re-inventing the wheel. Back-end servers expose their data and functionality to clients through a set of definitions and protocols known as Application Programming Interface (API). A common type of API used in web applications is a RESTful API which uses the HTTP protocol. It implements the Representational State Transfer (REST) software architecture style that defines how

communication between server-client takes place based on a set of constraints (Lokesh Gupta 2022):

- Uniform Interface: The interface defined for resources should be always consistent such as having one logical resource identifier per resource.
- Client-server decoupling: Both applications are independent from one another and thus unconcerned with the other's inner working. This allows them to be developed separately.
- Statelessness: Each request must contain all necessary data for its processing to take place. Every request is treated as new since there are no sessions and the server does not remember anything about the client.
- Cacheable: Resources can be stored in a cache and recalled at server or client to improve performance.
- Layered system: Architecture can consist of multiple layers in between the client and server which are transparent, that is, their existence is not evident to the client.
- Code on demand: While normal responses contain static representations of resources, executable code can be also allowed to support the application.

The front-end corresponds to the presentation layer of the application, and falls into two main categories: single page application (SPA) and multiple page application (MPA). The MPA is the traditional approach and consists of server-side rendering which requires page reloading. SPA is the modern approach that is decoupled from the back-end and improves performance as after the initial load no subsequent reloads are required. (Asper Brothers 2019) As the SPA interacts with the back-end its application state changes and may grow in complexity. A solution for state management like Redux covered in section 5.2.2 may help in its simplification.

Databases in the data layer are used for storing and managing data, and require adequate data models that correspond to business requirements. Deficiencies in the model will require compensation through additional logic that adds complexity making it difficult to understand because it is not derived from business requirements (Herald Lynx 2018).

2.4 Technology Stack

The technology stack is the set of both software and hardware that comprise the overall solution. In the case of the marketing web application, as per specification by Alpha the PERN stack was chosen. It consists of the PostgreSQL database, Express.js web application framework, React.js UI JavaScript library and the Node.js JavaScript runtime environment. As for the hardware part of the technology stack it will be deployed to Azure, a cloud computing service by Microsoft.

The advantage of this software technology stack is that it allows the usage of JavaScript both in the front and back end. This is also the case for any stack which uses Node.js for the back end and JavaScript framework for the frontend. Using the same programming language in both back and front end allows in some cases for reusability of code, the usage of same software packages and results in an overall more fluid development experience. The developer has to only concern himself with JavaScript's programming paradigms.

2.4.1 PostgreSQL

PostgreSQL is one of the most popular open-source relational database in the world. It is highly conformant to SQL standards (ISO/IEC 9075 "Database Language SQL") making it stand out when compared with other relational database systems. It adhers to 170 out of 177 mandatory features to qualify for full core conformance. As of date, no other database management system claims full support (The PostgreSQL Global Development Group 2022). Additionally, being an open-source project that is not managed by a corporate entity but rather driven by a strong user community, its development is streared towards features that are highly valued by its users. As a result there is a large number of extensions and applications ready to enhance the PostgreSQL base functionality. (Ellingwood 2020)

2.4.2 Express

Express.js is an open-source Node.js web framework that is minimalist, unopinionated and flexible. Its non-dictating nature gives developers the freedom to implement features as they best see fit for their individual use cases such as: database, authentication, caching, templating and so on (OpenJS Foundation 2017a). The framework at its core relies in routes and middlewares, and provides a thin layer on top of Node.js making it very light and fast. (OpenJS Foundation 2017b)

Middlewares in Express are functions that execute in series in the middle of the request-response cycle. Each middleware in succession can execute logic, end execution, can alter the request object and/or execute the next successive middleware with the next() function. Middlewares can be of several type and operate at different levels: application-level, router-level, built-in, third-party (OpenJS Foundation 2015).

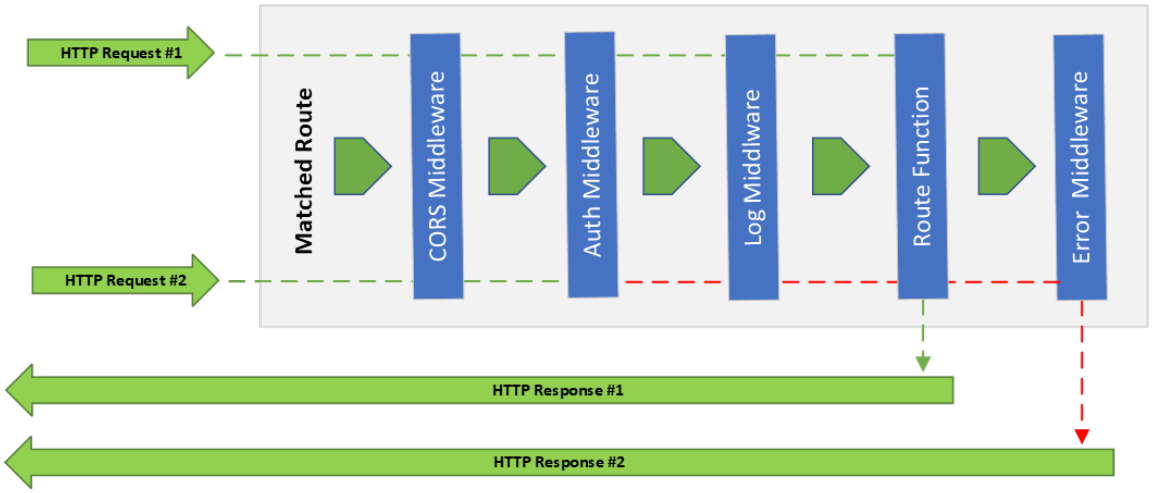

Figure 1: Express Request-Response Life Cycle

In Figure 1 shows that when a request matches a route it goes through a chain of middlewares where at any point it can return a response. In the figure, the red arrow represents an error response of some sort, and in the case of a green arrow, it represents that the request executed all previous middlewares successfully, including the current one. Whenever a response is emitted from a middleware it means that the request will not continue to go any further through the chain of middlewares but come to an end as a response is sent back to the source of the request.

2.4.3 React.js

React.js is a library for creating user interfaces (UI) through the creation of components which in turn make up an application. By separating the application into various components logic is isolated, details are abstracted and make the code reusable. (Meta Platforms 2016)

The Document Object Model (DOM) is a tree-like structure composed of elements that make up a webpage. Any changes performed to the DOM are slow and expensive because the browser has to perform CSS calculations and re-render the view. React uses a light version of the DOM called the Virtual DOM and keeps it synced with the real one. When changes are made to the Virtual DOM a new updated version of it is created and compared with the previous one. React is able to determine what elements need to be updated and performs changes to the real DOM in batch updates to optimize and speed the process. (GeeksforGeeks 2022a)

In React, components can be created using both functions or classes. Originally, classes were the only way to create stateful components given that they have a state object that determines their look and behaviour (GeeksforGeeks 2021). In contrast functional components are simple JavaScript functions which contain no state but whose limitations can be overcome by the use of various React hooks like useState to be able to create state variables and useEffect to create side-effects in components (GeeksforGeeks 2022b).

2.4.4 Node.js

Node is a runtime environment based on Google's Chrome v8 JavaScript engine. It allows the execution of JavaScript outside of the browser. It is a runtime because the v8 engine is included in a C++ program along with various objects that create an environment for the executed JavaScript and allows for example access to the file-system and networking that would be otherwise not possible with a browser bound v8 engine. (Hamedani 2018)

Node.js is single threaded insofar as to the JavaScript engine which does one thing at a time; but since Node.js is more than the v8 engine, that is not accurate. The asynchronous code is actually offloaded to the C++ application part that does have access to multiple threads, meaning it can do multiple things at the same time. Node.js uses the event loop to monitor the call stack and the callback queue. As functions are being called they are added into a stack of calls, as they end executing they are then popped off the stack. But when an asynchronous call is encountered that would otherwise block the stack until it finishes, it is handled by Libuv and assigned to a pool of worker threads. When these asynchronous operations finish, the callbacks are moved to the callback queue where they wait until the event loop sees that the call stack is empty and proceeds to push the callback to the call stack where they execute in the main thread. Node.js does not wait idle for asynchronous operations to finish before handling more requests. This makes it highly scalable and a perfect match for I/O-intensive applications. (Roberts 2014)

2.4.5 Azure Cloud Services

Microsoft Azure is the world's second top cloud computing platforms accounting for 21% of market shares in Q4 of 2021 (Synergy Research Group 2022). It consists of combination of around 200 products and cloud services that help business build, host and manage their solutions (Microsoft 2022a). Azure has a global infrastructure with multiple data centers grouped into different regions and connected through low latency network.

Within these regions there are physical locations known as high availability zones that maintain both data accessible as well as applications operating in case of datacenter failure. (Microsoft 2022b)

Among the notable services that Azure provides is App Service: a platform as a service (PaaS) for building and deploying your code. If facilitates the hosting of applications such as Web Apps, mobile back ends and REST APIs without requiring the management of infrastructure. Applications can automatically scale according to your needs, be highly resilient though high-availability, and their deployment can be automated directly from GitHub, Azure DevOps or a Git repository (Microsoft 2022c). The service also supports a list of pre-defined application stacks that use Node.js, Python, Java, ASP.NET, among others, but you are not limited to them as you can also run your own containerized application by providing a custom Docker container image (Microsoft 2022d).

