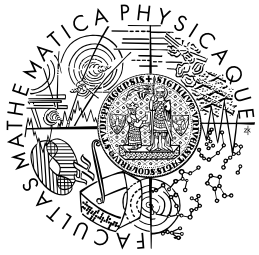


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Michal Půlpán

Development and deployment of SaaS solution for logistics automation in e-commerce

Department of Software Engineering

Supervisor of the master thesis: Mgr. Petr Škoda, Ph.D.

Study programme: Computer Science

Prague 2024

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to take this opportunity to express my deepest gratitude to all those who supported me during my years of study. First and foremost, a special thanks to my supervisor Mgr. Petr Škoda, Ph.D. for all the support, guidance, help and encouragement throughout my studies and this thesis. I also want to express my heartfelt thanks to my girlfriend, Marie, my parents, my sister, and the rest of my family. Your unwavering support, patience, and tolerance for all my time-consuming hobbies, my studies, and this thesis mean the world to me.

Title: Development and deployment of SaaS solution for logistics automation in e-commerce

Author: Bc. Michal Půlpán

Department: Department of Software Engineering

Supervisor: Mgr. Petr Škoda, Ph.D., Department of Software Engineering

Abstract: E-Commerce has seen rapid growth that significantly changed the retail landscape. This situation requires brands to do more than just offer products; they must present a distinct identity, maintain brand values, and establish deeper relationships with customers. Brands must focus on differentiating and drawing attention to themselves while keeping up with customer care. Traditionally, logistics communication during order delivery is usually left to the shipping carrier, but it is an important element in the order lifecycle. It provides an opportunity to increase brand awareness and strengthen relationship with the customer. This thesis proposes a solution to bridge the gap between streamlined shipping logistics and its use as a potential marketing channel. Introduces a SaaS platform tested in a real-world environment on thousands of parcels. Allowing sellers to manage parcel data transfers to carriers, print shipping labels, and efficiently track parcel statuses while providing shipment information to recipients through custom branded email notifications and tracking pages. This approach not only improves operational efficiency, but also reinforces brand engagement throughout the delivery process.

Keywords: e-commerce logistics process automation web application software as a service

Název práce: Vývoj a nasazení SaaS řešení pro automatizaci logistiky v e-commerce

Autor: Bc. Michal Půlpán

Katedra: Katedra softwarového inženýrství

Vedoucí práce: Mgr. Petr Škoda, Ph.D., Katedra softwarového inženýrství

Abstrakt: E–commerce trh v oblasti prodeje zboží zaznamenal růst, který významně ovlivnil celý maloobchod. Prodejci se dostali do situace, kdy nestačí produkt pouze nabízet, nýbrž je nutné budovat značku a posilovat vztahy se zákazníkem. Značky se tak musí odlišit od konkurence, upoutat na sebe pozornost a zároveň klást důraz na udržení kvality péče o zákzníka. Logistická komunikace během doručování objednávky je zpravidla přenechávána přepravci, ale představuje důležitý prvkem v životním cyklu objednávky. Nabízí možnost zvýšit povědomí o značce a posílit vazbu se zákazníkem. Tato práce navrhuje řešení spočívající v automatizaci procesů datové komunikace s přepravcem a v jejím využití jako potenciálního marketingového kanálu. Představuje SaaS platformu testovanou v reálném firemním prostředí na tisících zásilkách. Ta umožňuje prodejcům spravovat přenos dat k přepravcům, tisknout přepravní štítky a efektivně sledovat stav zásilek. To vše s upravitelnými notifikačními e-maily a sledovací stránkou určenou příjemci. Tento přístup nejenže zefektivňuje zaběhnuté procesy expedice, ale zároveň posiluje značku během ponákupního marketingu v průběhu přepravy objednávky.

Klíčová slova: e-commerce logistika automatizace procesů webová aplikace software as a service

Contents

Introduction

In recent years, e-Commerce has experienced rapid growth, changing the retail environment across the globe. This trend, strongly reflected in the Czech Republic, has placed online shopping not only as an alternative to physical retail, but also often as the preferred shopping channel for a wide demographic. The rapid rise of e-Commerce in the Czech Republic, along with the broader Central and Eastern European region, introduces competitive challenges and opportunities. According to the [1] E-commerce Study 2023 by Czech Association for Electronic Commerce (Czech Association for Electronic Commerce (APEK)), the Czech market was worth about $8 billion in 2023 with 61% of the Czech Internet population (15 +) shopping at least once a month online. As more consumers turn to online shopping, the market has become saturated with the large number of vendors that demand attention with significant marketing budgets. This situation requires retailers to do more than just offer products; they must present distinct identities, maintain brand values, and establish deeper connections with their customers. Sellers must aim to differentiate themselves, turning the focus towards building a recognisable presence while keeping up with customer care.

The competitive core of e-Commerce enforces brands to refine their strategies. In this context, the battle is not just about sales, but also about becoming the go-to-shop within the product domain. Ogunmola and Kumar [2] emphasise that the growing competitive environment in online retail forces brands to continuously innovate, especially to improve the shopping experience to differentiate themselves and achieve a dominant position in the market. Brand must wisely think about every touch point, from website quality to user interface, user experience, customer service, and logistics, as an opportunity to boost brand awareness and values.

The logistical aspect of e-Commerce, often seen as a backend operation, has come to the forefront as an important part of customer satisfaction and brand differentiation. Efficiency in order processing, reliability in shipping, and transparency in delivery updates are now increasingly important in the customer experience. In today's fast digital world, consumer's patience for slow order processing has significantly decreased. According to the mentioned study by APEK in September 2023, a growing number of customers report that transparency in delivery times and fast order processing are among their main considerations when choosing between two vendors. This highlights a clear trend: Customers are willing to pay a premium for the assurance of a faster and more transparent delivery. This shift brings a new challenge to e-Commerce businesses; slow order processing is no longer just a logistical issue but is directly related to customer retention and brand loyalty. With that said, it is clear that the customer paying more attention to the delivery time of their order, will be likely to appreciate continuous updates of their order status in a user interface similar to the e-Commerce store they purchased from. This presents a problem that many e-Commerce platforms and retailers are dealing with: how to streamline their logistic operations to meet the demands of modern consumers.

The solution proposed in this thesis aims to address these needs by creating a simple-to-use platform designed for dispatching orders to the shipping carriers,

seamlessly sending data to the carriers, printing shipping labels, and updating order statuses. In addition, this platform will serve as a new marketing communication channel, offering a branded parcel tracking experience.

To ensure the applicability of the platform in real-world scenarios, this thesis will also include the implementation of a connector for SAP Business One. This integration will enable the seamless exchange of data between third-party software and SAP Business One. The platform will be tested in a company that operates in both the Business to Business (B2B) and Business to Customer (B2C) segments of the fashion e-Commerce industry, handling more than 100 packages per day. This environment presents an ideal setting for evaluating the platform's capabilities.

Motivation

Process of dispatching orders, communicating with shipping carriers, and providing customers with timely update is full of inefficiency and challenges. Traditionally, these operations involve various manual interventions, leading to delays and errors that directly impact customer satisfaction and brand loyalty. In an era where consumers value speed, it is not viable to manually upload data set to the carriers web interface and then request shipping labels if everything goes well. For a company that cooperates with multiple shipping carriers, this process becomes quickly unsustainable. It has to be automatic with direct feedback of data errors and import problems. For example, if the shipping address is not valid or if the carrier raises any other error with the provided data set or its own service. Having said that, each carrier is an isolated company without any unification when it comes to the data they accept and provide. Bridging the gap between the communication interface of each carrier and generalising parcel shipping statuses quickly becomes a very appreciated task. As a result, businesses can seamlessly integrate new shipping carriers without being bogged down by the specific implementation details and the varying data formats each carrier uses.

When a company collects data from its shipping carriers, it is important to utilise this information. Leveraging such data not only streamlines operations but also provides a competitive advantage by improving decision making and improving customer satisfaction and brand recognition. We will use this opportunity to present the tracking data with a company branding to increase brand awareness and create a new and unexplored marketing channel that customers are not used to and therefore resistant to.

The goal is to transform logistics from a potential pain point into a competitive advantage for e-Commerce businesses, thus not only meeting but exceeding customer expectations by providing them with a branded tracking page and automatic e-mail notifications with branding.

Project goals

The project is driven by a set of clear goals designed to address the challenges identified in the e-Commerce logistics domain and the software development itself:

G1: Streamline logistics operations: Develop a platform that simplifies the process of dispatching orders to shipping carriers, automating data exchange, and minimizing need for manual intervention.

G2: Modern cloud based multi-tenant solution: Create application with multi-tenant architecture allowing it to be used by multiple companies deployed to the cloud with automated integration and deployment.

G3: Create branded shipping customer experience: Introduce a new marketing communication platform using the data collected from the shipping carrier that allowed each company to specify custom branding for the parcel tracking page and parcel status notification emails.

G4: Integration with existing systems: Develop a solution that can be easily integrated with existing businesses' system.

G5: Validate in a real-world setting with SAP Business One integration: Test the platform in a live e-Commerce environment, handling a significant volume of orders in daily operations.

Solution overview

The proposed solution will be a Software as a Service (Software as a Service (SaaS)) platform designed to modernize and simplify e-Commerce order data dispatch logistics. As its core, the platform will facilitate order dispatching to the shipping carrier, label printing, and periodic updates of order statuses.

The entire code base will use continuous integration (Continuous Integration (CI)) practices and automatic deployment (Continuous Deployment (CD)) to the Amazon Web Services (Amazon Web Services (AWS)) ensuring high availability, security, and fast response times with resource scaling based on traffic.

The platform's user interface will be intuitive, well-documented, and easy to use, requiring minimal training for staff, and will provide customisable options for businesses to maintain their brand identity throughout the customer's post-purchase journey. The branded tracking pages and notification emails are not just an enhancement of the logistics process; it is a redefinition of how businesses communicate with their customers, transforming every shipment into opportunity for engagement and brand reinforcement.

The testing and validation of the platform will take place in a company operating in both B2B and B2C segments of the fashion e-Commerce industry, dealing with more than 100 orders daily and running an instance of SAP Business One with which the platform will have to exchange data. This real-life usage will provide thorough testing of the platform and provides valuable insight.

This thesis is organised to comprehensively address the dual aspects of exploitation and exploration within software development, together with the challenges of solution analysis, implementation, and integration encountered in working with SAP Business One.

- **Related Work 1:** Reviews industry options in the sector and discusses project objectives.
- **Analysis 2:** It represents the actual work process within a given problem domain with the result of functional and non-functional requirements for a proposed platform.

- **Architecture 3:** Outlines the architectural design of the proposed solution, describing its components and their interactions.
- **Technical design 4:** Describes technical specifications and design considerations to create the platform.
- **Implementation 5:** Details the development process of the platform.
- **Deployment 6:** Explains the platform deployment strategy, including cloud hosting and service provisioning on AWS.
- **SAP Integration 7:** Discusses and presents the integration with SAP Business One, focusing oon thedevelopment of an application for direct data exchange with the system.
- **Evaluation 8:** Covers the evaluation used to validate the functionality of the platform in a real-world day-to-day business setting.

1. Related work

This chapter dives into the overview of existing solutions within the scope of parcel logistics in the dispatch process, generating labels, and shipment tracking, with a focus on the Czech market. Understanding these projects and their limitations helps us define the market gap we are trying to fill. Particularly in offering a cloud-based multi-tenant solution integrating customised tracking page and email notifications for recipients. The overview underscores the importance of innovating beyond current offerings, which primarily lack features such as a simple dashboard for data viewing, custom tracking pages for improved customer communication, and automated email notifications.

1.1 Related projects

Projects handling data communication with carriers are, of course, heavily biased with the demographics they are targeting. Although the process is usually very similar, shipping companies and customers with their e-Commerce platforms or Enterprise Resource Planning (ERP) solutions are different. Hence, we will limit ourselves to the Czech logistics environment, where several systems facilitate the integration with local carriers. However, these solutions often fall short in several areas:

- None operates as a cloud-based multi-tenant solution that offers the business a hassle-free platform for their logistic needs without the necessity for in-house infrastructure or maintenance.
- Usually, they require a distinct approach and data model for different carriers. This makes it more difficult to integrate.
- They typically do not provide a custom tracking page for end-to-end customer communication, missing an opportunity to enhance the customer experience with a branded informative user interface.
- Current solutions typically do not allow a single company to use multiple shipping carrier contracts through set of different API credentials. This limitation can be problematic for businesses with multiple warehouses, each requiring different shipping carriers due to their unique logistics needs.

Let us take a look at some options available in the Czech market.

1.1.1 Balíkobot.cz

Offers integration directly into ERP/e-Commerce systems, however lacking the flexibility to modify parcels outside of these systems. Requiring that the user using this software can modify the source data in ERP and not being able to use any Balíkobot user-interface might be a strong limitation. The next limitation might be the custom label format provided by Balíkobot.cz. Although the carrier validates it, sometimes a different layout might lead to inefficiency or even errors at the sorting centres or when loaded to the delivery vehicle. On the other hand,

the amount of integrated carriers and ERP integrations makes Balikobot very easy to start using. One thing to consider is that Balíkobot does not handle customer communication at any level. It is only strictly used for data transfer and printing the shipping labels.