Azure Storage is a cloud solution with a variety of data services for storing different types of data objects which include binary large objects (Blob), file shares using the server message block (SMB) protocol, tables for NoSQL data, virtual hard disks and messaging queues. The cloud storage solution is secure, features high availability and scalability, and provides access to stored data through a REST API. There are also client libraries in a variety of languages that can be leveraged by developers to create applications and services. (Microsoft 2022e)

Azure Monitor Logs / Log Analytics is a service for collecting log and performance data from resources that are subject to monitoring. The log data can be queried using language that can quickly sift through millions of records. The data that is captured can be used for visualization in a dashboard as well as in alert rules for keeping up to date and minimizing reaction time. (Microsoft 2022f)

DevOps services from Azure allow teams to work on projects from its conception to deployment. With features like boards to support Agile development for planning, tracking progress and ticket handling. There is a repository feature for version control, pipelines for continous integration and continuous deployment, you can automate the build and release code. There is also a test plans feature for testing your code through a variety of approaches such as manual and continous testing. With DevOps your team can share information through the built-in wiki. (Microsoft 2022g)

3 Requirement Analysis

A requirement in its most general sense simply means something that is needed, but in the context of software development it means what the product does or has to have in order for its user to accomplish a given goal (Ashrafi & Ashrafi 2014, 111). They fall into two categories: functional which deal the system's behavior, and non-functional which deal with nonbehavioral properties such as usability, reliability, performance, maintainability & security (Ashrafi & Ashrafi 2014, 113-116). Requirements are collected in a two steps process, the initial discovery and the subsequent gathering. The discovery phase defines the scope by identifying the basic needs; requirement gathering on the other hand is done within the defined boundaries of the scope and extracted from its depth through elicitation techniques: interviews, questionnaires, workshops, modeling, observation, and document analysis. (Ashrafi & Ashrafi 2014, 110)

3.1 Domain Analysis

Domain in requirement analysis refers to an area of connected activities that share a set of concepts and rules (Ashrafi & Ashrafi 2014, 162). Domain analysis' objective is to evaluate the context of requirements, mainly the problem space and reveal the concepts that are relevant to the solution (Ashrafi & Ashrafi 2014, 154).

Alpha is a software design and production company that operates in the IT consulting and service sector. The domain of the requirement analysis pertains to IT services of marketing in the context of car inspection.

Beta has as members both car inspection companies as well as chains that offer their services at multiple locations (stations). They use a car inspection software named Delta which integrates to the Alpha's ERP system Gamma. These companies need to continuously reach out to past customer as well as potential leads to keep the inflow of business and to fuel business growth. The different channels for marketing are SMS messages, e-mail and printed ads. Companies would like to have the ability to create templates for the various channels, and to add the correct customer and campaign data to the templates before executing the marketing campaigns. Gamma stores customer data of previous customers in the ERP system but it is subject to change and relying on it may result in ineffective marketing as a result of a vehicle ownership change. In the case of potential new customers Gamma is not allowed to send unsolicited SMS messages nor e-mail, just printed ads. To target both old and new users, Epsilon contains up to date data about all vehicle registrations in Finland.

3.2 Domain Dictionary

Domain dictionary is an organized collection of key concepts pertinent to the domain and derived from the domain analysis. It binds the stakeholders who need to confirm the concepts and the analyst who uses them as the basis for constructing a conceptual model of the system. (Ashrafi & Ashrafi 2014, 172)

Table 1: Domain Dictionary

Name	**Type**	**Description**
Gamma	Entity	ERP system used by car inspection companies
Delta	Entity	Car inspection system used by stations
Zeta	Entity	Publisher and direct marketing company
Chain	Entity	Car inspection company with 1 or more car inspection stations
Station	Entity	A location where cars are inspected
Campaign	Object	A car inspection service offer
Template	Object	Blueprint of marketing material
Image	Object	Visual design or picture used in print or email templates
Superuser	Role	Has overall visibility and permission
Administrator	Role	Administers a chain and its stations
User	Role	User with station limited permission scope

3.3 Business Dictionary

A business rule dictionary is a collection of classified statements to which business processes must adhere to. Documentation of business rules is necessary given that they often apply to more than one use case; doing so separately in a single collection avoids redundancy and detraction of the use case's descriptive flow. Additionally, documenting rules is not sufficient: we must assign a unique identifier to each one so that they can be linked to use cases and among themselves. (Podeswa 2005, 133). Overall, a single repository shared among key stakeholders aids communication and understanding of the business dynamics.

Table 2: Business Rules Dictionary

Id	**Definition**	**Type**	**Source**
BR1	Administrator role is responsible for managing a chain and all its stations.	Role	Domain Dictionary
BR2	Superuser is a role for users who are allowed to change the state of the marketing tool in general. They are allowed to view all existing chains and their corresponding stations.	Role	Domain Dictionary
BR3	User is a role that is associated with a single station and it is limited to its content.	Role	Domain Dictionary
BR4	A campaign may apply to one, multiple or all chain stations.	Fact	Product Owner
BR5	Each campaign template can be customized on a station basis.	Fact	Product Owner
BR6	A station can have one or none default (ongoing) template per template type.	Fact	Product Owner
BR7	If a campaign is active and does not have a template assigned, it will use the default template for a given station.	Fact	Product Owner
BR8	Zeta first receives customer data	Fact	Zeta

Id	Definition	Type	Source
	from Epsilon, pre-processes it and later provides it to the marketing webapp.		
BR9	Customer data provided by Zeta should be deleted within a 2 month window	Constraint	GDPR / Finnish Data Protection
BR10	The SMS and Printing marketing is conducted by Zeta. It fetches all campaign-template association data and templates from the marketing web app, and the campaign data from Gamma. Zeta builds the marketing material to be sent by replacing placeholders in the templates with the campaign and customer data.	Process	Product Owner
BR11	E-mail marketing material is build from the customer data provided by Zeta. As in BR10, the e-mail templates placeholders are replaced with the campaign and customer, and subsequently sent through a mail server provided by Gamma.	Process	Product Owner
BR12	When Zeta uploads customer data to the marketing webapp, it should return a unique identifier. The id can be used by Zeta to verity that the instance of data collection was deleted and when the deletion was done.	Process	Domain Expert Jussi from Zeta

3.4 Context Diagram

The context diagram's purpose is to give at a glance a representation of the overall system along with the entities involved and to highlight the most important interactions. The diagram is made up of three components: the system the center, external entities and their interactions with the system. This diagram is not concerned with the details but

provides a system bird's eye view that is useful when defining the scope of the system and allows at a glance confirmation of the system's boundaries. (Ashrafi & Ashrafi 2014, 209)

The diagram in Figure 2 depicts the entities that interact with the Marketing Web App such as Gamma, Delta, Zeta, the various types of user roles (Superuser, Administrator, User) and the objectives when said entities interact with the system.

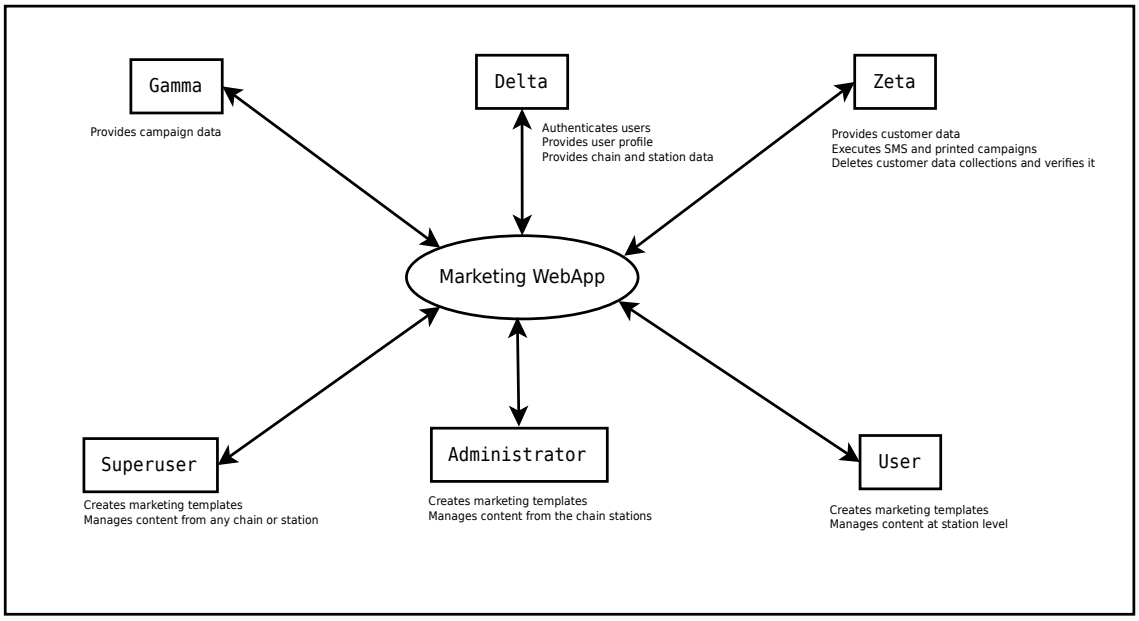

Figure 2: Context Diagram

3.5 Behaviour Modeling

Use cases function as behavioral contracts, that is, they specify how the system behaves and how external entities interact with each other. They are the prime source of functional requirements and the foundation of use case modeling. Use cases deal with a unit of related activities that are constrained to its boundaries. If the use case models too much or becomes too complex, it indicates that it needs to be broken down into separate ones.

3.5.1 Marketing Web App Use Case

The use case depicted in Figure 3: Marketing Web App Use Case conveys at a glance the three different types of user roles that interact with the Web App: superuser, administrator and user. As stated before in the domain and business rule dictionary, the scope of system access varies from one role to another but a hierarchy is implied by the generalization lines drawn from the superuser and administration roles to the user.

Generalization in this case means that the roles are based on user thus they inherit its capabilities. All user roles can create, view, list, edit and delete templates, as well as define ongoing ones used by default in campaigns where no assignment has been made explicitly. Campaign data cannot be modified in anyway as it is handled in the Gamma ERP system.

To the right of the use case are the secondary actors which assist the marketing web app to accomplish the primary actors goals. Chain and station data comes from Delta while the campaign data is provided by Gamma. Zeta can upload and delete customer data that is used in e-mail marketing. It also fetches SMS and print templates, and the campaign-template associations that are necessary for campaign execution.

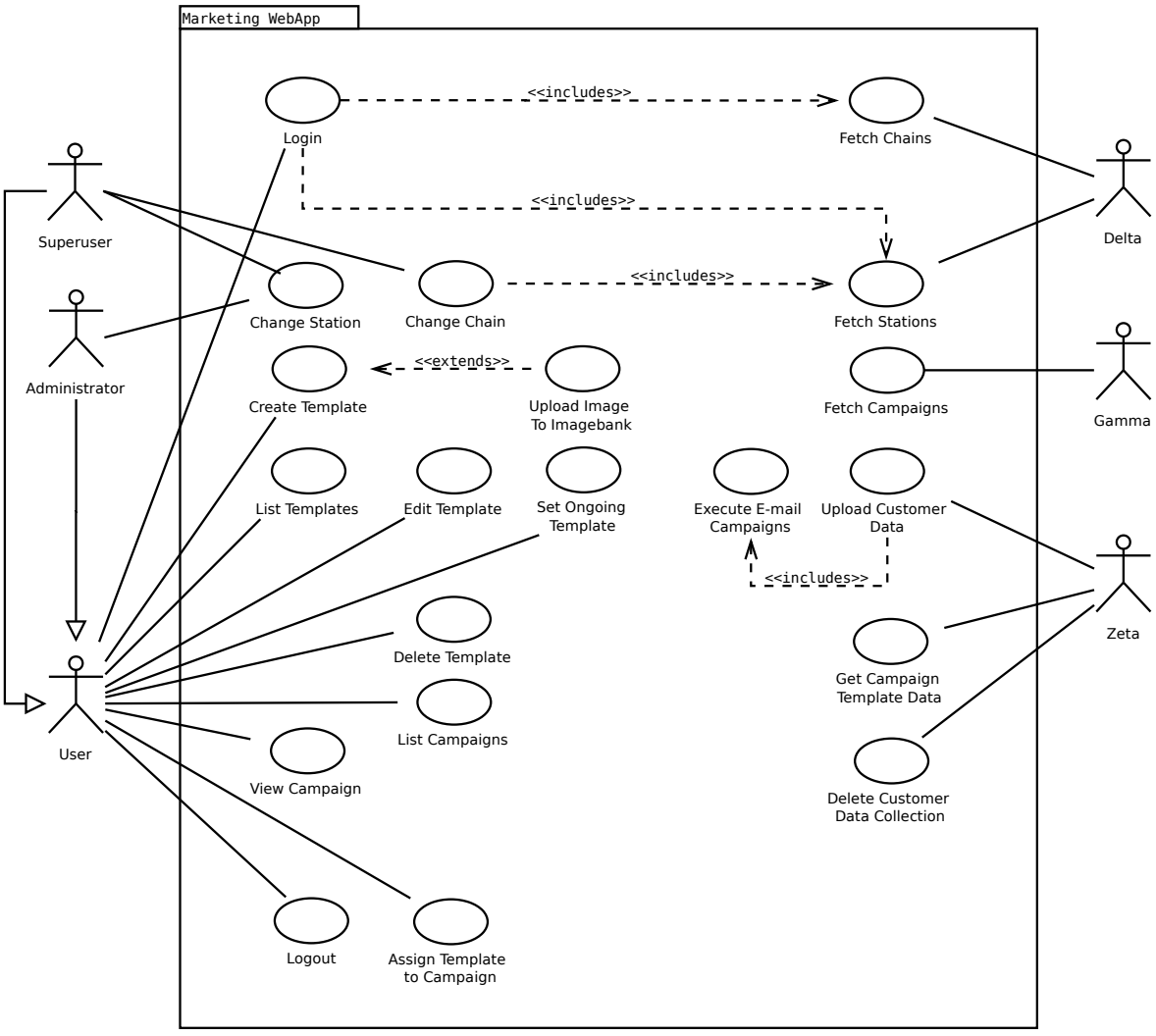

Figure 3: Marketing Web App Use Case

3.5.2 Print Template Editor Use Case

The use case in Figure 4 is straightforward and lacks secondary actors. The primary actor in the use case diagram can add two types of template elements: images and text. Images elements can be imported from the Imagebank or an image file uploaded directly. As for text elements: the font family, font size and colour can be defined. Both element types can be resized and positioned anywhere in the document. The elements stack position can be changed to determine the order in which they are rendered. The print template can be saved to the database as well as exported as a PDF file.

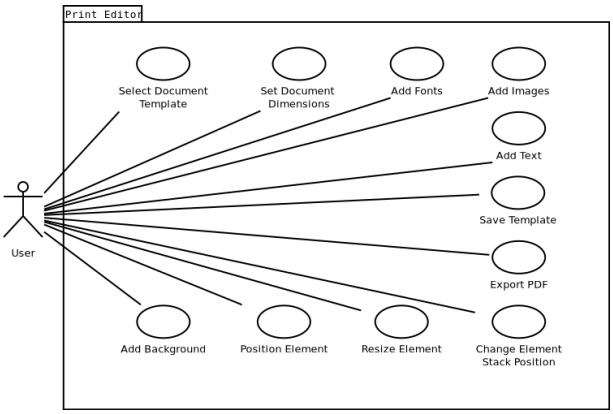

Figure 4: Print Editor Use Case

4 Database Design

Database design is a process consisting of three phases: conceptual, logical and physical design. The conceptual design focuses on creating a model of data as defined by the business requirements and is independent from all physical considerations. The model consists of the main entities and the relationships between them (one-one, one-to-many, many-many). The logical database design consists of using the previously created model as the foundation and refining it so that it includes all required entities with their attributes, and the entities related to one another. The model is independent of the database management system (DBMS) as well as all physical consideration. In this phase of design is where the process of normalization takes place as expounded in section 4.2.1. The physical design is the final phase of the process where the design is implemented and consequently takes into account the particular DBMS that will be used. Here relations, indexes, integrity constraints are defined as well as security measures are implemented. (Connolly & Begg 2015, 354-358)

4.1 Conceptual Design

In the conceptual database diagram in Figure 5: Conceptual Diagram the main entities along with their relationships are identified. It is important to note that the Chain, Station and Campaign entities are not to be stored in the database as relations since they come from the Gamma API. They are visualized in the diagram to understand their relationship and any constraint that are set on attributes that represent them as part of other relations. It is not necessary for a conceptual diagram to contain all the entities as in the final implementation.

When reading the conceptual digram, a line represents a relationship between two entities. The numbers at both ends separated by ".." is the UML notation that represents the cardinality (multiplicity) of the relationship. When reading the diagram for an entity and its relationship to another, the cardinality that is at the opposite end of the line is what is relevant for that entity. In the case of the Campaign and SMS Template relationship as shown in Figure 5, the Campaign entity would be read as follows: A campaign can have 0 or more (*) SMS Templates, and for its counterpart, a SMS Template can belong to 0 or many campaigns. In the case of the print template, it may contain 0 or many images, but an image can be contained by 0 or many print templates. In this way, an image can be used by many print templates.

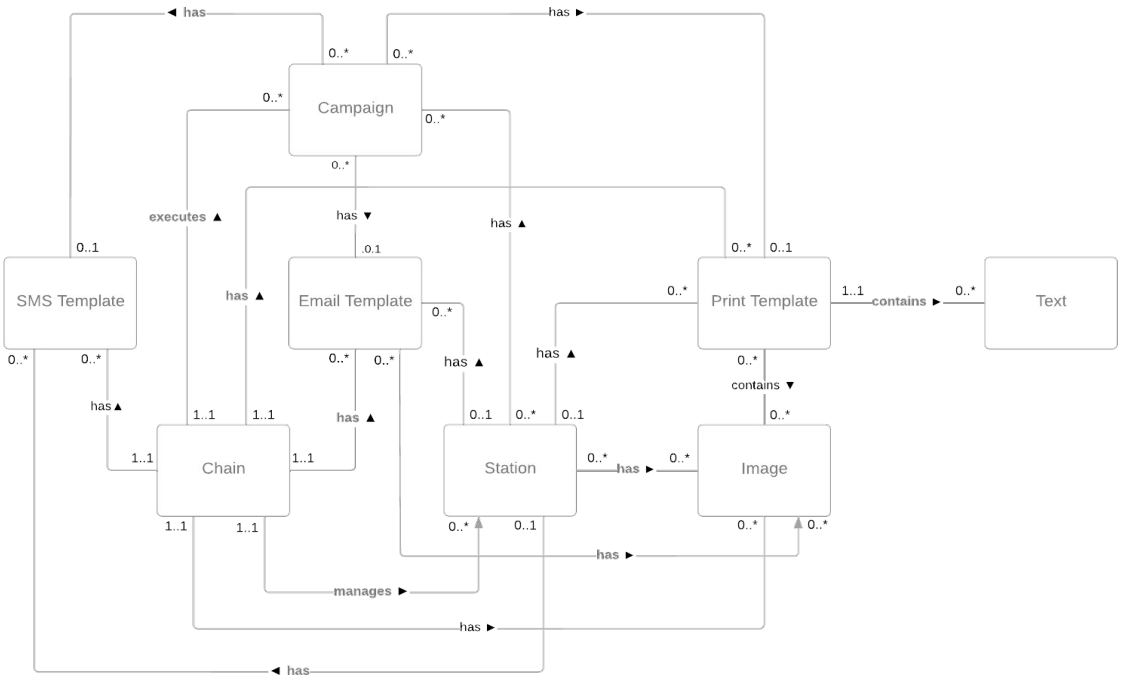

Figure 5: Conceptual Diagram

A more accurate and simpler conceptual diagram is represented in Figure 6 below. It shows the entities that will be stored as relations in the database. The table for campaigns is not to be stored in the database in its bare form as that data resides in the Gamma. However, the campaign id along with the templates that a campaign is indeed stored in the marketing web app database.

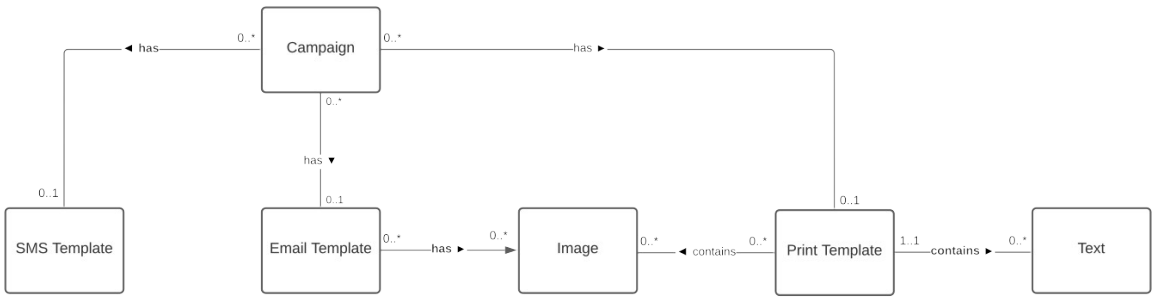

Figure 6: A more accurate and simpler conceptual diagram

4.2 Logical Design

There are three different tables for all template types: SMS, email and print. Each of their respective tables holds information about the template like to which chain and site it

belongs, whether it is an ongoing template, the user who authored it, time of creation and template content when applicable.