Overall, Balíkobot is definitely a considerable solution, but being integrated directly into the ERP makes it a bit slow to use and is usually inaccessible from outside the company network when needed. And, most importantly, it deals only with company processes, not with customer communication and presentation.

1.1.2 LabelPrinter.cz

LabelPrinter, as the name suggests, is designed for label printing and data transfer. Like Balíkobot, LabelPrinter functions solely as warehouse software to transfer data from companies to carriers. In addition,, it operates only as an on-premise Windows service runtime at the client's computer. This approach requires local infrastructure and maintenance, potentially increasing operational overhead in small to medium-sized enterprises, limiting scalability and accessibility.

It might also be beneficial to mention that companies usually also use "in-house" solutions, which are generally a bespoke set of scripts designed for data transfer without additional features like customer communication. These are often difficult to maintain and lack functionalities such as shipping status mapping, essential for recognising final delivery statuses.

1.2 Addressing the shortcomings of existing solutions

Problems of data communication with carriers present numerous challenges with existing solutions, particularly in their ability to scale and offer a seamless user experience across different carriers. This platform confronts these issues, presenting a new approach to the landscape of automation logistic expedition and post-purchase experience.

1.2.1 Unified data model

Existing solutions often lack a unified approach to data handling, which complicates integration with different carriers. Our platform introduces a unified data model normalising data formats across all carriers, simplifying the integration, leaving the complexity of understanding different models to the platform itself. This fits very well with the goals presented in the Project goals section. Specifically, **G1** and **G4** are closely related to the complex integration of external systems.

1.2.2 Centralized dashboard

Having of a user-friendly dashboard makes it convenient to monitor and manage shipments. However, in the presented solutions, most of it is left to the

existing ERP which is usually not made to handle logistics data. This shortcoming again reflects very well few of our goals, **G1** and **G2**. A cloud based multi-tenant solution trying to streamline logistics operations for the operators in the warehouse lacking a user-friendly dashboard would be very difficult to operate and would probably require each integrator to create their own interface for data presentation, which we do not want. Providing a modern web application with a dashboard accessible from anywhere gives a great competitive advantage and improves the platform operator's experience. With an overview of all data sent and retrieved from carriers and direct functionalities for label and consignment list printing, the user does not have to use any other interface when working with parcels.

1.2.3 Parcel status unification

The set of parcel statuses provided by the carriers is, of course, very distinct. Every carrier API is different; hence, it provides different data and communicates in a different way. Consolidation of various statuses into a standardised set, allowing users to easily understand and manage shipments without getting lost in carrier-specific environments. Supporting our goal of **G4** simple integration with an existing system, where a company typically tries to convert shipment statuses from a carrier into a more uniform format. It will, of course, also help with the practical demonstration of the integration resulting from **G5**.

1.2.4 Branded tracking and notifications

Enhancing post-purchase communication is often overlooked, yet it directly supports the objective outlined in **G3**. Businesses usually leave this channel to the third party, such as carriers, and focus on prepurchase marketing, which is usually, in digital marketing, standard Pay-per-click (PPC) adverts. However, PPC campaigns often rely on cookies to target and retarget ads based on user behaviour. As this segment becomes more regulated and with the reopening of the discussion on updating ePrivacy legislation [1] in the European Union, obligations start to come from ad service providers themselves, such as the Google V2 consent mode [2]. Leaving aside the fact that most smaller online retailers generally do not even reflect the changes and ignore them, thus putting themselves at a disadvantage in online advertising, a large proportion of customers are becoming more and more difficult to reach. Communicating with a customer, when the most important part of the purchase is happening, could be a key to greater brand recall in today's advertising overload.

1.2.5 Simplified integration

The technical challenges and costs of implementing and installing a software solution on premise might act as a barrier to many businesses. The cloud-based

[1] The EU's ePrivacy Regulation, initially established in 2002, along with the General Data Protection Regulation, both influence the tracking of visitors from the EU.

[2] The Google V2 consent mode allows website owners to adjust how their Google tracking tags behave based on the consent status of their users. The consent status might be set through the cookie bar.

solution eliminates these obstructions while supporting both the objectives outlined in **G2** and **G4**.

1.2.6 Role-based access control

Security across the platform demands a control over user permissions. By implementing role-based access, we can improve security and ensure that users have the appropriate permissions for their role.

1.2.7 Versatile carrier communication

Businesses often work with multiple shipping carriers with different contracts and settings. For example, each warehouse might require a different contract with the carrier when it is in a different region. This can be difficult to manage. Hence, it is necessary to implement a solution that can manage multiple locations (projects) in one profile with different carrier API credentials and carrier settings while distinguishing between shipments and users from different locations. This can simplify integration complexity in a multi-location environment while supporting the **G4** goal.

2. Analysis

After framing context of the platform by presenting both motivation and goals with previewing related solutions, we take a closer look at the overall analysis for our software. The purpose is to present the requirements placed on the system, primarily determined by the standard ordering process in the eCommerce sector. In addition, the requirements resulting from the planned test deployment will be presented as well as the definition of the specific requirements that will guide the development of the platform. This analysis is the foundation for a solution that not only meets technical specifications but also addresses practical business needs within a logistics sector and day-to-day usage.

This chapter will begin with a necessary introduction to the order dispatch process 2.1. Understanding this process is necessary to frame the context in which the entire system operates and in which users operate. Then, after understanding the context, in the 2.2 section, an approach to testing the system will be presented, both from both the integration and user perspective. Finally, after defining the environment in which the system is set and a way to verify the functionality, we can go to the 2.3 section. This section will present the functional and non-functional requirements on the system.

2.1 Order dispatching process

This section dives into the general life-cycle of an order from the moment it is placed to the final delivery. We will take into account the most simple and straightforward approach, which is usually the starting point for many companies and warehouses. Defining this process helps to understand weaknesses and identify opportunities for automation and efficiency improvements. Suppose a customer of a company is shopping in an e-Commerce store:

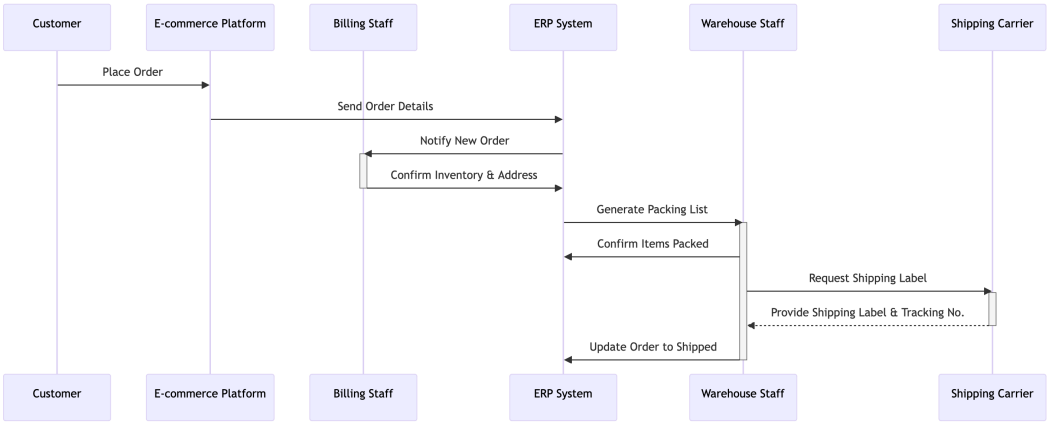


Figure 2.1: Sequence diagram of order dispatching process

1. **Order placement:** Customer completes the checkout process with the shipping details and preferred shipping method. Order information is saved in the e-Commerce platform's database.

2. **Order transfer to ERP:** The order details are automatically transferred to the ERP system. This transfer can occur at scheduled intervals or automatically, depending on the integration setup between the e-Commerce platform and the ERP.

3. **Order confirmation and inventory check:** The operator of the ERP system in the billing department processes the order with the validation of the shipping address and confirms the order.

4. **Packing list and invoice generation:** Once the shipment is confirmed, the ERP system generates a packing list that details the items to be shipped. At the same time, an order invoice is created.

5. **Uploading shipments data:** With the items collected and packed, the next step involves generating a shipping label. Order data, including recipient information and insurance, are exported from ERP to the format accepted by the carrier and uploaded to the carrier interface to retrieve the tracking number for each order.

6. **Synchronizing tracking number with ERP:** The list of selected orders in ERP is altered with the tracking number retrieved from the carrier.

7. **Shipping label and consignment list printing:** After orders receive tracking numbers, the shipping labels and the consignment list are downloaded from the carrier interface. The labels are then affixed to the consignments.

8. **Shipment dispatch:** Shipments are handed over to the shipping company courier after signing the consignment list as a confirmation of receipt.

9. **Updating status and controlling delivery:** The list of parcel statuses is manually downloaded from the shipping company interface and uploaded to the ERP system

After a brief introduction to the process, we can see that points 5-9 are quite challenging. Since the company may be working with multiple carriers at the same time, we get into a situation where the operator has to repeat these points for each carrier, making the process unsustainable and very time-consuming. Not to mention that the company has to adapt to each carrier and create data exports and imports for each carrier separately. In addition, the process of updating shipments is very complex and prone to errors. For a visualisation using the sequence diagram, refer to 2.1.

2.2 Real-world applicability

Platofrm's real-world applicability will be verified through integration and testing within an operating company. Practical implementation will focus on incorporating three major shipping carriers in the Czech republic - Česká Pošta, PPL, and Packeta - to allow the company to make a seamless transition to use the platform. This means that the platform will gain three carrier implementations

with testing to offer to the rest of the user base. Together with testing integration capabilities with external systems, namely SAP Business One, it will be necessary to create a connector module presented in chapter 7.

2.3 Requirements

This section introduces the concept of software requirements. In general, requirements are descriptions of the system's functionalities and what it should do while reflecting the needs of actors. Requirements can be classified into two groups [3]:

1. *Functional requirements:* describes how the system should react to particular inputs and how the system should behave in particular situations.

2. *Non-functional requirements:* constraints on the services of functions by the system. Including development process constraints and constraints defined by some standards. Non-functional requirements often apply to the system as a whole, rather than individual features.

The whole system has three expected user roles:

1. **Operator:** Role attributed to individuals employed by the company, interacting with the platform through its user interface.

2. **Developer:** Individual in this role utilises the platform's API for developing third-party integration.

3. **Customer:** This role represents the end recipients who are waiting for the packages dispatched by the company using the platform.

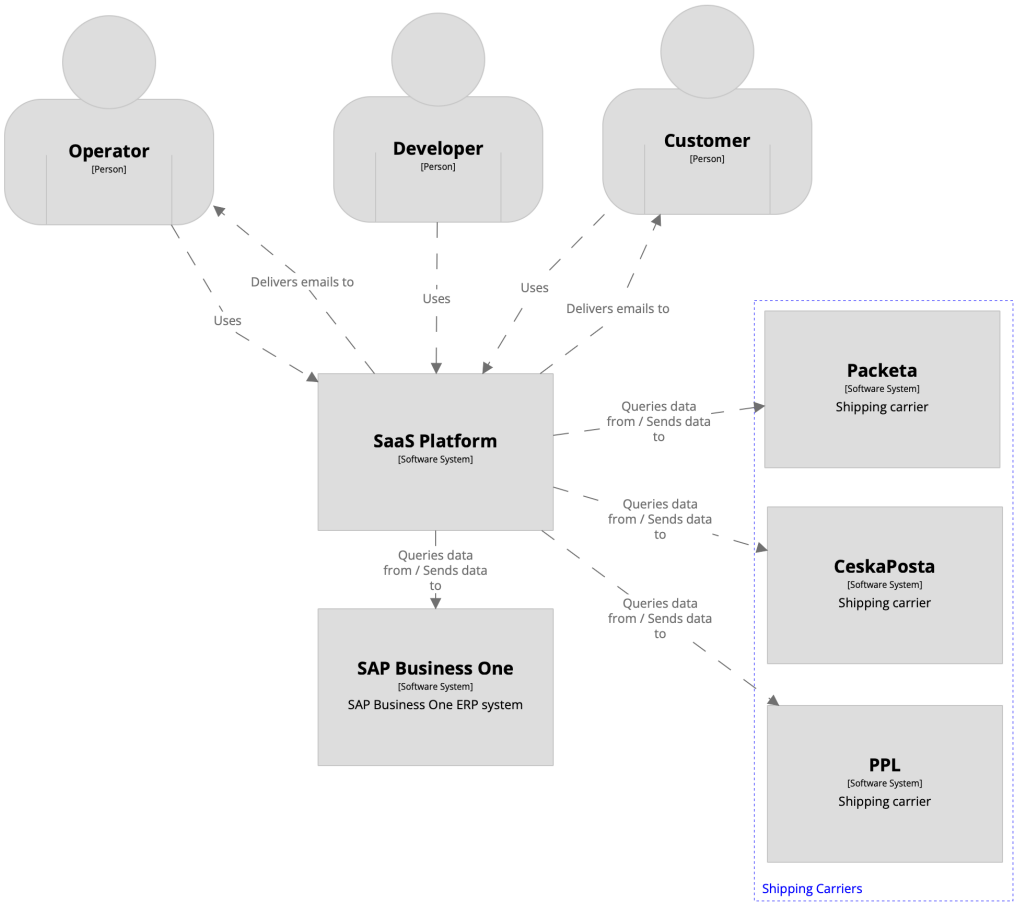


Figure 2.2: C4 diagram with system context

To provide context, the platform operates as a SaaS model and involves three key actors. We will demonstrate integration capabilities using SAP Business One, along with production integration with three carriers: Packeta, PPL, and Česká Pošta. For a visual representation of the system context, refer to Figure 2.2.