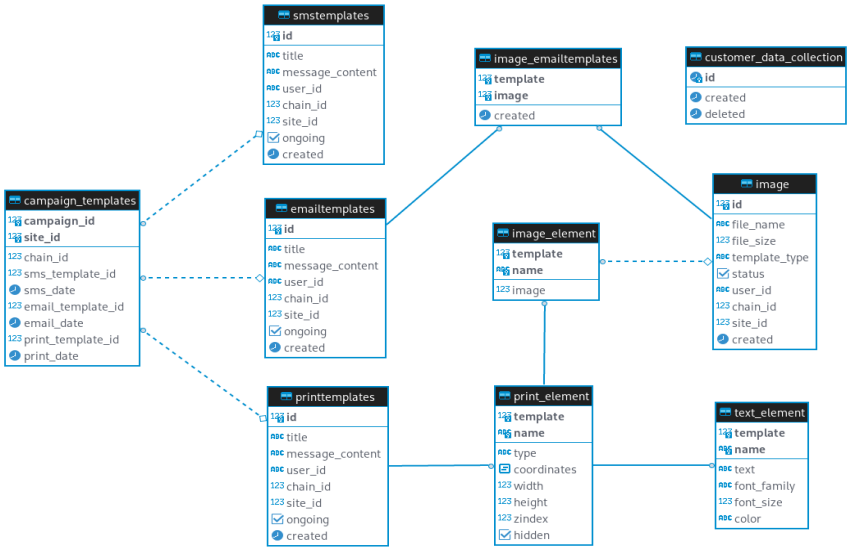

Figure 7: Logical Entity Relationship Diagram

The campaign_templates relation is an associative entity (bridge table) used for allowing many-to-many relationships. In it a campaign can be associated with many stations along with different types of templates with each entry. The campaign_templates relation is uniquely identified by the campaign id and the car inspection station id (site_id) which means that at any time, only one association of the different types of templates can exist per campaign and station. In this way campaigns are associated with individual sites explicitly. There is also the ongoing attribute for each template which means that in case there is no explicit association of templates to a campaign, the template defined as ongoing would be used by default in the campaign.

In the case of the print template, the actual content is stored primarily in the print_element, that is, all the data that is common to elements type image and text. The rest of the data that concerns each individual element type is stored in the corresponding table: image_element or text_element. The text_element holds the actual content of text elements, font family, size and color.

4.2.1 Normalization

Storing more data than necessary to meet the business needs is not only an unnecessary waste of space, inefficient when performing operations on it and can lead to data inconsistencies. An error as a result can turn out very costly when critical processes rely

on it. The remedy to this issue is normalization: the process of refining a relation using a set of rules to decompose it into smaller ones that keep data redundancy to the necessary minimum, and to provide data integrity while meeting data requirements.

To decide which attributes are to be stored together as part of a relation, it is necessary to understand how they relate to one another. For example, in a relationship R where values A and B are part of it and denoted by R = (A, B), then $A \rightarrow B$ means that for a value of A (determinant) there is one and only one value B (dependent), and therefore exist in a functional dependency. In a relation which functional dependencies exist are important because undesirable ones result in redundant data and update anomalies. (Connolly & Begg 2015, 456-457)

In functional dependencies the determinant and the dependant can consist of a single attribute (atomic) or of multiple ones (composite). Since the determinant can be composite, it may be composed of more than the necessary parts for the functional dependency to hold. If a subset of the determinant holds the dependency this is referred to as partial dependency. If nothing can be removed from the determinant to uphold the dependency, it is said to be a full functional dependency. Lastly, a transitive dependency exists in a relation R = (A, B, C, ...) when the determinant A determines B, which in turn determines C, if and only if A is not determined by attribute B nor C.

The different types of undesirable functional dependencies are taken into account in the normalization rules. There are many normal forms (NF), but the most common are:

- 1NF – Splits a record into multiple ones when the intersection of a row and column contains multiple values. All attributes in a relation must be atomic.
- 2NF – The relation is in 1NF and no partial dependency exists.
- 3NF – The relation is in 2NF and no transitive dependency is present.

Normalizing relations mitigates insertion, update and deletion anomalies. An insert anomaly occurs when a record cannot be inserted due to missing attribute value related to the primary key which cannot be null. Another type of insert anomaly is when an instance of a functional dependency already exists, an insertion that does not uphold because of a missing or incorrect determinant will result in inconsistent data. An update anomaly results when a functional dependency repeats in multiple rows, any modification to it in one instance requires the same change to all which is inefficient and failing to do so results in inconsistent data. A delete anomaly occurs when a relation stores data of independent

entities, to remove data of one results losing data pertaining to the other. Normalized relations are decomposed appropriately into separate relations thus the previously mentioned anomalies are avoided. (Connolly & Begg 2015, 455-456)

Table 3: Normal forms of marketing web app tables

Table	**1NF**	**2NF**	**3NF**
smstemplates	Yes	Yes	No
emailtemplates	Yes	Yes	No
printtemplates	Yes	Yes	No
print_element	Yes	Yes	Yes
image_element	Yes	Yes	Yes
image	Yes	Yes	No
campaign_templates	Yes	No	No
customer_data_collection	Yes	Yes	Yes

The relations for the various templates would under different data requirements suffice with the primary key (site_id, title) because sites (car inspection sites) belong to a chain. This would mean that chain_id is functionally dependent on site_id, thus redundant. But given that a template can apply chain wide with site_id being NULL requires chain_id in the relation. There could be several records with site_id NULL with different chain_id. Trying to address the two scopes for templates would in this case justify a surrogate key as the only way to uniquely identify the relation. The smstemplates, emailtemplates and printtemplates relations are in 2NF since there exist no partial dependency. They are however not in 3NF given that chain_id is transitively dependent on site_id.

The image table requirements are similar to those of the template types. The requirements of different scopes and involving NULL value for site_id requires that a surrogate key is used. The relation highest normal form is 2NF.

The campaign_templates as seen in Table 1 is not in 2NF and consequently not in 3NF either since the relation’s primary key is composite (campaign_id, site_id) and chain_id is functionally dependent on site_id resulting in a partial dependency. Therefore, it is not necessary to store the chain_id attribute in the relation. This is however a special case since no table exist for chain nor site entities. Gamma keeps this information and provides

it to the marketing webapp through its API. Storing the chain_id data allows for determining which campaign-template associations fall under a given chain without relying on the Gamma API.

4.3 Physical Design

The creation of the database is rather straightforward converting the logical design specifically for PostgreSQL. What is important to mention here is that for the template entities a surrogate key has been created called id. A surrogate key is an artificial key that is factless as it bears no inherent relation with the table's data. The surrogate key is necessary because a composite key that would include the chain_id, site_id and title would not uniquely identify each tuple (row). Data requirements indicate that site_id may be NULL for templates that apply to chain-wide, and the fact that PostgreSQL treats NULL values as distinct from one another, inserting an identical record with the same chain_id, site_id and title would result in duplication. A workaround for this issue is creating a partial index for each template type as the following example for smstemplates:

```
CREATE UNIQUE INDEX idx_smstemplates
ON smstemplates (chain_id, (site_id IS NULL), title)
WHERE site_id IS NULL;
```

A partial index will create an index not for a whole table but for a part of it, as constrained by the WHERE clause when the predicate "site_id IS NULL" returns true. The (site_id IS NULL) in the list of attribute when creating the index makes a boolean TRUE or FALSE depending if the value is NULL or not. Hence, even though PostgreSQL consider NULL values as not equal to one another, this index will prevent row duplication because of the conversion of NULL into a boolean value. The only key to truly identify the relation is the surrogate key id.

5 Software Development

The software development process is divided into the back-end and the front-end. The back-end encompasses authentication, services and middlewares with sample code that is described. The front-end covers how authentication was implemented, the SPA components with their descriptions as well as state management.

5.1 Back-end

The backend is responsible for serving the static content as well as serving as the Application Programming Interface (API) for the marketing web app. This API is connected to two others, Gamma where it obtains campaign data, and Delta from where it obtains user profile information. The marketing API also serves template data to Zeta that is used for SMS marketing as well as which template is associated to its respective campaign. In addition, Zeta uses the marketing API to submit customer data information sourced from Epsilon that will be used by the marketing web application when sending e-mail advertisement. The Figure 8 shows the connection to Azure AD B2C for authentication as explained in the next section, other Azure services that are integral to the back end such as the PostgreSQL Server as the database, Blob Storage for images and Log Analytics for logging.

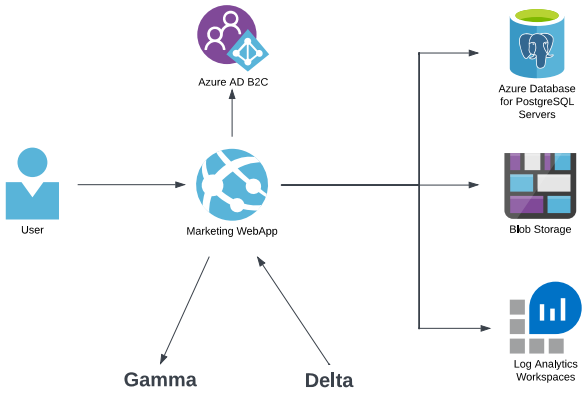

Figure 8: Back End Diagram

The structure of the back end is divided into folders for configuration, routes, services, middlewares, tests, utility functions and database scripts. The package.json file contains scripts for starting the server in production, development and test mode, for building the application (front end is included in public directory), and deploy for pushing to repository.

5.1.1 Authentication

The marketing web app authentication is done using Azure AD B2C which provides business-to-customer identity as a service. While it supports several authentication protocols, in this instance the Oauth 2.0 standard is used to enable single sign-on solution that works across Gamma, Delta and the marketing web app.

The back end relies on two packages: passport and passport-azure-ad. The idea behind passport is simple and works just like a passport but for authenticating requests. It is implemented as a middleware that is added to routes. There are multiple strategies for authentication, but the marketing web app uses oauth bearer token. When making requests, the token is passed including it as a header in the HTTP requests and subsequently validated.