2.3.1 Functional Requirements

After reviewing the related work presented in Chapter 1 and analysing the process detailed in Section 2.1, an initial set of requirements was created. These were primarily derived from readily available information, such as documentation and technical descriptions of existing solutions.

The whole list was then discussed with the company management (including IT and marketing) where the platform will be deployed for testing; see Section 2.2. At the same time, the requirements were continuously communicated to the company's warehouse staff, who are considered as **operators** described in 2.3 and will use the platform to get an overview of the processes and various situations that occur regularly and irregularly.

This newly acquired information provided the opportunity to design the requirements so that it would fit perfectly into the company's daily operations. The requirements were then slightly modified according to our own requirements for

the platform, such as the limitation to the possibility of using one instance by multiple users, i.e. the platform should be designed as Software as a Service.

FR1: Operators can sign up and verify their accounts using a verification code sent to their provided email address.

FR2: Operators can change password to their accounts using a verification code sent to their provided email address.

FR3: Operators can log into their accounts using valid credentials.

FR4: Operators can switch the interface language between Czech and English.

FR5: Operators can create multiple projects within their account to manage data separately (e.g., for different warehouses or companies).

FR6: Operators can select and work within a specific project.

FR7: Operators can rename any of their projects.

FR8: Operators can delete any of their projects.

FR9: Operators can configure a default shipper for all shipments within a project.

FR10: Operators can configure settings for shipping carrier APIs, including authentication (e.g., tokens, IDs, secrets) and other required fields.

FR11: Within each project, operators can create multiple sellers to customise the location and branding of the tracking page and the email notifications.

FR12: For each seller, operators can set the name, localization, and branding elements such as logo, primary colour, contact information (URL, email, phone, store name), and social media links (Facebook, Instagram, YouTube, TikTok). Operators can also enable customer email notifications for specific parcel statuses.

FR13: Operators can remove any seller from a project.

FR14: Operators can enter edit mode of the seller.

FR15: In seller edit mode, operators can switch views between web and email to preview customer-facing pages and emails.

FR16: Operators can generate API access tokens for developers to use.

FR17: Operators can switch between projects to which they have access.

FR18: The operator can invite other operators to the projects.

FR19: Operators can invite new operators to collaborate on projects.

FR20: Project collaborators can be assigned different roles (Owner, Admin, Member) with varying levels of permissions.

FR21: Operators can view a paginated list of all shipments.

FR22: Operators can adjust the number of shipment items displayed per page.

FR23: Operators can navigate through the shipment list pages (next or previous).

FR24: Operators can easily identify shipments by carrier (using colour coding and names) and those created on the current day directly from the list.

FR25: Operators can apply filters to search through shipments based on textual data (reference, email) using four criteria (equal to, contains, starts with, ends with), date-time data (creation date) using a range picker, and list types (carrier, status) selecting multiple values.

FR26: Operators can select multiple shipments across carriers and perform bulk actions on the selected items.

FR27: Operators can send shipment data to carriers for all selected bulk shipments.

FR28: Operators can generate shipping labels for selected shipments.

FR29: Operators can generate a consignment list for selected shipments.

FR30: Operators can initiate the creation of a new shipment with a single click on the shipment list page.

FR31: When creating a new shipment, operators can specify details such as recipient, insurance, payment method and amount, carrier, and carrier services.

FR32: Operators can add multiple parcels to a single shipment.

FR33: Operators can preview or delete shipments after they have been sent to the carrier.

FR34: Operators can edit or delete shipments before they are sent to the carrier.

FR35: The system will automatically update the status of the shipments.

FR36: If allowed by the seller, a notification email is sent to the customer when the status of the package is updated.

FR37: Developers can retrieve all project shipments through the API.

FR38: Developers can create or update individual or multiple shipments through the API.

FR39: Developers can list all parcels from the project via the API.

FR40: Developers can retrieve labels for selected shipments through the API.

FR41: Developers can initiate the sending of shipment data to carriers for selected shipments via the API.

FR42: Customers can receive branded email notifications about updates in the status of the parcel when permitted by the seller.

FR43: Customers can view the tracking page, customised with the seller's branding, displaying the parcel statuses.

2.3.2 Nonfunctional Requirements

The non-functional requirements were shaped by understanding the broader operational context. These requirements focus on the quality attributes of the platform.

2.3.2.1 Usability

The user interface should be intuitive, requiring minimal training for warehouse and billing staff. The dashboard of the platform is designed primarily to be used with a mouse and keyboard on standard desktop screens, but should also support touch interactions for versatility. In addition, the tracking page is optimised primarily for touch interactions on mobile phones to enhance accessibility and ease of use for customers on the go. Provide user documentation, including guides for key processes.

2.3.2.2 Extensibility

The system should be able to easily integrate new APIs of the shipping carriers according to user demands. Any new carrier integration should be seamlessly incorporated into the existing system, ensuring that from a user's perspective, the interaction remains uniform across all carriers. This means that the user can initiate shipping processes with a single action, regardless of the carrier, allowing the system to handle the specifics in the background.

2.3.2.3 Scalability

Design the system to scale effortlessly to accommodate increases in both user base and request volume. The deployment strategy should enable automatic scaling based on current load, ensuring consistent performance even during peak operational periods. This approach ensures that the system remains responsive and efficient as demand grows.

2.3.2.4 Maintainability

To ensure that the source code is clean and easy to maintain, we will adhere to recognised coding standards and best practices. Specifically, we will use the Airbnb coding standard, which is widely respected for maintaining high-quality code in JavaScript environments. Furthermore, ESLint will be employed as a linting tool to automatically check for errors and enforce these standards consistently throughout the development team. Use a CI/CD pipeline for simple deployment and minimal downtime.

2.3.2.5 Multi-tenancy

The system must support a multi-tenant architecture, allowing multiple companies to use the service simultaneously while keeping their data isolated.

2.3.2.6 Integration

Offer an API that supports integration with external systems with clear documentation. Authentication should be handled by generating a long-lived token.

2.3.2.7 Customization

Allow for easy user customisation, including branded tracking pages and email notifications, to maintain consistency with the brand identity.

3. Architecture

The architecture of a software system can be much more than a simple assembly of technological components; it might serve as a blueprint for the project that determines its layout and sets its future direction. In essence, software architecture is a structured approach to development that supports system functionality and ensures that it meets current and future needs, as stated in [3]. This chapter delves into the essential role of software architecture in project development, offering a foundational understanding for readers unfamiliar with the concept. Moreover, it addresses how the architecture underpins the system's ability to meet a range of non-functional requirement introduced in the previous chapter, what approaches to architecture can be chosen in our case, and presents architecture of our platform.

The significance of software architecture can be related to architecture in the building industry. Just as architects design buildings while aiming to meet specific purposes, needs, and environments, software architects design systems to meet specific operational standards and goals. Providing a clear visualisation and description that can help stakeholders easily understand system's structure. Sets a direction for all the following design and development activities by describing the structure of the system, its components, and their interactions. Establishing a software architecture early in the project enables a common understanding between all parties involved, developers, designers, and business stakeholders. This shared understanding is an important factor in aligning project goals with technological implementations and helps manage expectations throughout the project lifecycle.

In the previous chapter 2, we have introduced the concept of software requirements presented in Section 2.3. In this context, non-functional requirements presented in Section 2.3.2 play a pivotal role. Non-functional requirements describe not what the system does but how it does it. These are the parameters that enhance the functionality and make the software robust, usable, and maintainable. Let us reiterate the architectural requirements set in the previous chapter and how they impact the architecture itself.

- **Usability:** In the terms of architecture, focusing on usability influences both ends of the system - what users see and what they don't see. The system must support a responsive interface that adapts to different devices, desktops for administrative tasks, and mobile devices for tracking. This leads to the modular design, where the separation of components helps handle user interactions and data processing.

- **Extensibility:** To accommodate future expansion, such as the addition of new shipping carriers, the architecture is designed around the plug-and-play model. This involves defining clear interfaces for carrier modules, allowing new carrier integrations to be added without disrupting existing functionality. The system interacts with the carrier module through a standardised API encapsulating the complexity and ensuring that new features can be integrated.

- **Scalability:** The architecture supports scalability through both vertical and horizontal scaling strategies. The use of stateless principles in the development, system can scale out across additional servers without issue of data consistency or user session management.
- **Maintainability:** The system's architecture is segmented into manageable components that follow the single-responsibility principle, making them easier to maintain and update.
- **Multi-tenancy:** The multi-tenant architecture is critical for efficiently serving multiple businesses simultaneously on the same platform while ensuring data isolation. This requirement goes hand in hand with others, such as scalability and maintainability. It also requires data isolation and the associated storage and access requirements.
- **Integration:** The architecture includes a comprehensive API layer that not only supports internal operations but also offers external integration capabilities. The API layer should be designed using REST principles.
- **Customization:** To support a higher levels of customization, the architecture allows clients to define branding of their tracking pages and notification emails according to their identity, without altering the core functionality. The backend supports this by managing customise elements as configurable parameters stored per tenant, which the system applies dynamically at runtime.

Having discussed the non-functional requirements that shape our system architecture, it is essential to explore different architectural approaches that could potentially meet these criteria. This step involves considering different architectural approaches, as the final choice of architecture will significantly affect the way the system is structured and how it functions.

3.1 Architectural approaches

The process of selecting an architectural approach involves evaluating several well-established patterns, each offering different benefits and trade-offs. Among these, the event-driven architecture, client-server architecture, and multi-tier architecture are well-known and considerable approaches. Each approach has unique characteristics that could enhance or detract from the non-functional goals of our system. Firstly, we will present them separately, then compare them and finally choose the pattern that suits the best.

3.1.1 Event-Driven architecture

As stated in [4] Event-Driven Architecture sits around the production, detection, consumption, and reaction to events. An event is a significant change in state, or an update that has something of interest as occurred within the system. As see in Figure 3.1 architecture comprises three main components: event

producers, event routers, and event consumers. This architecture enhances responsiveness and can be highly scalable due to its asynchronous nature, which is ideal for systems that require real-time updates and asynchronous processing. However, Event-Driven architecture can be complex to implement and maintain due to its distributed nature and the difficulty in tracing event chains and debugging.

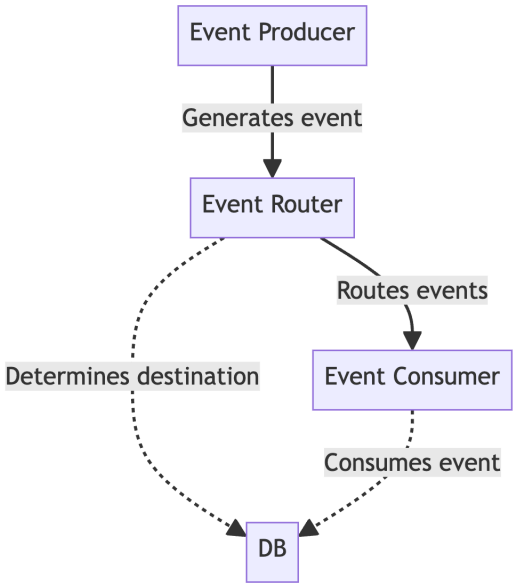


Figure 3.1: Event-Driven architecture diagram

3.1.2 Client-Server architecture

Client-server architecture divides system into two entities: clients who request services and servers that provide the services. As stated in [5] the functional characteristics of a client and a server are examples of programs that interact with each other within an application. The functionality of this architecture is highly flexible, as a single server can serve multiple clients as seen in Figure 3.2 or a single client can use multiple servers.

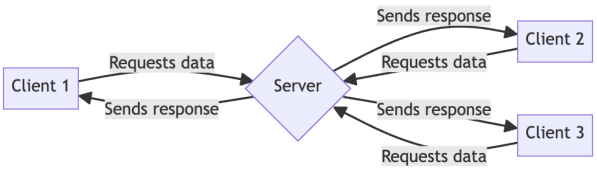


Figure 3.2: Client-Server architecture diagram

3.1.3 Multi-Layer (N-Tier) architecture

Multi-layer architecture, often referred to as n-tier, organises a system into logically separated layers that each handle specific types of processing as can be seen in Figure 3.3. Typically, these include a presentation layer (user interface), an application layer (business logic), data layer, and database layer (data management). This separation helps better organization and allows for independent scaling, maintenance, or updating of each layer. Supports scalability and simplifies the development process by allowing teams to work on different layers independently.

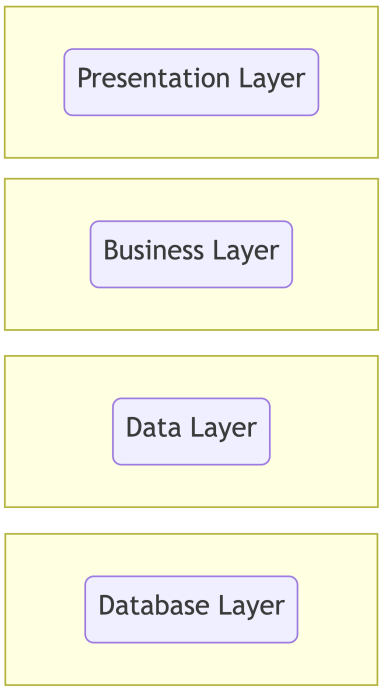


Figure 3.3: Multi-Layer architecture diagram

3.1.4 Comparison and selection

Comparing these architectures, the event-driven architecture offers high responsiveness and is good for systems that require real-time capabilities due to its asynchronous capabilities. The client-server model provides a robust architecture for handling interactions between centralised servers and multiple clients, which makes it suitable for traditional web applications. The multi-layer architecture offers flexibility in development and maintenance by separating concerns across multiple layers.

For our platform, the three-tier architecture, a specific form of multi-layer architecture, appears most suitable. This architecture divides the application into the presentation tier, logic tier, and data tier, which aligns well with our requirements for a scalable, maintainable system that can efficiently handle multiple user interactions and complex business processes. Additionally, this pattern complements our need for a multi-tenant environment. Since the communication within layers is not cross-tier, we can support isolation between different tenant data. The three-tier architecture as shown in Figure 3.4 provides a balanced approach, offering a clear separation of concerns while maintaining simplicity in connectivity between the client and the server. It allows for efficient data processing and easier scalability management, as each layer can be scaled independently based on demand.

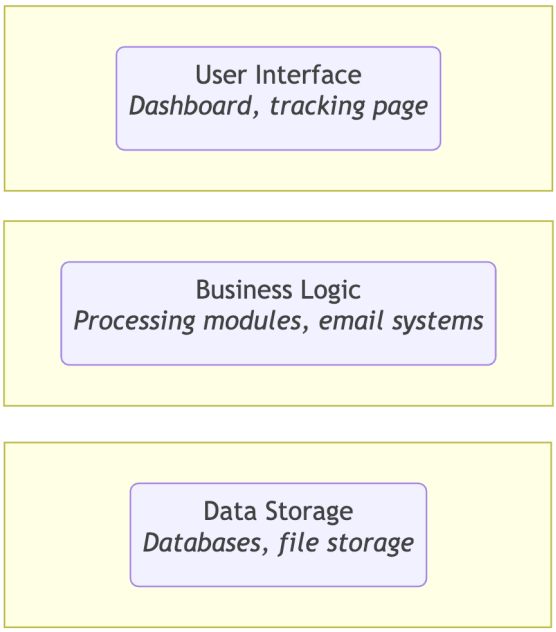


Figure 3.4: High-level layered architecture diagram

3.2 System Architecture

With the selection of a three-layer architecture as the most suitable model for our platform, we now present the architectural components proposed. This section outlines the high-level structure of the system, focusing on the main components and their roles without delving into detailed implementation specifics. Let us take a look at the high-level diagram shown in Figure 3.5 and describe the components shown in the figure.

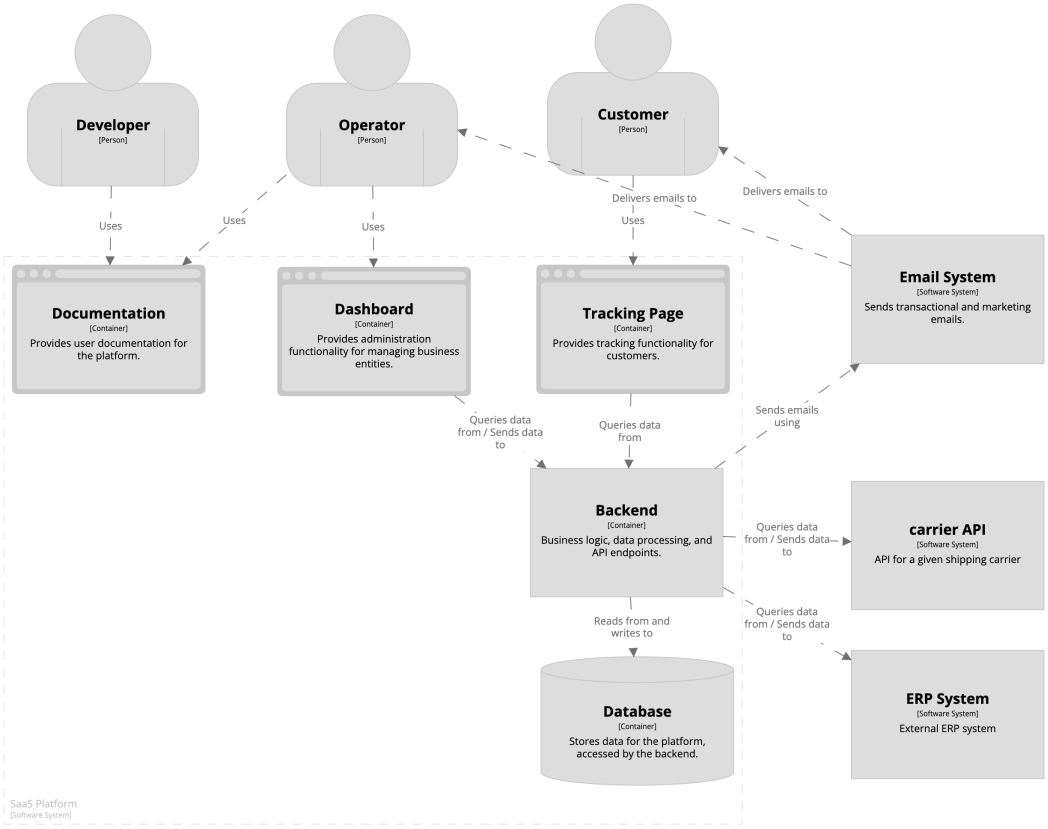


Figure 3.5: C4 container diagram of the software system

3.2.1 Frontend Components

The frontend of the platform consists of three main components, each serving a different purpose:

- **Dashboard:** The central user interface for operators. It enables operational management, including shipment tracking, carrier management, and analytics. The dashboard is designed to be used primarily in desktop environments, but is responsive to be used on smaller screens.
- **Tracking Page:** An interface that allows end customers to track their shipments. This page supports custom branding, allowing businesses to provide a cohesive brand experience. Optimised for touch interaction, it improves accessibility and ease of use on mobile devices.
- **Docs:** A documentation website that provides users with guidelines, API documentation, and setup tutorials. This component is crucial for onboarding new Operators and supporting existing ones by offering easy access to necessary technical and usage information.

The backend serves as the core of the platform, integrating with external systems and APIs from shipping carriers, handling all data synchronisation tasks, shipment dispatch, and updates between the platform and shipping carriers. The business logic processes data from the frontend to ensure that all operations adhere to all business processes within our domain, which consists of shipment

processing, user management, and the generation of customer notifications. In addition, an email system is integrated to manage communications with users by email based on specific triggers and events within the platform. This system is designed to support customizable email and tracking page templates, which allows for alignment with the branding requirements of different tenants, enhancing the customisation capability of the platform.

3.2.2 Database

The database serves as the central repository for storing all operational data along with user information. The database schema is designed to support multi-tenancy, implementing data isolation strategies that keep tenant data separate and secure. This setup is crucial in adhering to the multi-tenancy requirements of the architecture, ensuring that each tenant's data is accessed and managed securely without interference from or to other tenants.

3.2.3 Overall System Interaction

The frontend communicate with the backend via secure, REST API, which abstract the complexity of business processes and shipping carrier integrations. The backend processes requests, interacts with the database for data retrieval and storage, and communicates with shipping carries or requests email notification sending. It also opens another REST API for secure communication between the system and external services.

3.3 Conclusion

This architecture provides a blueprint for the development and operation of the platform. It supports the non-functional requirements outlined in the previous chapter 2.3.2, such as scalability, maintainability, and extensibility, while also providing a flexible and user-friendly environment for both operators and customers. All while being able to serve multiple tenants at the same time.

4. Technical design

In today's fast paced technological landscape, the large number of options from spectre of programming languages and associated frameworks presents both an opportunity and a challenge at the same time. As we move into the technical details of the architecture presented in the previous chapter 3, it is important to recall an idea mentioned in [3]. Establishing a robust architecture in the early stages of development is key because it will become significantly more expensive in future phases than at the beginning. These phases and decisions are very closely linked. Because, as important as how we lay out the application, it is equally important how and with what we write it.

In this chapter, we examine the technical decisions that shape the development and operation of the platform. We will explore the selection of programming languages and frameworks, detailing how these choices work together to create a robust, scalable foundation for the system. Additionally, this chapter will address an approach to a multi-tenancy design paradigm - a key architectural feature that enables us to efficiently manage resources and serve multiple clients within a single application instance. Now, with an added layer of detail regarding the programming languages, frameworks, and technologies, this architecture can be brought to life.

4.1 Programming Language and Frameworks

Choosing the right programming language and frameworks is a crucial decision that influences not only the development process but also the longevity of the software itself. Impacts every phase of the development life-cycle, from initial implementation to maintenance and the capacity to scale in response to future demands. In the following section, the platform technology stack will be presented with the reasoning behind these decisions and the alternatives taken into consideration. We will go thought programming language selection and its runtime, as well as supportive frameworks.

The core technologies that form the backbone of the platform include TypeScript for programming, React for the frontend development, Koa as the backend framework and PostgreSQL for data management. Let us dive into more detail and reasoning behind these decisions.

4.1.1 Programming Language

In the world of full-stack web development, TypeScript has evolved as a popular choice for both frontend and backend development, largely due to its widespread and support from community. The decision to use TypeScript across the entire stack is aligned with the project's goals of scalability, maintainability, and productivity.

Static type checking with TypeScript offers significant advantages in terms of code quality and reliability. It makes TypeScript a more verbose and complex language to write, but in the long run and in such a large project it helps to create a more self-explanatory code-base. One of the primary reasons for selecting

TypeScript is its ability to provide a similar developer experience across both the frontend and backend. This enables to easily transition between working on a client and server-side code with minimal context switching.

We can see a strong upward trend in the popularity of TypeScript. Meanwhile, JavaScript continues to have its first place as the most used programming language according to both the Stack Overflow Developer Survey from 2022 [6] and 2023 [7], number of developers actively using TypeScript grows. Placing it at the fifth place of the survey in both years in the professional developers' community. This ensures a wealth of resources, tools, and libraries. Since TypeScript is a superset of JavaScript, we can also consider it the winner of the survey.

4.1.1.1 JavaScript and TypeScript

JavaScript is a dynamically typed language. This means that variable types are determined at runtime. This flexibility allows fast development but can introduce errors that are hard to catch until the actual code is executed.

```
let myVar = 'Hello, world!';
myVar = 100; // This is valid in JavaScript
```

Listing 4.1: JavaScript dynamic typing example

However, TypeScript introduces static typing, allowing developers to specify variable types. This catches type errors at compile time, leading to more reliable code.

```
let myVar: string = 'Hello, world!';
myVar = 100; // Error: Type number is not assignable to string.
```

Listing 4.2: TypeScript static typing example

JavaScript lacks a built-in mechanism for enforcing the structure of objects. This can lead to different inconsistencies in object shapes during the execution.

```
const a = [
    {
        name: 'Bob',
        age: 30
    },
    {
        name: 'Alice'
    }
]
```

Listing 4.3: JavaScript different object shapes

On the other hand, TypeScript provides interfaces and type aliases to define the structure of objects. Making the code more predictable and easier to debug.

```
interface IPerson {
    name: string;
    age: number;
}

const a: IPerson[] = [
```

```
7       {
8           name: 'Bob',
9           age: 30
10      },
11      {
12          name: 'Alice'
13      } // Property age is missing in type { name: string } but
        required in type IPerson.
14  ]
```

Listing 4.4: TypeScript enforcing object shape

In conclusion, while JavaScript's flexibility makes it suitable for small projects or prototypes requiring quick iterations. TypeScript type system and object management features provide a more structured and error-resistant approach. These attributes are crucial for developing complex applications, making TypeScript the preferred choice for enhancing code quality and long-term project sustainability.

4.1.1.2 Alternatives

When deciding on the programming language for full-stack web development, Python was a strong consideration. With its large community, popularity, and robust web-frameworks Flask and Django, Python offers an interesting ecosystem for web development. The simplicity and readability of Python make it an attractive option, especially for fast prototyping and projects with a strong focus on developer productivity.

Dynamic Python typing creates challenges for larger and more complex applications. Although dynamic typing offers flexibility and development speed in the early stages of development, it can lead to type-related errors that are only caught at run-time. Recent versions of Python introduced optional type hints that allowed developers to specify types for variables similar to TypeScript. However, these hints are not enforced by the Python runtime itself. This adds a layer of type safety, although it remains optional and not as integrated as a TypeScript type system.

Performance benchmarks, as presented in [8], demonstrate Node.js, and therefore JavaScript, performance compared to Python in real world scenarios. JavaScript should generally outperform Python in the measured scenarios. However, the choice of technology lies in the effectiveness of the developer with a specific language and framework. While Python developer experience and large number of libraries make it a strong candidate, the advantages offered by TypeScript type safety and JavaScript performance make TypeScript a more strategic choice for our needs.