5.1.2 Services

Services are objects with different functions as attributes that contain logic that interacts with the database or network. It is a way to abstract details and keep them in their own level of abstraction. When create, retrieve, update, delete (CRUD) operations are performed the route logic is not cluttered with unnecessary details making the code reusable, shorter and more readable.

```
9   router.get(
10    '/campaigns',
11    [ authenticate, loadProfile, userScopePermission ],
12    async (req, res, next) => {
13      try {
14        const { chainId, siteId } = req.query;
15        const campaigns = await campaignService.getAll(chainId, siteId);
16        return res.json(campaigns);
17      } catch (error) {
18        next(error);
19      }
20    }
21  );
```

Figure 9: Campaign Service in Campaigns Route

In Figure 9 it is clearly shown how short, readable and simple the campaign service logic can be when fetching all campaigns.

5.1.3 Middlewares

The back end uses several middlewares that are built-in to the Express.js framework, custom ones for the marketing web app, and non-custom sourced from the node package manager (NPM) repository. Examples of middlewares that are built-in are the

express.json() for parsing json payloads included as part of requests, and the express.urlencoded() for parsing urlencoded payloads. Line 11 in Figure 9 shows an array of middleware that are applied to the route for path “/campaigns”.

Middlewares sourced from the NPM repository are cors and express-fileupload. Cors uses HTTP headers to define from which origin other than its own a resource is allowed to be loaded. In other words, a browser client visiting website A will not be able to load resources from website B using JavaScript unless there is the Access-Control-Allow-Origin header in the response that allows it for the A origin. As for the express-fileupload middleware, it makes file uploads accessible in the request object using req.files. (MDN 2022)

Table 4: Middleware List

Middleware	**Type**	**Description**
express.json	built-in	Parses json payload in requests
express.urlencoded	built-in	Parses urlencoded payloads
cors	package	Enables cross-origin resource sharing
express-fileupload	package	Makes file-uploads accessible in the request object
loadProfile	custom	Loads profile to request object
entityLoader	custom	Loads entity to request object
entityPermission	custom	Determines user entity permission
userScopePermission	custom	Determines users permission for a given scope
errorHandler	custom	Handles request errors
logger	custom	Logs request to Azure Log Analytics
apiAccess	Custom	Validates requests from Zeta using some header checks

There are several custom defined middlewares contained in Table 4 starting with the loadProfile middleware which is responsible of fetching the user profile and adding it to the request object. Fetching the profile depends on the profile token that the user supplies in the request header. The profile contains useful data on the user like which chain and sites it belongs to, as well as the type of user that will be used later when determined permissions.

The entityLoader as its name indicates, is responsible for loading an entity in the route which it is used, for example when the user wants to retrieve an entity. It depends on whether the profile is available in the request object so that permission could later be enforced, only then the entity will be added to the request object.

When determining if a user has permissions to access a given entity, the entityPermission middleware as in Figure 10 uses the profile and entity to work out if access should be granted. A superuser will have access to any entity regardless. An administrator which is bound to a chain in scope will have access to the entity, if and only if it belongs to the chain he manages. In case the user type would be that of a regular user, access is granted if the entity's chain and site matches that of the user profile.

```
const entityPermission = (req, res, next) => {
  const { user, profile, entity } = req;
  if (!user || !profile) throw 401;
  if (!entity) throw 403;
  switch (user.userType) {
    case 'superuser':
      next();
      break;
    case 'admin':
      const adminOfEntity = user.chainId == entity.chain_id;
      if (adminOfEntity) {
        next();
      } else {
        throw 403;
      }
      break;
    default:
      const userBelongsToSite = profile.sites.includes(entity.site_id);
      if (userBelongsToSite) {
        next();
      } else {
        throw 403;
      }
  };
};
```

Figure 10: entityPermission middleware

The errorHandler middleware is responsible for returning the appropriate response depending on the type of error which occurred as shown below in Figure 11.

```
const errorHandler = (err, req, res, next) => {
  if (err.code) {
    switch (err.code) {
      case '23502':
        return res.status(400).json({ error: 'not null violation' });
      case '23503':
        return res.status(400).json({ error: 'resource is in use'})
      default:
        return res.status(400).json({ error: 'bad request' });
    }
  }

  if (!isNaN(err)) {
    switch (err) {
      case 400:
        return res.status(400).json({ error: 'bad request'});
      case 401:
        return res.status(401).json({ error: 'unauthorized'});
      case 403:
        return res.status(403).json({ error: 'forbidden' })
      case 404:
        return res.status(404).json({ error: 'resource not found'});
      case 500:
        return res.status(500).json({ error: 'internal server error'});
      default:
        return res.status(400).json({ error: 'bad request'});
    }
  }
  next(err);
};


module.exports = errorHandler;
```

Figure 11: errorHandler Middleware

The userScopePermission is similar to that of the entityPermission but does not depend on the entity but on the chain or site id included in the request query parameters. Superuser is authorized by default, the admin is authorized when the chain id matches, and a regular user only when chain and site id are the same. The middleware rejecting user access returns an HTTP 403 (Forbidden) response.

The logger middleware's function is just to capture some request details such the ip address, user id, HTTP method, path of the request and status code of the response. It is stored offsite using Azure Monitor for auditing purposes later on. It responds to the "finish" event of the response when the last segments of headers and body have been given to the operating system to send through the network .

```
const { loggerConfig } = require('../config');

const loggerWorkspaceId = process.env.LOGGER_WORKSPACE_ID;
const loggerSharedKey = process.env.LOGGER_SHARED_KEY;

const logger = (req, res, next) => {
  try {
    res.on('finish', () => {
      const { method, path, ip } = req;
      const { statusCode } = res;
      const user_id = req.user ? req.user.oid : 'anonymous';
      const date = new Date();
      const datetime = date.toISOString()
      const message = `${ip} ${user_id} [${datetime}] "${method} ${path} - ${statusCode}"`;
      const messageObject = { ip_address: ip, user_id, datetime, method, path, statusCode };

      logFunction(messageObject);
      console.log(message);
    });

    next();
  } catch (err) {
    next(err);
  }
};
```

Figure 12: Logger Middleware

The last middleware is the apiAccess middleware which validates requests to endpoints in the marketing API that are meant for Zeta. It checks for a set of HTTP headers that are mandatory when making a request, especially a key and the way it is crafted. The key changes according to the timestamp used. It is valid for ±1 hour from the specified time.

5.1.4 API

Zeta requires access to the marketing API to fetch campaign-templates associations so that it can determine which templates to use for a marketing channel when executing a campaign. The endpoint is "/api/campaign-templates" uses the apiAccess middleware whose sole function is to check for a set of headers so that the request can be validated. Zeta can also fetch all templates from the "/api/templates" endpoint, and this too is subject

to the apiAccess middleware. The template's placeholders will be replaced in Zeta's end with the actual template data.

5.2 Front-end

The marketing front-end is a single page application composed of React components. Routing within the application consisted of assigning components to defined routes using the react-router-dom package. The state management was simplified by implementing Redux Toolkit (RTK) as described in 5.2.2. For styling the front-end, the makeStyles function from Material-UI library was used. This function returns a hook that links style sheets to functional components. Additionally, the library also provides user interface components used in application components. The following text covers the authentication, state management and primary components of the front-end.

5.2.1 Authentication

The front end uses the package @msal-react which wraps around @msal-browser. There is a provider component MsalProvider which gets passed an instance of the PublicClientApplication, an instantiated class with some configuration used for the authentication and authorization functions in single page applications (Microsoft 2022h). In the MsalProvider, the MsalAuthenticationTemplate component is used to wrap the content that will be shown only if the user is authenticated otherwise attempt to sign the user in.

5.2.2 Redux & Redux Toolkit

Managing the state of a front-end application becomes more challenging as the codebase grows in size and complexity. As the state changes, it is difficult to keep track of it and reason about issues as they unfold. Redux is a state container for JavaScript applications that addresses these challenges by taking approach to state management that is predictable, centralized, easy to debug and able to work with about any user interface layer (Abramov 2022a).

Redux at its core consists of a store, actions, reducers. The store is a JavaScript object where data about your application is stored. Actions are also objects with a type attribute describes something they do that optionally contain payloads and are executed when dispatched to reducers. A reducer is a function that takes in the state and an action returning a new updated version of the state, and therefore should not cause side effects.

Using Redux can introduce a fair amount of boilerplate code in your application which can be tedious to type. You have to write code for re-fetching data to keep an updated state, do caching, etc. Redux Toolkit is a toolset that the previously stated issues by being opinionated, very simple to setup and generates a lot of code for you automatically removing boilerplate code (Abramov 2022b). The toolkit includes RTK Query which is a tool for fetching and caching data such as that of an API. You can define queries for endpoints, define tags for the content fetched and invalidate those tags for automatic re-fetching of the data as needed based on the names of your endpoints. The RTK Query for React will automatically create hooks for queries that fetch or mutate data, provide access to errors and isLoading variable which is quite handy. In the case of the Figure 13, the endpoint getAllPrintTemplates will have the hook named useGetAllPrintTemplatesQuery. For a query that actually mutates the state like in the case of addPrintTemplate, it would be useAddPrintTemplateMutation.

```
export const marketingApi = createApi({
  reducerPath: 'marketingApi',
  baseQuery: fetchBaseQuery({
    baseUrl: getBaseBackendApiUrl(),
    prepareHeaders: async (headers, { getState }) => {
      const token = await getToken();
      const profileToken = await getToken('doris');
      if (token) headers.set('authorization', `Bearer ${token}`);
      if (profileToken) headers.set('ProfileAuthorization', `Bearer ${profileToken}`);
      return headers;
    }
  }),
  tagTypes: ['SmsTemplate', 'EmailTemplate', 'PrintTemplate'],
  endpoints: (builder) => ({
    getAllPrintTemplates: builder.query({
      query: ({ chainId, siteId }) => ({
        url: `printtemplates`,
        params: { chainId, siteId }
      }),
      providesTags: (result) => [
        { type: 'PrintTemplate', id: 'LIST'},
        ...result.map(template => ({ type: 'PrintTemplate', id: template.id }))
      ],
    }),
    getPrintTemplatesById: builder.query({
      query: (id) => ({ url: `printtemplates/${id}`}),
      providesTags: (result, error, id) => [{ type: 'PrintTemplate', id }]
    }),
    addPrintTemplate: builder.mutation({
      query: ({ payload, chainId, siteId }) => ({
        url: `printtemplates`,
        method: 'POST',
        body: payload,
        params: { chainId, siteId }
      }),
      invalidatesTags: [{ type: 'PrintTemplate', id: 'LIST' }]
    }),
```

Figure 13: RTK Query Marketing Api Slice

5.2.3 Campaigns

The campaigns (kampanjat in Finnish) route in Figure 14 displays a list of campaigns with starting and ending dates, the number of stations to which it applies, the vehicle types, price, code. The channels that are highlighted mean that there are templates associated with them. The listing of campaigns is dependent on the pair of drop-down menus, the first for chains and the second for stations. When the link for the campaign is clicked on, the user is redirected to the campaign page where there are three distinct sections for each marketing channel. At the top there is a text field for the default or ongoing template.