4.1.1.3 Compilation and execution

After choosing TypeScript as the go-to language, in the context of platform technical design, it is important to understand how this language integrates into execution environments. The client side of the platform will run in browsers, which cannot execute TypeScript directly. The same applies to the backend - given the deployment requirements, we are limited to Node.js. This implies that our TypeScript code must be compiled into JavaScript.

4.1.2 Frontend

Given the popularity of JavaScript/TypeScript web development, the number of options when choosing the go-to library is substantial. This choice influences the development experience and affects the application's long-term maintainability. Among the many options, ranging from Vue.js, to AngularJS - React emerges as the library of choice for the platform. Coupled with Create React App (CRA) [1], this combination offers a solid foundation for development needs. This decision leverages the existing knowledge base and optimises the workflow.

4.1.2.1 React

Developed by Meta Platforms, React has become one of the most popular libraries for building User Interface (UI). According to the survey [7], React is one of the most common web technologies used by the respondents. The declarative approach of React allows us to create complex UIs from isolated pieces of code called "components" [9] within a virtual Document Object Model (DOM), a lightweight JavaScript representation of the real DOM. Those components can be of two species; more on that later. They usually rely on the extended JavaScript so-called JSX syntax. As stated in the React documentation [10], JSX allows one to write HTML-like markup inside a JavaScript file, keeping the rendering logic and content in the same place.

As already mentioned, React was the go-to frontend library chosen for the platform. Given its large community that contributes to its large number of tools, supportive libraries, and resources, it is a strategic choice. There are two main approaches to working with React. Let us take a look at both of them.

4.1.2.1.1 Class components

Initially, React development was heavily based on class components. Each component encapsulates the behaviour and state within a class inherited from `React.Component`. It must have explicitly stated `render()` method returning JSX. Class components allow one to define life-cycle methods, for example, inside the `componentDidMount` method.

```
1 import React, { Component } from 'react';
2
3 class Welcome extends Component {
4   render() {
5     return <h1>Hello, {this.props.name}</h1>;
6   }
7 }
```

Listing 4.5: React class based component exmaple

4.1.2.1.2 Functional components

In recent years, React community experienced a large shift from Class-based components towards functional components. This change was brought about by

[1] As of writing this thesis, CRA is obsolete and no longer directly supported by React

the concept of hooks. Hooks let developers use React features like state access or life-cycle methods to the functional components. With hooks, the developer can set a state, propagate context to nested components, or even cache a component or some sort of calculation.

We can simply migrate the class-based component 4.1.2.1.1 into a functional component:

```
1 import React, { useState } from 'react';
2
3 const Welcome = (props) => {
4   const [name, setName] = useState(props.name);
5   return <h1>Hello, {name}</h1>;
6 }
```

Listing 4.6: React class based component exmaple

Given these options, the obvious variant of functional components was chosen. This approach aligns with modern React best practices and external library integration.

4.1.2.2 Alternatives

As mentioned previously, there are several alternatives to React for web development in the TypeScript environment.

- **Vue.js:** JavaScript framework usually highlighted by its simplicity. Vue is written in JavaScript/TypeScript with HTML-based template syntax. Vue uses the so-called single-file components. Special file formats that allow one to encapsulate the template, logic, and styling of a Vue component are a single file [11]. Similarly to React, Vue.js uses virtual DOM.
- **Vue.js:** Developed and maintained by Google, is a full-fledged Model View Controller pattern (MVC) framework providing much more functionality than React and Vue.js out of the box for the price of higher complexity and unnecessary features given the project architecture.
- **Svelte:** Represents an interesting alternative to React given its performance orientated approach, eliminating the runtime overhead of virtual DOM by shifting the work to compile time. This produces highly optimised vanilla JavaScript.

While all options offer interesting features and different approach to problems of web development, React was chosen for compelling flexibility, strong community support, and large ecosystem.

4.1.3 Backend

Choosing the right backend framework in Node.js was a key decision. This part of the application should carry all the business logic and complexity associated with a multi-tenant architecture. Therefore, it was important to carefully select a robust solution that would be sustainable and scalable in the long term. The decision was to adopt Koa over popular frameworks, for example, Express.

4.1.3.1 Koa

Koa [12] is a web-framework for Node.js designed by the Express team. However, the aim is to have a smaller and more robust foundation for a web API. Koa stands out with its "middleware-first" architecture, a principle that places a chain of middleware functions executed upon request. This offers significant advantages for use-cases of the platform requiring different levels of authorisation, and data isolation mechanism between tenants.

4.1.3.2 Middleware architecture

At the core of the Koa philosophy are the middlewares used to streamline the handling of HTTP requests. The Koa middleware is designed to be reusable, allowing for a highly flexible and modular system that can be adapted to most use cases. The middleware in Koa is a JavaScript function attached to the endpoint as an array. Each middleware can perform operations, make changes to the requests, and the response objects even with top to bottom propagation of data. As a good example, a slightly modified logging middleware used in our Koa backend can be presented.

```
1  import Koa from 'koa';
2  import { Logger } from '../utils/logger';
3  import { container } from 'tsyringe';
4  import { RouterContext } from '@koa/router';
5
6  export const requestLoggingMiddleware = async (ctx: RouterContext
       , next: Koa.Next) => {
7    const logger = container.resolve(Logger);
8
9    // Don't forget to clean body to not disclose sensitive values
10   logger.info('Started handling request', {
11     path: ctx.path,
12     method: ctx.method,
13     body: ctx.request.body,
14   });
15
16   await next();
17
18   logger.info('Completed handling request', {
19     path: ctx.path,
20     method: ctx.method,
21     body: ctx.response.body,
22     status: ctx.status,
23   });
24 };
```

Listing 4.7: Koa logging middleware

We can clearly see that we can perform both request and response operations. The middlewares are chained directly in the router of the app like:

```
1    router.get(
2      '/projects/:projectId',
3      requestLoggingMiddleware,
4      authenticationMiddleware,
```

```
5       authorizationMiddleware([Role.ADMIN, Role.OWNER, Role.MEMBER
        ]),
6       getProjectAction
7     );
```

Listing 4.8: Koa router example

In this example, we can see a sample GET route with three chained middlewares before the actual action execution.

4.1.3.3 ORM a data models

For managing the database and building data models within Koa backend, Knex.js and Objection.js was used. Knex.js serves as a query builder, allowing for direct interactions with the database. Used together with Objection.js, an Object–relational mapping (ORM) built on top of Knex.js, it is an efficient way to manage and interact with database entities in an object-like structure. Implementation details will be explored in the following chapter 5 as well as in the programming documentation found in A, focusing on the implementation of the application itself.

4.1.3.4 Runtime

Building an application to run as a Lambda function in the Node.js runtime ensures that scalability is built into the core architecture. The Serverless framework simplifies the deployment process, allowing for seamless updates, management, and scheduled runs of Lambda functions. By leveraging serverless technologies, we make sure that the backend can accommodate varying request loads with minimal overhead.

4.1.4 Database Management System

Selecting a Database Management System (DBMS) that aligns with application's data complexity and requirements is always a crucial. The backend needs to perform complex data retrievals and also needs to store the possible configurations and customisation of tenant's data, namely the branding layouts and shipping carrier configurations. The data stored in the database are mostly structured with few mentioned exceptions. Given that PostgreSQL was a good choice given its performance and ability to store complex data types.

4.2 Multi-tenancy and its possible approaches

Multi-tenancy refers to a software architecture approach, designed for cost efficiency and ease of maintenance. It's key principle is to simulate, otherwise needed on-premise deployment or a dedicated instance of the software, on a single instance. This model works with the premise that the data are kept isolated from each other using several possible approaches. Let us define key terms and take a look at possible approaches to this interesting architectural model.

- **Tenant:** As stated in [13], the tenant is a group of users who share the same view on the application they use. The view usually includes the data they access, the configurations shared between the groups, and much more. Usually, tenants are from different legal entities; hence, a tenant can be a company, for example.
- **Single-tenancy:** For completeness and a better understanding of the forthcoming information, it is good to define the term "single-tenancy". It is an architecture in which a single instance of a software application and supporting infrastructure serves one tenant. This approach is usable for a SaaS software, however, comes with a cost where for each tenant it is necessary to pay for additional infrastructure resources. In practice, this approach is commonly used when moving old on-premise software to the cloud with a dedicated deployment pipeline that sets up each instance based on tenant demand.

4.2.1 Approaches

After defining the foundational concept, it is important to consider how to approach design of the multi-tenancy - balancing between security, cost efficiency, and ease of maintenance. Let us take a look at the possible approaches to it proposed in both [13] and [14] and suggest the best way that best suits our needs.

4.2.1.1 Multiple databases

Multiple databases approach illustrated in Figure 4.1, know as "database level tenancy" is very similar to the proposed single-tenancy. In this setup, each tenant uses a separate database, hence maximising the data isolation. This can lead to the best possible data isolation in SaaS multi-tenant software, but to worse maintainability.

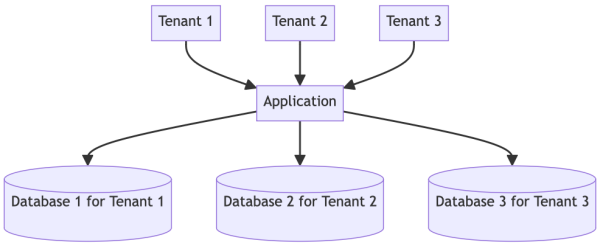


Figure 4.1: Multi-tenancy with multiple databases

4.2.1.2 Single database, multiple schemas

This model involves a single database with multiple schemas, also known as ”schema-level tenancy”. Each schema serves a different tenant. Infrastructure cost is significantly reduced compared to the previous approach, bringing some implementation complexities.

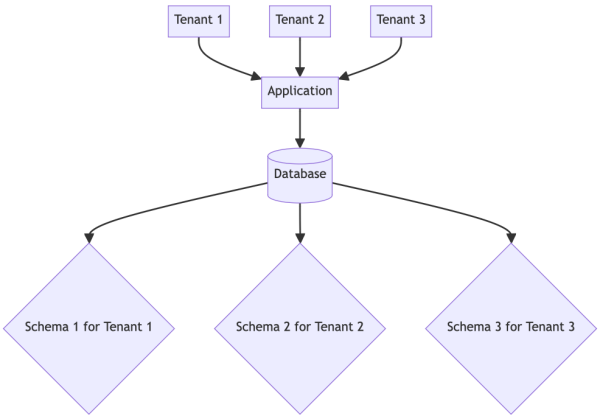


Figure 4.2: Multi-tenancy with multiple schemas

4.2.1.3 Single database, single schema

Known as ”record level tenancy” is designed that all tenants share the same database and the same schema. Tenant’s data are stored in the same tables differentiated by column or columns containing a tenant identification. This approach necessitates strict data isolation within application queries, as the application layer is the only enforcer of data separation.

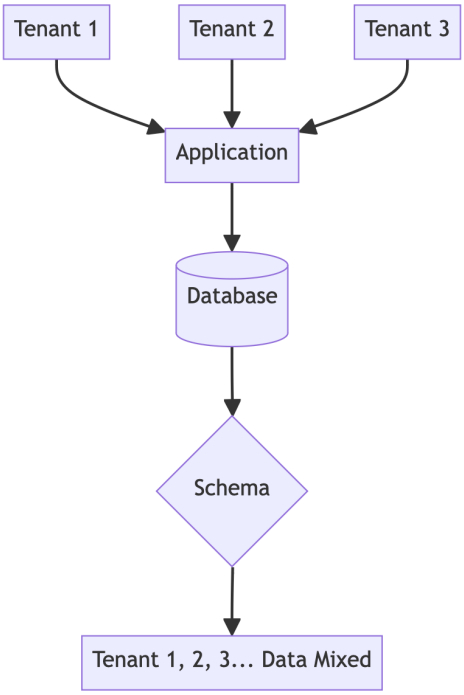


Figure 4.3: Multi-tenancy with single database and schema

4.2.2 Implementation in the platform

The backend of the platform achieves multi-tenancy through the concepts of ”Projects”. Projects are entities that bundle multiple users into a single tenant, adopting the ”record-level tenancy” seen in Section 4.2.1.3.

This setup allows users to share project-biased data among themselves with role-based access, ensuring that the data of each tenant are isolated and secure. Without giving much detail that would compensate for the clarity of the design in figure 4.4 we can see a simple UML diagram of the relation ship. Both `User` and `project` have many more relations; however, these have been removed for now. Data isolation is ensured through ORM queries that are project-biased, thus preventing accidental data leaks between tenants.

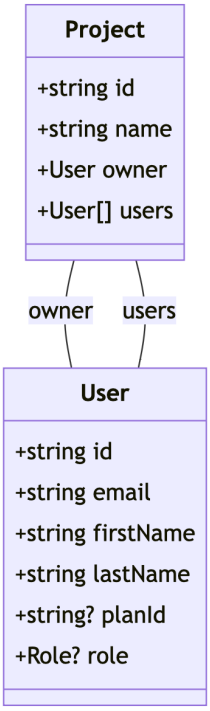


Figure 4.4: Simplified UML diagram of the User and Project relation

5. Implementation

In the preceding chapters, we have dived into the architectural 3 and technical design 4 of the platform, highlighting the theoretical and strategic decisions that form our system. This chapter transitions from conceptual outlines into concrete details of the implementation phase serving as a bridge connecting the high-level design decisions discussed earlier with the technical details covered in the programming A and administrative D manual.