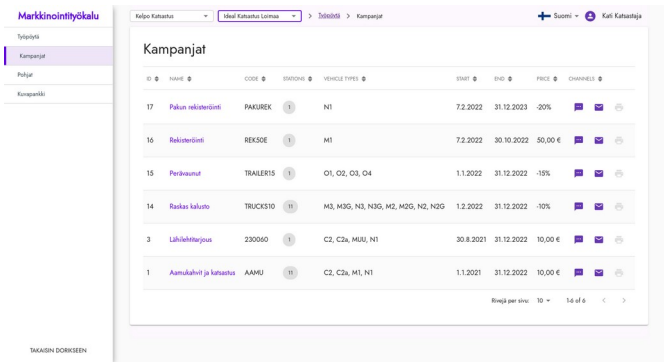

Figure 14: Campaign List

Below the default template text field, there is a drop down menu that is searchable with autocomplete function used for setting explicitly a template for the campaign. The save button at the bottom records the template assignment as shown in Figure 15.

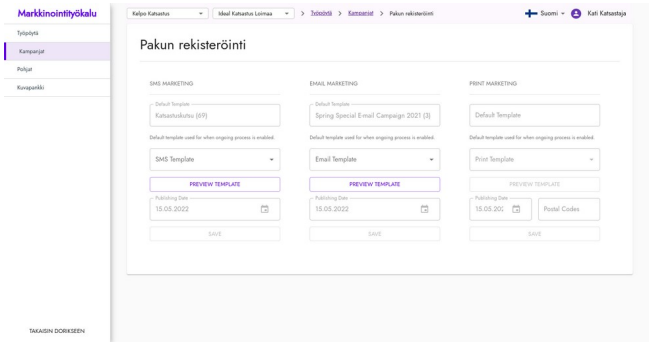

Figure 15: Campaign View

The user can preview the template straight from the campaign view by clicking the preview button in each marketing channel. The selection of the correct template can be confirmed as exemplified in Figure 16.

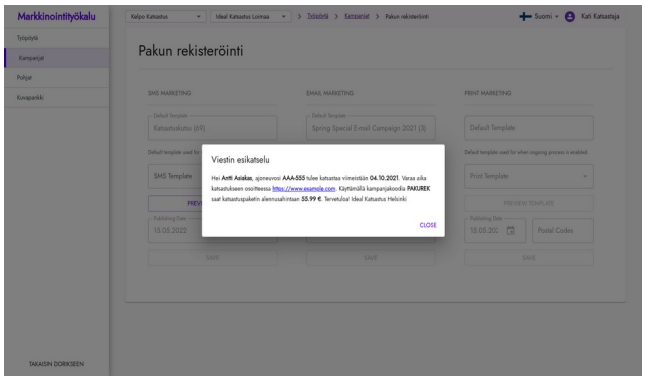

Figure 16: Template Preview

5.2.4 Templates

The TemplateListView component shown in Figure 17 contains child components that list templates for each of the template types. The attributes listed for each template are id, title, content and date. On the far left of the list there is the “jatkuva” column where ongoing templates can be defined by clicking on the start and confirming the action. There can only be one ongoing template per type and site. Henceforth the template will be used as the default for any campaign that applies to the site and has not been explicitly assigned a template.

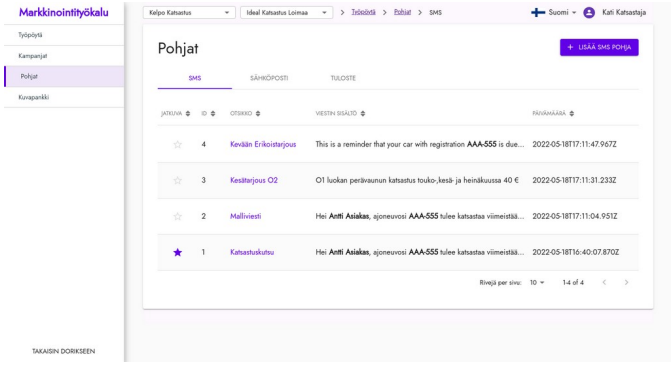

Figure 17: TemplateListView Component

Users add template clicking on the “Lisää SMS pohja” which is Finnish for add SMS template. The button is placed in the same location for each template type list. After proceeding to create a new template the user is presented the template form which in the case of the SMS and E-mail you are displayed a live preview on the right like shown in Figure 18. The variable placeholders buttons will insert placeholders for campaign and customer data in the template message. On the bottom there are three buttons: first from left to right saves and exits to the template list after confirming action, the second cancels the action and goes back, and the last will delete the template after confirmation.

Figure 18: Template Form Component

The only difference between SMS and E-mail templates form is that in the latter there is a text editor which allows you to format the text and include images

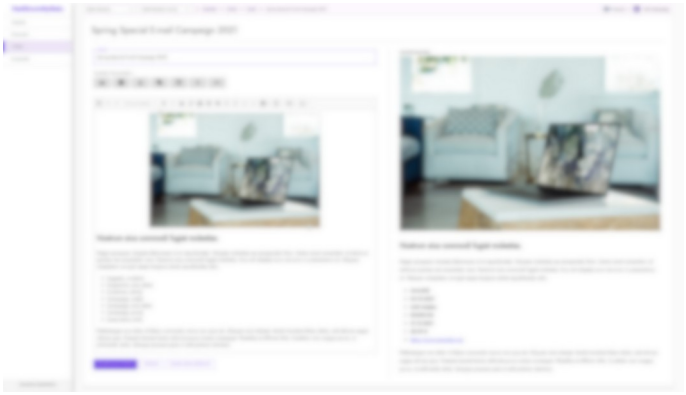

Figure 19: Email Form Component

5.2.5 Imagebank

The Imagebank is a react component for browsing images stored in two separate containers in Azure Storage, one for images used in printed marketing and the second in e-mails. A tab exists for each image category as shown in Figure 20. Images for print templates can be added by clicking on the upload image button. The images used in e-mails are uploaded as part of the template creation process.

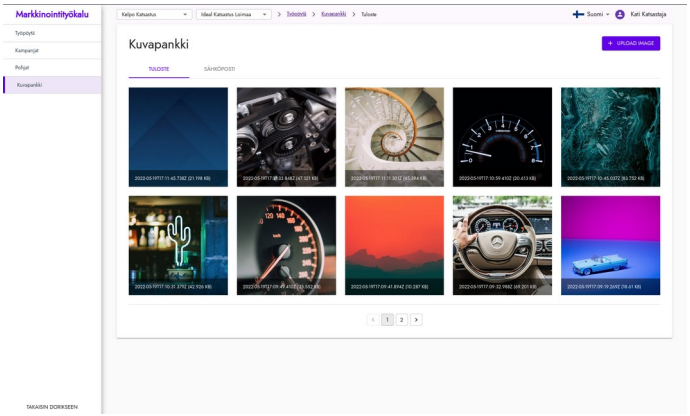

Figure 20: Imagebank

Images in the image bank can be clicked to display the image component which displays image information. In this page as depicted in Figure 21, for both print and e-mail categories, images can be deleted on the condition that they are not associated with any email or template.

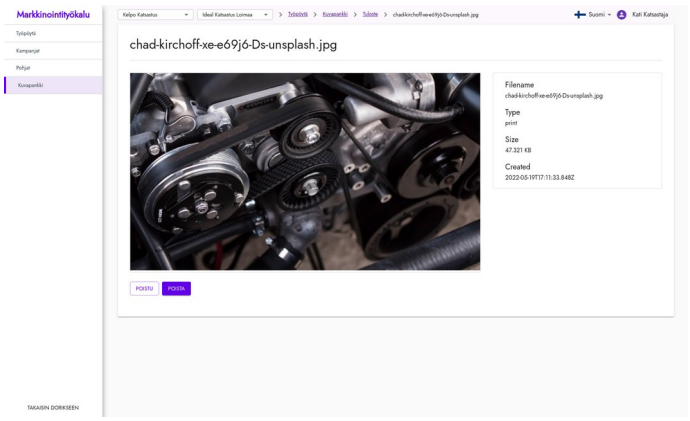

Figure 21: Image View

5.2.6 Print Editor

The print editor is a React component for creating print templates which relies substantially on event listeners to accomplish its task. The representation of the print template is first created using HTML and later translated into a PDF using the pdf-lib library. The templates are made up of print elements such as images and text which can be imported and created respectively, their visibility can be toggled, and elements can be removed too. Figure 22 shows that each element represents a layer whose order in the template is determined by how they stacked up against one another. Layers can be dragged and reordered which changes the element's CSS zIndex value, and eventually determines in what order they are rendered in the PDF.

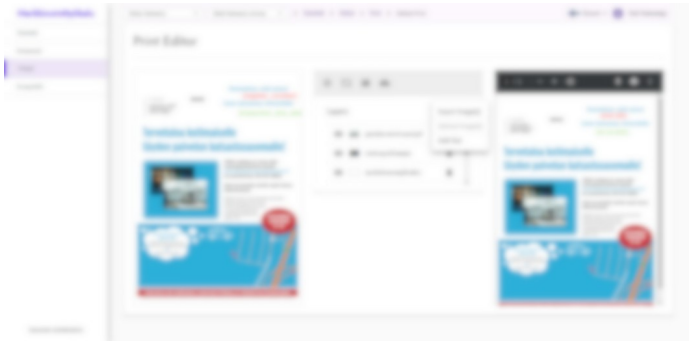

Figure 22: Print Editor

The editor's main HTML element has the dimensions of a A4 page at 300 PPI/DPI (2480 x 3508 pixels) which is the minimum requirement for sharp images that are good for quality prints. The element is transformed using CSS to scale and match width wise the allocated space while retaining its aspect ratio. Print elements can be moved around the area, but as of now only text elements can be resized. Depending on the image type the quality can

be impacted when they are resized therefore that functionality is not currently available. In a future version there will be the possibity of adding vector graphics which will allow resizing without impacting quality. Print templates' data can be saved to the database and downloaded as a PDF file.

Text elements can have their color and font type changed as shown in Figure 23 . The text can contain placeholders which are later replaced with the respective data. When positioning text elements the alignment is to the left making correct positioning a result of some trial and error, align center feature will be added later.

Figure 23: Adding Text Element

5.3 Unit Testing

A software code changes throughout its development life cycle. How flexible it is determines how easy it is to adapt to change and maintain. Having tests gives freedom for code to evolve; changes can be automatically verified to be in compliance with expected behaviour. When testing is excluded from coding every change brings with it potential bugs. Yet, having tests that are not properly written is as bad or even worse than having no test at all. This is due to the fact that both test and production code need to change together. If tests are not properly written it will be difficult to change them and also difficult to change production code. This results in tests being a liability instead of an asset. (Martin 2009, 123-124)

Jest is a JavaScript testing framework that aims at simplicity and requires no initial configuration to work. It can run tests in parallel to improve performance as well as serially. With Jest provides a Mock Function API that makes easy to mock modules, return values, and so on.

When running tests with Jest, as depicted in Figure 24 you can define beforeAll and afterAll functions for setup work and for doing clean up afterwards. Work that needs to be done repeatedly before and after each test can be defined inside beforeEach and afterEach functions.

```
const request = require('supertest');
const app = require('../server');
const { pool } = require('../config');

let server;
beforeAll(async () => {
  await pool;
  await pool.query('ALTER SEQUENCE smstemplates_id_seq RESTART WITH 1');
  await pool.query('TRUNCATE TABLE smstemplates CASCADE');
  server = await app.listen(8888);
});

afterAll(async () => {
  await pool.end();
  await server.close();
});
```

Figure 24: Setup and Teardown

Groups of related tests can be included inside of a describe function call as demonstrated in Figure 25. The supertest npm package is useful for tests that include HTTP assertions and can be used in conjunction with Jest.

```
describe("When testing routes for SMS templates", () => {
  it("a template can be added", async () => {
    const payload = {
      title: 'smstemplate',
      message_content: 'smstemplate content'
    };

    const res = await request(server)
      .post('/api/smstemplates')
      .query({ chainId: 1, siteId: 1})
      .send(payload);
    expect(res.status).toEqual(201);
    expect(typeof res.body.id).toBe('number');
    expect(res.body.title).toEqual(payload.title);
    expect(res.body.message_content).toEqual(payload.message_content);
  });
```

Figure 25: Group of test with describe()

The Figure 26 shows a run of the tests and the results. The “-i” option or its equivalent “--runInBand” is supplied when running Jest so that tests are ran in serial as opposed to parallel; running serially guarantees that fetching an entity does not precede its creation.

```
[human][archlinux][±][development → origin ↑119 {2} U:1 ?:9 ✗][/workspace/projects/mdsp/backend]
▪ npm run test

> ykl-mt-api@0.0.0 test
> TZ=utc NODE_ENV=test jest -i --verbose --detectOpenHandles

 PASS  tests/emailTemplates.test.js
  When testing routes for E-mail templates
    ✓ a template can be added (208 ms)
    ✓ a template can't be duplicated for site scope (59 ms)
    ✓ a template can be retrieved (860 ms)
    ✓ a template can be updated (95 ms)
    ✓ a template can be deleted (163 ms)

 PASS  tests/smsTemplates.test.js
  When testing routes for SMS templates
    ✓ a template can be added (161 ms)
    ✓ a template can't be duplicated for site scope (68 ms)
    ✓ a template can be retrieved (825 ms)
    ✓ a template can be updated (128 ms)
    ✓ a template can be deleted (171 ms)

Test Suites: 2 passed, 2 total
Tests:       10 passed, 10 total
Snapshots:   0 total
Time:        8.257 s, estimated 9 s
Ran all test suites.
[human][archlinux][±][development → origin ↑119 {2} U:1 ?:9 ✗][/workspace/projects/mdsp/backend]
▪
```

Figure 26: Running tests

6 Deployment to Azure Cloud

In Azure a subscription needs to be set up in order to be able to be billed for usage of resources. Applications that run in App Service always need to be run using an App Service Plan which defines the computing resources available like the size of the virtual machine, the pricing tier and so on. These as well as other resources that are related are group in containers called Resource Groups. As a initial setup of the infrastructure a resource group was created with an app service plan and app service to run the marketing web app, a postgresql database, log analytics workspace, storage account and storage containers.

6.1 Azure Pipelines

Azure Pipelines is a service part of Azure DevOps for building, testing and deploying code. It allows for Continuous Integration (CI) of changes through the automation of merging and testing. With Azure Pipelines you can also extend CI to Continuous Delivery (CD) to also automate deployment to various environments such as testing, quality assurance and production. Doing CI minimizes integration difficulties that arise when it is not a continuous practice such as postponing it close to release day. CD means a project is always ready to be shipped in its current state provided that testing is met.
(Pittet 2022)

Pipelines in Azure DevOps can be created using the classic user interface editor or through a YAML file: a human-friendly data serialization language. The pipeline for the marketing web app project was done using the YAML file because it is less abstracted and requires deeper understanding. In Figure 27: Pipeline Trigger and Variables the trigger for the pipeline is set to be the development branch, any changes committed will result in the pipeline's execution. There are several pipeline variables set like the subscription used to be billed, the web app to where the code will be deployed, and lastly the agent's virtual machine image name that refers to the platform to build on.

```yaml
# Node.js Express Web App to Linux on Azure
# Build a Node.js Express app and deploy it to Azure as a Linux web app.
# Add steps that analyze code, save build artifacts, deploy, and more:
# https://docs.microsoft.com/azure/devops/pipelines/languages/javascript

trigger:
- development

variables:
  # Azure Resource Manager connection created during pipeline creation
  azureSubscription: 'YklMtDevSubscriptionManager'

  # Web app name
  webAppName: 'YKL-MarketingTool-Dev-APP'

  # Agent VM image name
  vmImageName: 'ubuntu-latest'

```

Figure 27: Pipeline Trigger and Variables

The pipeline consists of two stages: build and deploy. The build stage in Figure 28 consists of one job with following steps:

1. Check out the backend repository.
2. Check out the frontend repository.
3. Install Node.js.
4. Go into the backend repository and install the npm packages.
5. Go into the frontend repository and install the npm packages.
6. Build the frontend.
7. Copy the build to the backend ‘public’ directory.
8. Archive the build.
9. Upload the build.

```yaml
stages:
- stage: Build
  displayName: Build stage
  jobs:
  - job: Build
    displayName: Build
    pool:
      vmImage: $(vmImageName)

    steps:
    - checkout: git://YKL-MarketingTool/backend@development
    - checkout: self

    Settings
    - task: NodeTool@0
      inputs:
        versionSpec: '12.x'
      displayName: 'Install Node.js'

    - script: |
        cd $(Agent.BuildDirectory)/s/backend
        npm install
        cd $(Agent.BuildDirectory)/s/frontend
        npm install
        REACT_APP_DEPLOYMENT_URL="https://ykl-marketingtool-dev-app.azurewebsites.net" npm run build
        cp -R build $(Agent.BuildDirectory)/s/backend/public
      displayName: 'npm install & build'

    Settings
    - task: ArchiveFiles@2
      displayName: 'Archive files'
      inputs:
        rootFolderOrFile: $(Agent.BuildDirectory)/s/backend
        includeRootFolder: false
        archiveType: zip
        archiveFile: $(Build.ArtifactStagingDirectory)/$(Build.BuildId).zip
        replaceExistingArchive: true

    - upload: $(Build.ArtifactStagingDirectory)/$(Build.BuildId).zip
      artifact: drop
```

Figure 28: Azure Pipeline Build Stage

The deploy stage in Figure 29 depends on the build stage and runs only when it succeeds. It consists of one job with one step. It runs the application given environmental variables, inputs provided for the task like the Azure subscription to use, the app service type and name, runtime stack, application settings and the start-up command for the deployment.

```yaml
- stage: Deploy
  displayName: Deploy stage
  dependsOn: Build
  condition: succeeded()
  jobs:
  - deployment: Deploy
    displayName: Deploy
    environment: $(environmentName)
    pool:
      vmImage: $(vmImageName)
    strategy:
      runOnce:
        deploy:
          steps:
          Settings
          - task: AzureWebApp@1
            displayName: 'Azure Web App Deploy: YKL-MarketingTool-Dev-APP'
            env:
              DATABASE_PASS: $(DATABASE_PASS)
              LOGGER_SHARED_KEY: $(LOGGER_SHARED_KEY)
              EMAIL_PASS: $(EMAIL_PASS)
              STORAGE_ACCOUNT_KEY: $(STORAGE_ACCOUNT_KEY)
              DORIS_API_CODE: $(DORIS_API_CODE)
              KAPA_API_CODE: $(KAPA_API_CODE)
              GRANO_API_CODE: $(GRANO_API_CODE)
            inputs:
              azureSubscription: $(azureSubscription)
              appType: webAppLinux
              appName: $(webAppName)
              runtimeStack: 'NODE|12-lts'
              package: $(Pipeline.Workspace)/drop/$(Build.BuildId).zip
              appSettings: '-
              startUpCommand: 'TZ=utc PORT=3000 NODE_ENV=production node server.js'
```

Figure 29: Azure Pipeline Deploy Stage

7 Reflection

The project was developed in part as a freelance project. Although its development was technically done using the Scrum framework, it was not included as part of the thesis. Having a single developer can lead to ad-hoc processes, and often it did. The epics and user stories that were part of the planning phase of every development iteration were done and managed but not as consistently as it should have been. Committing to framework that is based on Agile principles of inspection and transparency would have resulted in faster and more consistent delivery of value to the customer.

The database model had a few issues that were ultimately resolved to meet the business requirements. Before that was the case, the development continued until it could not. It became evident that not only the data model needed to change but also the business and presentation logic. The issue was caused because of not being aware that it was possible to create folder structures in Azure Storage by how images were named when uploading them. All images belonging to all chains were stored on containers without any folders for separation. This meant that there could only be one instance of an image name for the whole system. As a temporary solution a timestamp was added to each image's name for making them all unique and the primary key for the relation was incorrect. The data model was eventually fixed as well as the way the images were uploaded. It did however require the tedious task of finding and changing everything to implement the change. The data model has to be right from the beginning, when the business requirements are reflected in it, the development just makes more sense.