This chapter will explore the structure and organisation of the project. We will delve into the backend implementation, focusing on how it manages multi-tenancy - a feature allowing the system to serve multiple tenants without sacrificing security. Special attention will be paid to the integration of carrier modules, which are essential for the platform. Moreover, the chapter will cover the implementation strategies for web clients, developed with ReactJS.

5.1 Project Structure

Choosing an appropriate project structure is a fundamental decision in software development that significantly impacts the efficiency of the development and maintainability of the project. For our platform, the choice between a Monolithic repository (monorepo) and multiple repositories (multi-repo) was critical. This decision influences the cooperation in future development, how the project is integrated and deployed, and how changes are managed across different parts of the project.

A monorepo refers to a development strategy where the code for multiple projects is stored in the same repository. This approach is contrasted with multi-repo, where each project or service has its own repository. For this platform, the use of a monorepo has offered several benefits. With all code in a single repository, managing dependencies becomes easier. There is no need to publish internal packages that are used across services, and updates to shared libraries are reflected across whole codebase. This reduces the risk of the so-called dependency-hell[1] and simplifies upgrades. CI/CD pipelines can be more efficiently managed when all projects share the same repository. Changes in one part of the platform can trigger build in another, ensuring integration and consistency across the platform. This setup simplifies the process of rolling back the changes in all affected parts when necessary. Monolithic repository also ensures that all components of the project are always synchronised with each other. The unified versioning approach arises from the compatibility of the individual services, especially when making API changes or updating shared libraries. On the other hand, monorepo also brings some challenges that need to be considered, particularly around the permission management of the repository. In the multi-repo setup, access can be controlled at the repository level, allowing for straightforward management of who can access what. In addition, multi-repo might be more appropriate in large-scale projects. As the repository grows, so

[1] A term describing frustration when multiple packages have dependency on incompatible version of the same package.

does the time to clone and the consumption of resources for the automatic CI/CD pipelines. However, this can be solved by adopting shallow cloning[2] and defining pipeline strategies to determine which parts of the project need to be rebuilt based on the changes made.

After weighing the benefits and challenges, a monorepo approach was chosen for the platform. This decision was driven by the streamlined dependency management and easier integration between services. In addition, the simplicity of the deployment process and overall project management significantly influenced this choice. The monorepo structure not only simplifies the operational workflow, but also enhances the ability to manage the project efficiently as it scales. This approach ensures that all components of the system are in-sync and can be updated or rolled back simply.

5.2 Backend Implementation

The backend of our platform plays an important role in orchestrating the workflows and managing multi-tenancy. Built using the KoaJS framework as a REST API, it provides a robust foundation for the entire platform. In this section, we will provide some implementation specifics of the backend with a description of the logic in it. The whole backend is build around few key components:

- `entites`
- `services`
- `actions`

The `entites` define Objection.js `Model` representing a database table where each instance of that class represents a table row. The `services` define an injectable dependencies which are used for database queries using an ORM Objection.js. And finally `actions` representing the API action called from the API endpoint.

5.2.1 API Design and multi-tenancy

The design of the API tries to adhere to RESTful principles, aiming to provide a clear and logical representation of information with stateless operations. Each endpoint is crafted to meet specific business requirement and corresponds closely to the entities with retrieval, creation, update, and delete operations.

The key part of the backend is handling authentication and authorisation. In order to respect the Don't repeat yourself (DRY) principles, both of these operations are implemented as KoaJS middlewares. Given our deployment strategy described in Chapter 6, we could have gone in the direction of using an Amazon Cognito as authentication. However, this would bring vendor lock-in in a fairly critical part of the application logic. It would be a reversible solution, but it could present a serious problem and interfere with the whole system. That is something

[2]Option in a `git` allowing to start working in the repository without downloading every version of every file in the entire history.

we have decided to put off; hence, custom middleware was implemented to verify access tokens sent with each protected request, and methods handling token generation and regeneration.

After authorisation and authentication, in tenant bias endpoints with a parameter containing project ID in the route, the parameter is stored and used strictly for data retrieval.

5.2.2 Carrier modules

To implement the different shipping modules, the backend uses the abstract class `AbstractCarrierModule`, which defines the interface for all shipping modules. Each specific implementation of a carrier module, in the current state of the platform, for example, for Packet, PPL or Česká Pošta, extends this abstract class and implements its methods to convert generic operations into carrier-specific API calls. This design pattern encapsulates the variability between different carriers, providing a unified interface to the rest of the application. It simplifies the addition of new shipping carriers, as only a new module inheriting from the abstract class needs to be created without altering the existing system.

5.2.3 Sending e-mails

Email communication is an integral part of the platform, used for notifications and confirmations. The email sending functionality is externalised through Amazon AWS Simple Email Service, leveraging the `SendEmailCommand` and `SESv2-Client` from the `@aws-sdk/client-sesv2 package`. This approach decouples the email sending capability from the application logic, allowing scalable and reliable email delivery managed by the AWS infrastructure.

5.2.4 Generating waybills

Generating waybills is a functionality provided by the backend, especially for the shipping operations when the warehouse is handing the parcels physically over. The backend utilises the `pdfjs` library to create PDF documents. This library was selected after evaluating alternatives such as `jsPDF` and `Puppeteer`. However, these alternatives are based on a browser rendering. This significantly simplifies the whole development process since we can render and export HTML templates. However, being browser-based also meant a heavy reliance on a dependency such as Chromium for example. This posed a challenge in the Lambda environment due to execution time and resource constraints, including dependencies. The `pdfjs` library, on the contrary, offers a more lightweight solution that fits well within the serverless architecture, providing quick and efficient PDF generation without the overhead associated with browser-based rendering engines.

5.3 Web Client Implementation

Web clients, primarily developed using ReactJS, is a crucial component of the platform. Provides the user interface through which operators and customers interact with the system. This section will discuss the client-side part of our

platform, focusing on routing, state management, and the integration of support for the multi-tenancy and dynamic functionality of the platform.

5.3.1 Client-Side Routing and State Management

Client-side routing is implemented using the `react-router-dom` library, which manages page navigation between different components without refreshing the page. The state within the application is managed using combination of React Context API and a local state management through hooks such as `useState`. For global state management, particularly for user authentication and project selection which is critical for maintaining multi-tenancy, the Context API provides a way to pass data through the component tree without having to pass props down manually at every level.

The operators dashboard supports multi-tenancy by storing the currently selected project ID by the tenant within the URL. This ensures that all tenant-specific data fetched from the backend are scoped within the selected project. For both authentication and project management, dedicated page wrappers were created to handle the front-end logic. `AuthenticatedRoute` manages the routes that require user authentication but aren't meant to render tenant-specific data, only user-specific. `ProjectRoute`, indirectly extending the `AuthenticatedRoute`, on the other hand serves as a project fetcher based on the project ID in the URL. The indirect extension is meant as follows: if no project is returned for a given URL, the user is redirected to the page used to select the project. If this request fails to recover the expired access token, the user is logged out. It ensures that the user is not only authenticated, but also has the necessary permissions to access data related to a specific project.

API calls are abstracted into reusable hooks defined in an `actions` directory. These hooks provide methods to interact with the backend, handling CRUD operations for defined entities. Each action hook fetching data from backend utilizes the `executeApiAction` which standardises API call processes including error handling, success message rendering, as well as token refreshing if needed.

To ensure components have access to the necessary data without pop-drilling [3], React Contexts are used. Context providers are set up at higher levels in the application to store user details, current project setting, and much more. This method helps to make data updates and access more efficient throughout the application.

[3]Prop drilling is the process of passing down data or state through multiple layers of a component hierarchy.

6. Deployment

When building a SaaS platform meant for various users in various environments, it is important to ensure that the platform is not only adaptable and scalable, but also robust and secure. This necessity is the foundation on which this chapter is built. Combining modern cloud technologies with good practices, this section explores how these elements are used in order to ensure smooth and efficient deployment process. With a focus on Infrastructure as Code (IaC), this chapter highlights how this method is used within the AWS ecosystem providing in-depth look at the deployment procedure for both frontend and backend components discovering how they directly impact scalability, reliability, and security of the platform.

6.1 Current Deployment Strategy

The deployment strategy for this platform leverages AWS services with focus on IaC to automate and manage cloud infrastructure. Thanks to this approach, creating a consistent deployment process is achieved while reducing possibilities of human error and ensuring replication across different stages or environments.

Specifically, the strategy uses AWS S3 for hosting the static frontend(s), AWS Lambda for backend functionalities including scheduled background tasks and organising these diverse components into multiple stacks using AWS CloudFormation. Furthermore, this platform utilizes the PostgreSQL database using Relational Database Service (RDS), AWS Route 53 for domain routing, AWS Certificate Manager for SSL/TLS certificate management and AWS Simple Email Service for securing a high email delivery rate. This structure allows for a well-controlled infrastructure, allowing quick adjustments if needed.

6.1.1 AWS S3 for Static Frontend Hosting

Deploying static frontend applications of the platform employs the AWS S3 to host the React applications. AWS S3 provides a reliable, scalable and secure solution for serving static content, making it ideal choice for hosting a Single Page Application (SPA) applications like ours. All three frontend applications (documentation, tracking page and dashboard) use very similar deployment strategies. The S3 buckets are configured to serve a website with `index.html` with allowing public access and establish removal policies to ensure that the buckets are destroyed when needed. However, we cannot provide access to the S3 bucket just like that. The definition of AWS Cloud Front distribution (Amazon Content Delivery Network (CDN)) is vital to catch errors and unauthorised accesses by enforcing a Secure Sockets Layer (SSL) certificate managed by AWS Certificate Manager. Finally, Domain Name Server (DNS) routing for applications is configured with AWS Route 53, creating an A record that points to the AWS Cloud Front distribution. For a detailed visualisation of the deployment architecture for static frontend applications and how these components connect, refer to the diagram illustrated in 6.1

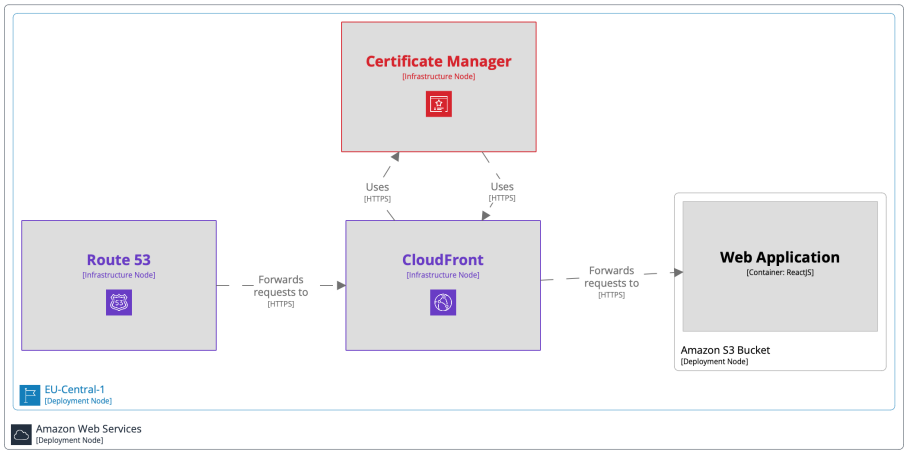


Figure 6.1: C4 Deployment diagram of static web application in AWS

6.1.2 AWS Lambda for Backend Services

For the backend services, the platform leverages AWS Lambda, a serverless computing service that runs code in response to events with automatic management of underlying computing resources. This choice supports a serverless architecture for the backend, benefiting from scalability of Lambda functions, which scale automatically based on the number of requests and are quite cost-efficient by charging only for the compute time consumed. Such a pricing strategy is important for our platform, primarily operational during the Central European working hours, ensuring that resource allocation during off-peak hours - such as nights, early morning and weekends - is minimized, thus aligning resource usage with the actual demand.

The entire `Koa` backend is encapsulated within AWS Lambda functions using the Serverless framework. This setup not only streamlines the deployment and operation of the serverless backend, but also enhances its extensibility and maintainability. The Serverless framework handles the integration of backend application into the AWS ecosystem, enabling leveraging the full spectrum of benefits of serverless computing.

Additionally, the backend is expanded with serveral scheduled tasks, configured to emulate traditional cron jobs within the AWS Lambda environment. These tasks are important for routine operations, such as fetching parcel statuses from carrier APIs and sending tracking emails to recipients. Specific Lambda handles are designed for database seeding and migrations tasks which are necessary for the deployment process. These handlers are invoked as part of the CI/CD.