The state management at first was done using plain Redux. There was a lot of boilerplate code, the management of fetching and re-fetching was not done initially to the satisfaction of the developer. Upon further investigation, Redux Toolkit with RTK Query and createSlice greatly simplified the state management and fetching data from the back-end with its hooks, and automated re-fetching. After implementing it, everything was clearer and worked as intended.

The Print Editor development was one of the most challenging part of the project as it was not clear how to go about creating it. It eventually was inspired by image manipulation software like Gimp or Photoshop which have layers that are stacked. Having an HTML element scaled down to an acceptable size that would contain all templates element where they could be repositioned and resized made sense. All that was needed was to translate their positions to the actual PDF document at the correct scale. The Print Editor is almost complete but still requires the addition of a few features. As of now images can

only be imported from the Imagebank, fonts are restricted to the standard ones provided by the PDFlib library. Users need to be able to also upload images from the editor, more fonts need to be added and layers need to be nameable. Finally the print templates need to be added to the API so that Zeta may retrieve them.

7.1 Ideas for Improvement

One approach whose implementation was contemplated after the fact was doing Test Driven Development. Test Driven Development (TDD) is a programming style which ties three activities together: testing, coding and refactoring. Tests set expectations, production code meets them, and refactoring improves the code structure without altering its behaviour (Agile Alliance 2022). Refactoring must adhere to the constraints of simple design as proposed in the book Extreme Programming Explained (Beck 2004):

- Passes all testing
- Code reveals its intention
- Contains no duplicated logic
- Least amount of code to get the job done

In Test Driven Development, testing, coding and refactoring are tied together though the observance of the three TDD laws at every step, and are as follows:

1. You cannot write production code until you have written a failing test.
2. You cannot write more test code than necessary for the test to fail and not compiling is failing.
3. You cannot write more production code than is necessary to pass the current failing test.

The TDD process as shown in Figure 30 starts with the creation of a failing test. When writing the test code, the goal is not to write a comprehensive test at once, this objective will be achieved gradually. Since no production code exists, very little code will be required for the test to fail. The second step is to write code just to pass (green) the test. Again, the goal is not to write the complete production code either, but to pass the failing test. When this is achieved—refactorization—the last step of the iterative cycle will take care to not just make the code work, but make it work well. Note that refactored code needs to pass the test code as well since it is part of the simple design criteria. The iteration of the whole process ends when the test completely validates the requirements and when the refactored production code passes the test. The overall idea is that writing test, production, and refactored code are not separate activities but are intimately connected. No activity gets to fall out of step allowing the whole process to advance in

sync. This results in just the necessary code that does what it is expected to and does it well.

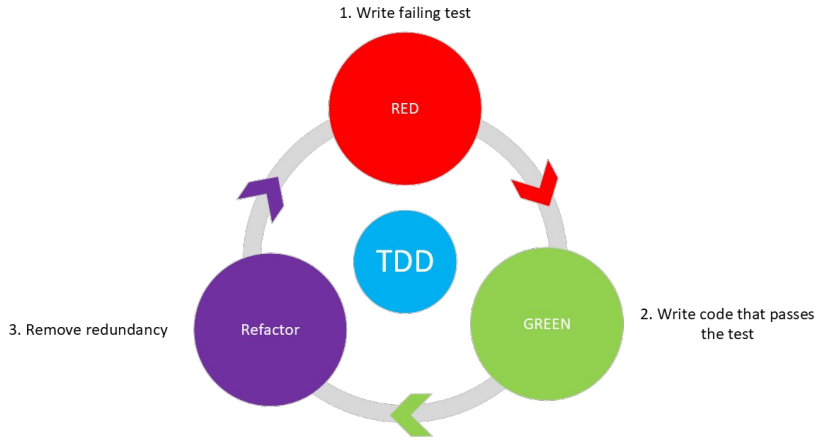

Figure 30: Test Driven Development

To conclude, the reasoning behind why Test Driven Development was not implemented in the midst of the project's development is, as stated by Deming (1986, 29): "Inspection does not improve the quality, nor guarantee quality. Inspection is too late. The quality, good or bad, is already in the product. As Harold F. Dodge said, 'You can not inspect quality into a product.'"

References

Abramov, D. 2022a. Redux - A predictable state container for JavaScript apps. | Redux. URL: https://redux.js.org/. Accessed: 5 May 2022.

Abramov, D. 2022b. Redux Toolkit. URL: https://redux-toolkit.js.org/. Accessed: 5 May 2022.

Agile Alliance 2022. TDD. URL: https://www.agilealliance.org/glossary/tdd/. Accessed: 15 February 2022.

Ashrafi, N. & Ashrafi, H. 2014. Object Oriented Systems Analysis and Design. Pearson.

Asper Brothers 2019. Single Page Application (SPA) vs Multi Page Application (MPA) – Two Development Approaches. URL: https://asperbrothers.com/blog/spa-vs-mpa/. Accessed: 20 May 2022.

Beck, K. 2004. Extreme Programming Explained. Addison-Wesley.

Connolly, T. & Begg, C. 2015. Database Systems: A Practical Approach to Design, Implementation, and Management. Pearson.

Deming, W.E. 1986. Out of the Crisis. The MIT Press.

DigiMantra Labs s.a. What Is A Full Stack Development? URL: https://digimantralabs.com/full-stack-development/. Accessed: 15 May 2022.

Ellingwood, J. 2020. The benefits of PostgreSQL. URL: https://www.prisma.io/dataguide/postgresql/benefits-of-postgresql. Accessed: 12 April 2022.

GeeksforGeeks 2022a. ReactJS | Virtual DOM. URL: https://www.geeksforgeeks.org/reactjs-virtual-dom. Accessed: 11 April 2022.

GeeksforGeeks 2021. What is Stateful/Class based Component in ReactJS ? URL: https://www.geeksforgeeks.org/what-is-stateful-class-based-component-in-reactjs. Accessed: 11 April 2022.

GeeksforGeeks 2022b. ReactJS Functional Components. URL: https://www.geeksforgeeks.org/reactjs-functional-components/. Accessed: 11 April 2022.

Hamedani, M. 2018. Node.js Tutorial for Beginners: Learn Node in 1 Hour. URL: youtube.com/watch?v=TlB_eWDSMt4. Accessed: 11 April 2022.

Haverbeke, M. 2015. Eloquent JavaScript: A Modern Introduction to Programming. No Starch Press. San Francisco.

Herald Lynx 2018. What is a Full Stack developer? URL: https://www.heraldlynx.com/full-stack-developer/. Accessed: 21 May 2022.

IBM 2020. Three-Tier Architecture. URL: https://www.ibm.com/cloud/learn/three-tier-architecture. Accessed: 15 May 2022.

International Organization for Standardization 2005. ISO/IEC 19501:2005. URL: https://www.iso.org/standard/32620.html. Accessed: 21 February 2022.

Lokesh Gupta 2022. REST Architectural Constraints. URL: https://restfulapi.net/rest-architectural-constraints/. Accessed: 20 May 2022.

Martin, R.C. 2009. Clean Code: A Handbook of Agile Software Craftsmanship. Pearson Education.

MDN 2022. Cross-Origin Resource Sharing (CORS). URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS. Accessed: 5 May 2022.

Meta Platforms 2016. React Top-Level API. URL: https://reactjs.org/docs/react-api.html. Accessed: 11 April 2022.

Microsoft 2022a. What is Azure? URL: https://azure.microsoft.com/en-us/overview/what-is-azure/. Accessed: 24 April 2022.

Microsoft 2022b. Global Infrastructure. URL: https://azure.microsoft.com/en-us/global-infrastructure/. Accessed: 24 April 2022.

Microsoft 2022c. App Service. URL: https://docs.microsoft.com/en-us/azure/app-service/. Accessed: 24 April 2022.

Microsoft 2022d. App Service Overview. URL: https://docs.microsoft.com/en-us/azure/app-service/overview. Accessed: 24 April 2022.

Microsoft 2022e. Introduction to Azure Storage. URL: https://docs.microsoft.com/en-us/azure/storage/common/storage-introduction. Accessed: 24 April 2022.

Microsoft 2022f. Azure Monitor Logs Overview. URL: https://docs.microsoft.com/en-us/azure/azure-monitor/logs/data-platform-logs. Accessed: 24 April 2022.

Microsoft 2022g. What is Azure DevOps? URL: https://docs.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops. Accessed: 24 April 2022.

Microsoft 2022h. PublicClientApplication | microsoft-authentication-libraries-for-js. URL: https://azuread.github.io/microsoft-authentication-library-for-js/ref/classes/_azure_msal_browser.publicclientapplication.html. Accessed: 5 May 2022.

OpenJS Foundation 2017a. Express FAQ. URL: https://expressjs.com/en/starter/faq.html. Accessed: 8 February 2022.

OpenJS Foundation 2017b. Express - Node.js web application framework. URL: https://expressjs.com/. Accessed: 8 February 2022.

OpenJS Foundation 2015. Writing middleware for use in Express apps. URL: https://expressjs.com/en/guide/writing-middleware.html. Accessed: 8 February 2022.

Pittet, S. 2022. Continuous integration vs. delivery vs. deployment. URL: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment. Accessed: 5 April 2022.

Podeswa, H. 2005. UML for the IT Business Analyst: A Practical Guide to Object-Oriented Requirements Gathering. Thomson Course Technology PTR. Accessed: 17 February 2022.

Roberts, P. 2014. What the heck is the even loop anyway? URL: https://youtube.com/watch?v=8aGhZQkoFbQ. Accessed: 11 April 2020.

Synergy Research Group 2022. As Quarterly Cloud Spending Jumps to Over $50B, Microsoft Looms Larger in Amazon's Rear Mirror. URL: https://www.srgresearch.com/articles/as-quarterly-cloud-spending-jumps-to-over-50b-microsoft-looms-larger-in-amazons-rear-mirror. Accessed: 5 May 2022.

The PostgreSQL Global Development Group 2022. PostgreSQL 14.1 Documentation. URL: https://www.postgresql.org/docs/current/features.html. Accessed: 12 April 2022.

W3Schools s.a. What is npm? URL: https://www.w3schools.com/whatis/whatis_npm.asp. Accessed: 14 May 2022.