The IaC approach for the deployment of the backend service is orchestrated thought AWS Cloud Development Kit (CDK), enabling automated provisioning of cloud resources. Key elements of the deployment include the following:

- **AWS Lambda:** The main building block of the backend service, AWS Lambda functions are integrated to run code in response events.
- **Data storage and management:** A PostgreSQL hosted on AWS Relational Database Service (RDS) provides a managed, scalable and secure relation database solution that accompanies automated tasks such as backups and patching.
- **Network configuration:** A dedicated AWS Virtual Private Cloud (VPC) is provisioned to encapsulate Lambda functions, ensuring that they operate within a secure and isolated network environment. Security groups within AWS Virtual Private Cloud (VPC) define access rules, providing a security layer for backend interactions.
- **API Gateway integration:** An API Gateway acts as the entry door backend services, managing incoming API requests and routing them to the appropriate AWS Lambda function.
- **Domain management and SSL/Transport Layer Security (TLS) encryption:** The deployment also uses AWS Route 53 for domain routing and AWS Certificate Manager to manage SSL/TLS certificates.
- **Static content hosting**: AWS S3 buckets are integrated to host static assets used for user-uploaded public content.

For a more in-depth look, refer to the diagram illustrated in 6.2

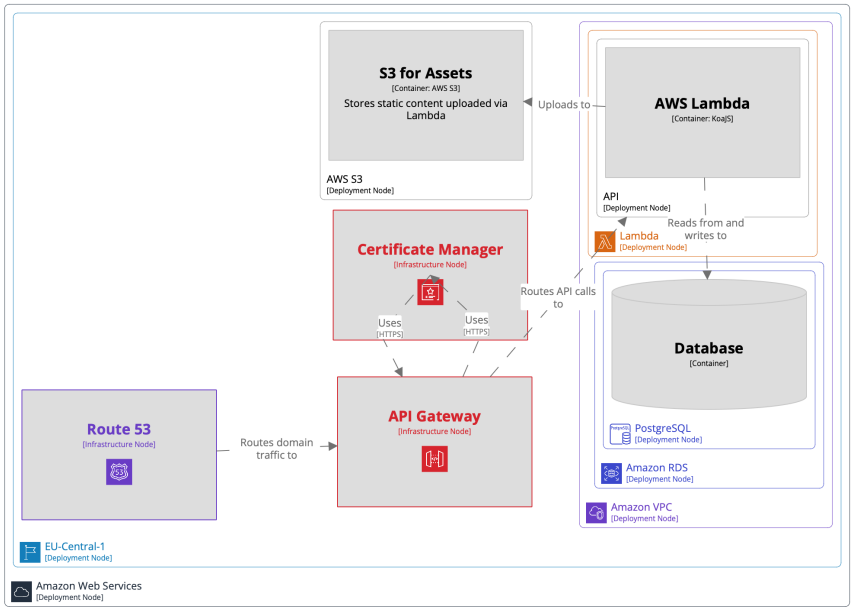


Figure 6.2: C4 Deployment diagram of backend service in AWS

6.1.3 AWS CloudFormation for Infrastructure Management

AWS CloudFormation plays a crucial role in managing the platform infrastructure. It perfectly aligns with the purpose of IaC - creating and managing resources with templates. CloudFormation enables the user to define the entire cloud environment as code that can be versioned, reused, and shared. Automating the deployment of resources in a consistent and repeatable manner. By organising resources into multiple stacks, the deployment can be segmented logically (e.g., networking, application layers, security), facilitating easier management and updates of specific components. Using the AWS CloudFormation, one can define the AWS Identity and Access Management (IAM) roles and policies that define the permissions and actions that can be performed on AWS resources. This policy is a key factor in reducing the scope of operations for the deployment process, ensuring that each action, from the deployment of Lambda functions to the management of log groups and the interaction with other services like S3 and RDS, is guided by a set of permissions.

In our case, several AWS CloudFormation stacks are created using AWS Cloud Development Kit (CDK) to programmatically define and manage CloudFromation stacks. As a result, the deployment process systematically manages the creation of several key AWS CloudFormation stacks, each designed to support different facets of the application's infrastructure: Having said that the deployment creates the following AWS CloudFormation stacks:

- **API Service Stack:** Central backend infrastructure, the API Service Stack comprises all the necessary resources deployed by the
  Serverless framework such as all Lambda handlers and events triggering the scheduled tasks. With necessary AWS Identity and Access Management (IAM) roles for the runtime environment as well as GitHub actions deployment.
- **Docs page stack:** Dedicated to the application documentation portal, the Docs page stack provides the infrastructure required to host and serve the documentation.
- **Tracking page stack:** Tailored for the tracking functionality, this stack establishes the infrastructure needed for the tracking page.
- **Dashboard page stack:** Focused on the administrative aspect of the application, the Dashboard Page Stack creates the infrastructure for the dashboard page.

6.2 Alternative Deployment Methods

Although the deployment path chosen for this platform is AWS with serverless architecture for deployment, it is important to explore alternative deployment methods that could offer different benefits or align with other needs. The two main alternatives that are presented are containerisation and utilisation of services from other cloud providers. After this brief overview of both options, we will present the potential risks of both of them.

6.2.1 Containerization

Containerization is a method that allows to encapsulate an application along with its environment and dependencies into a container that should run consistently on any infrastructure. Being either a local development environment or production on a remote server, containers are popular option and are very often the way to go for both development and deployment. This approach is complemented by technologies such as Docker. Docker can run completely free in both the development and production environments, significantly reducing cost. This gives freedom to the deployment environment. Containers can run on a bare Virtual Private Server (VPS) or in a container-specific environment developed to host containers without taking the costs of maintaining a server such as AWS Elastic Container Service (ECS). Containerisation offers numerous benefits, including:

- **Portability:** Containers can be moved across different environments or cloud providers without vendor-lock-in.
- **Efficiency:** Containers share the kernel of the host system, making them more lightweight and efficient than running separate VPS for each application.

6.2.2 Other Cloud Providers and Services

In today's world, choosing between cloud providers is not a simple task. One can choose between more abstract solutions (cloud platform as a service) that require less setup, hiding more complexity behind. This usually comes with some costs that are either financial or functional. The reason being is that these platforms usually run on outsourced hardware and that they should be widely accessible by application developers without expertise in deployment and server problematic. This is so because making processes simpler usually requires significant reductions in the configuration options and settings. A good example might be a Vercel, a cloud platform as a service company providing a simple-to-use solution to host web applications. However, AWS still provides a wider range of integrated services - from email sending services, to object storage, and much more. Even though Vercel is an excellent solution for hosting static-sites and frontend applications, AWS generally provides a much more configurable environment giving a wider control control over the whole platform. Cost-wise, considering the fact that Vercel is running on top of the AWS, it is expected that services will be more expensive.

6.2.2.1 Google Cloud Platform

Google Kubernetes Engine offers powerful and scalable container deployment solution. Google Cloud Run is a fully managed platform that automatically scales containers, similar to AWS Lambda, but with the benefits of containerisation. However, while Google Kubernetes Engine provides interesting container management and scalability, the overhead of managing Kubernetes can be significant. Requiring a deep understanding of Kubernetes architecture, management, and best practices, which might introduce additional complexity to the platform.

6.2.2.2 Microsoft Azure

Microsoft's Azure Kubernetes Service might provide a similar solution to Google Kubernetes Engine. This brings about the same issues related to Kubernetes complexity. However, using Azure Functions, which supports serverless computing, might be a better match to AWS Lambda, which supports an event-driven environment. The most significant difference between the two is the cold start time. An Azure function might require a cold start after 20 minutes of inactivity, taking even tens of seconds to start. The AWS Lambda usually takes no more than 1-2 seconds on cold start.

6.2.3 Conclusion

Although the current deployment strategy mainly uses AWS services and leverages a serverless architecture, it also exposes the platform to potential risks associated with vendor lock-in.

Vendor lock-in occurs when a project becomes so dependent on a particular cloud provider that migrating is technically challenging or expensive. However, the benefits of avoiding vendor lock-in must be balanced against the complexity of deployment and infrastructure management. If one wants to handle secure and scalable deployment following recommended practices on a bare metal without vendor lock-in, it becomes a very challenging task.

6.3 Continuous Integration and Continuous Deployment (CI/CD)

In every project, both CI and CD processes are important components to ensure code quality and minimize possible human error on repetitive manual task such as deployment. Using GitHub Actions as the running environment of the CI/CD pipeline allows the automation of various workflows ranging from code quality checks to deployment into two separate environments (staging and production), ensuring that every line of code in the `main` branch of the code repository undergoes through lint and build checks with follow-up deployment.

Since GitHub was chosen as the primarily code repository for the platform, it was a straightforward choice to use cloud runners in GitHub Actions as the go-to solution for the integration and deployment pipeline. When code changes in the repository, GitHub Actions are triggered to execute predefined jobs, such as typing checks with TypeScript, linting with ESLint, and the identification of spelling errors. These initial steps ensure that the code base adheres to the platform's standards and conventions.

Following quality checks, deployment workflows are activated based on pushing to the `main` branch publishing the code to both production and staging environments with all necessary migrations and updates. These workflows use AWS Cloud Development Kit (CDK) commands and AWS Command Line Interface (CLI) to deploy infrastructure changes and application updates to AWS services, effectively and relatively quickly, bringing the application from the repository to the world.

The integration and deployment pipeline is designed to ensure consistent code style with minimizing propagation of errors into the public production environment. The phases of the pipeline are the following.

- **Code quality checks:** On every pull request, automated workflow for linting, type checking and spelling is triggered. These steps are important for maintaing high code quality and catching potential issues early.
- **Build process:** For frontend applications, the build process compiles the source code into static assets. Backend services are prepared for deployment, ensuring that all necessary dependencies are correctly packaged.
- **Deployment to AWS services:** Deployment is carried out in stages, starting from staging to production environments. If something fails, the staging process is over, production is terminated, and the job fails. This phased approach allows for validation of changes in a controlled context before affecting live users. AWS Cloud Development Kit (CDK) is used to provision or update AWS resources, including Lambda functions for backend services and S3 buckets to host frontend assets. Infrastructure changes are managed through AWS CloudFormation stacks, allowing reliable and repeatable deployments. The deployment of a static web application is carried out in two phases after the infrastructure is set up. It begins by building it and then copying it to the AWS S3 bucket. The deployment of backend services, on the other hand, is carried out by the Serverless framework framework itself with AWS Identity and Access Management (IAM) policy defined within the setup phase.
- **Post-Deployment tasks:** After the build and deployment, database migrations and invalidating CDN caches ensure that the latest and valid content is served to user and that the database schema is up to date with what the application is expecting.

By integrating these steps into GitHub Actions, the platform benefits from an automated pipeline that minimises manual intervention in error-prone processes. For example, on 6.3 there are three stages of the code quality pipeline.

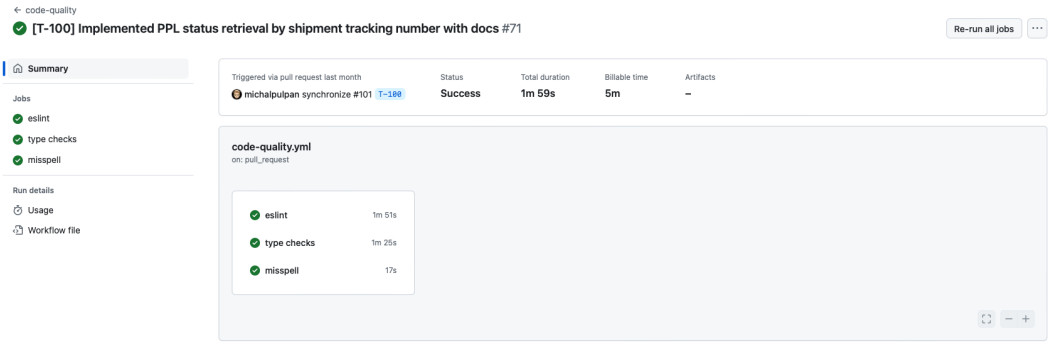


Figure 6.3: GitHub Actions - Pull request integration

On the other hand, on 6.4 we can see two phase deployment process firstly into staging environment with follow-up production for all components of the

platform. For a detailed view, we can see 6.5 where all the steps of the API production deployment can be seen.

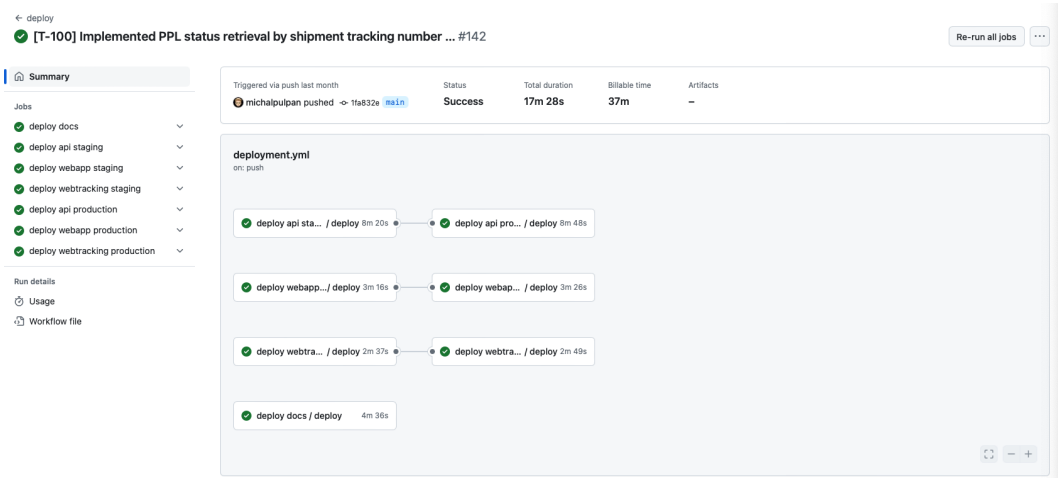


Figure 6.4: GitHub Actions - merge deployment

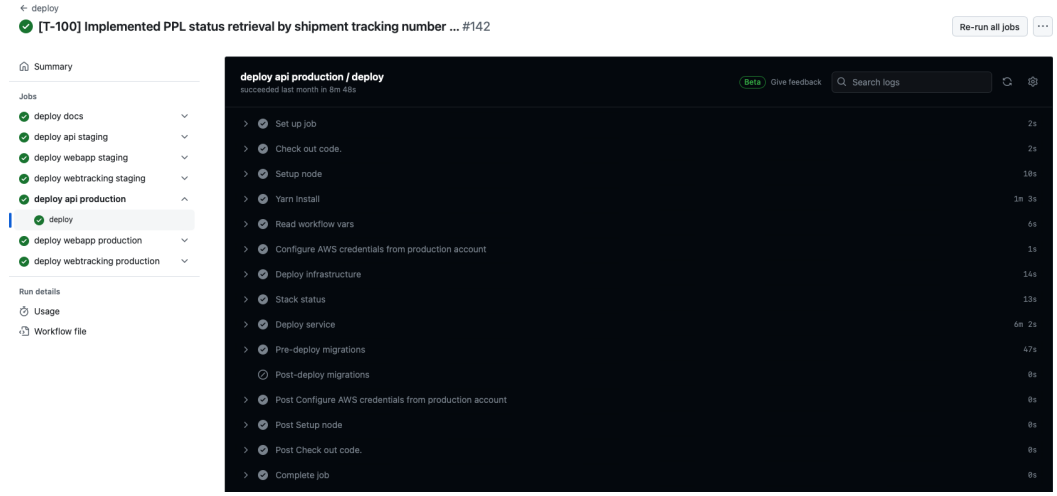


Figure 6.5: GitHub Actions - merge deployment API

This significantly speeds up the deployment cycle and ensures better stability of the application.

7. Integrating SAP Business One

In the modern landscape of daily business operations, seamless integration between external software solutions and the core ERP systems is not just a convenience; it is a necessity. As the number of external services needed for businesses to operate grows rapidly, comes the need to integrate those with minimising tight coupling and complex unmaintainable dependencies. SAP Business One, a leading ERP solution for small to medium companies, offers robust and unimaginable capabilities, but presents unique challenges when it comes to integration with third-party software.

This chapter dives into the complexity of establishing a direct connection with SAP Business One and its underlying database. Direct write interactions with database tables are highly discouraged due to potential repercussions on system warranty and technical support. However, it is important to understand that the caution advised against direct database modifications does not arise merely from overarching restrictions but stems from a recognition of the complex structure of SAP's database. Unauthorised alterations carry the risk of compromising the integrity of the system. It is worth mentioning that SAP Business One's pricing model is not only instance-based but also user-based. This introduces additional limitations and costs that businesses must consider and potentially accept. Or do they?

7.1 Possible solutions

Bridging the gap between SAP's robust functionalities and the needs of business utilising third-party software is not as straightforward as it might initially appear. In today's software environment, a common requirement is the need for a web service to programmatically transfer data between third-party applications and SAP. Despite SAP's widespread popularity, an official solution for this specific challenge was absent for a long time.

7.1.1 SAP Business One Data Interface API (DI API)

One of the foundational solutions provided by SAP is SAP Business One Data Interface API (DI API). This low-level programming interface offers direct access to SAP Business One objects, enabling developers to perform Create, Rear, Update, Delete (Create, Read, Update, Delete (CRUD)) operations on SAP data. The SAP Business One Data Interface API (DI API) was the go-to choice for many years because it was already installed with every SAP instance and the programmer could access SAP directly via `C#` interface already known from a SAP user interface. However, this convenience also introduces significant limitations. The SAP Business One Data Interface API (DI API) operates through a local *Component Object Model* (COM) that is installed alongside SAP Business One. This architecture requires that any code that uses the SAP Business One Data Interface API (DI API) must be executed in the environment where the COM is located. Consequently, this code must typically run on a Windows machine and be written in `C#`, which may not always align with preferred development

practices or the existing infrastructure of a company. Despite these challenges, a workaround exists in the form of a *wrapper library*. Although this does not address the deployment environment limitations, it enables the translation of the library's existing interface into one that is compatible with other programming languages. For example, it is then possible to port `C#` library into Python using tools such as the `makepy` library.

7.1.2 VCZ.WebService

A noteworthy solution to address several issues associated with using the SAP Business One Data Interface API (DI API) alone is the *VCZ.WebService* developed by Versino, a SAP Business One supplier. It was one of the first web services available for SAP users operating on the *SOAP* (Simple Object Access Protocol) standard. This makes *VCZ.WebService* a good choice for data transmission between SAP and a variety of third-party software. In particular, the connection to *VCZ.WebService* uses the standard SAP user licence.

This introduces key advantages, flexibility. Unlike using a pure SAP Business One Data Interface API (DI API), *VCZ.WebService* introduces a layer over the SAP Business One Data Interface API (DI API) that allows third-party software to run in various operating systems and environments. However, *VCZ.WebService* is not without its disadvantages. In today's world, using Simple Object Access Protocol (SOAP) is not considered a modern approach. Most programmers seek Representational state transfer (REST) services which more align with the modern architectural styles and preferences. Since the log-in is done using a standard SAP user, a programmer using *VCZ.WebService* has to use a licence provided by the company to use only for the WebService, raising security concerns. Furthermore, at the time of writing this thesis, *VCZ.WebService* is gradually being phased out in favour of newer technologies introduced in 7.1.3.

7.1.3 SAP Business One Service Layer

The introduction of SAP Business One Service Layer marked an evolution in SAP integration capabilities. Launched with version 9 of SAP Business One, the *Service Layer* is a modern REST-based interface that handles communication with SAP systems. The SAP Business One Service Layer is controllable only using HTTP operations, making it accessible from any programming environment able to perform HTTP requests, thus vastly broadening its applicability. It offers a well-documented, standardised way to interact with SAP objects and perform operations similar to the ones in ERP's user interface. Featuring user-defined queries and the ability to patch and post securely to the SAP database. User-defined queries are an interesting feature. They are normal SQL `SELECT` queries with the requirement to first be stored as a string in the SAP database and then called by the SAP Business One Service Layer for data retrieval. It is safe in this way, but limiting and time consuming for the user. Authentication still relies on the SAP user licence, generating a short-lived token through SAP Business One Service Layer introducing an overhead when using this service. Both limitations will be discussed later in this chapter 2 including a solution proposal. The transition from SAP Business One Data Interface API (DI API)

to adopting SAP Business One Service Layer reflects a broader trend toward web-based APIs for enterprise integration. However, being a first-party solution and providing key features with seamless SAP data manipulation, it still lacks the features needed for fast data queries and it's own authentication. Business does not want to provide it's own licence for which they have to pay extra and raise a security concerns with exposing the licence.

This project proposes a new approach to overcome these challenges. By introducing a publicly accessible solution through a reverse proxy equipped with its own authentication policies.

7.2 SAP Business One Service Layer Proxy with direct Database connector

Integrating SAP Business One with external applications such as e-Commerce platforms, different warehouse solutions, and our platform - presents complex challenges that existing approaches fail to address. These challenges call for a new solution (or at least an enhancement of the existing one) that allows secure access to the Service Layer over the Internet without compromising SAP credentials and thus creating SAP user accounts for each user of the API. This solution should not only allow new integration capabilities but also ensure that business can maintain the security and integrity of the SAP Business One and it's database.

7.2.1 Analysis

As we came to the conclusion, existing solutions for integrating SAP Business One with external applications fall short of meeting requirements of business and are not very straight forward to use in few aspects. This analysis explores the needs for creating a proxy for the SAP Business One service layer with a direct database connector.

7.2.1.1 Functional requirements

FR-SAP1: Implement own authentication system without compromising SAP credentials.

FR-SAP2: Maintain a unified SAP login session across all user interactions.

FR-SAP3: Forward requests/responses to/from the SAP Business One Service Layer.

FR-SAP4: Implement a direct route to execute `SELECT` database queries bypassing the SAP Service Layer.

FR-SAP5: Provide the ability to switch between production and development environments for both the SAP Service Layer and the direct database connector.

FR-SAP6: Implement a simple user management system for CRUD operations on users of the Proxy.

FR-SAP7: Provide authorization tools for role-based access to create administrators and users.

7.2.1.2 Nonfunctional requirements

- Local deployment close to SAP Business One Instance.
- Continuous Integration and Continuous Deployment (CI/CD).
- Ensure that the API is accessible via the HTTPS protocol from the public network.
- Expose the API Proxy endpoint under a public domain name.

7.2.2 Architecture

The architecture is designed to ensure seamless integration between external applications and SAP Business One via SAP Business One Service Layer and direct connector to the Microsoft SQL database underlying the SAP instance. The main part of the architecture is an application that serves as a proxy and manager of singleton connectors to the SAP Service Layer and Microsoft SQL database in both development and production environments. This application is strategically positioned behind the `NGINX` reverse proxy which serves as the entry point for all inbound requests.

The architecture consists of following key parts:

- Reverse proxy
- Proxy app
- Proxy database
- SAP Business One Service Layer
- SAP database

As illustrated in Figure 7.1, the architecture within the company network can be categorised into two principal systems. One being Proxy Server and the second SAP Business One.

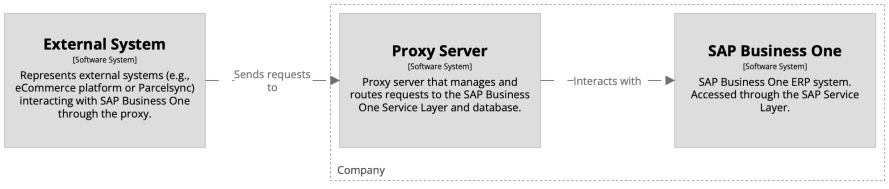


Figure 7.1: C4 Landscape diagram of communication with SAP

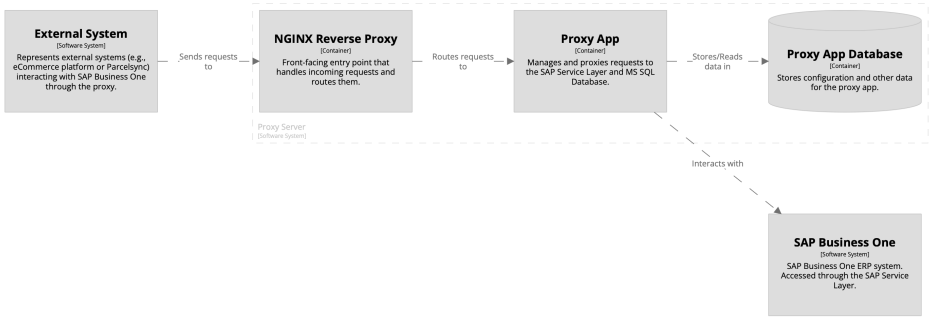


Figure 7.2: C4 System diagram of Proxy App

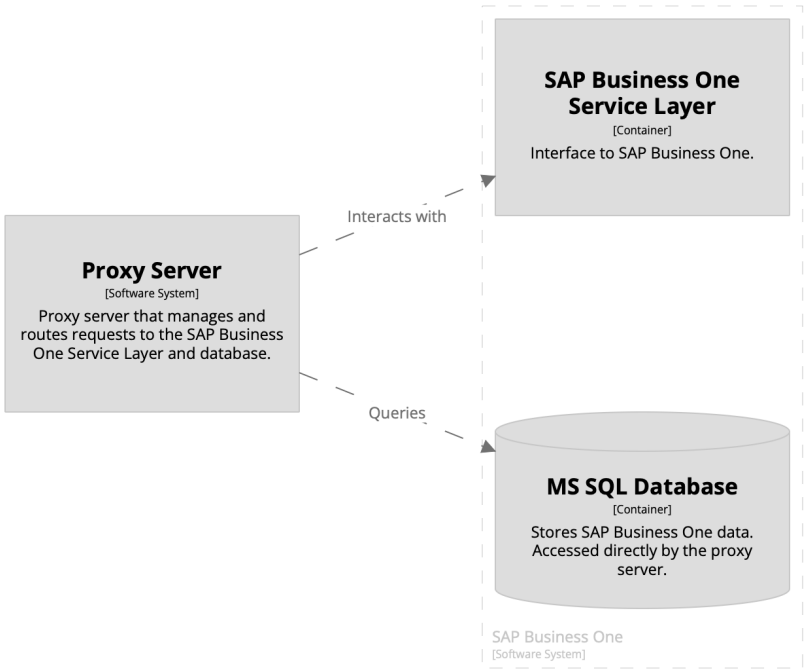


Figure 7.3: C4 System diagram of SAP Business One

A more detailed examination provided in Figure 7.2 presents the components that make up the Proxy Server, including:

7.2.2.1 Reverse proxy as the entry point

As the entry-point, the front-facing reverse proxy was chosen. Managing and directing incoming traffic to the Proxy app service

7.2.2.2 Proxy app and database

The proxy application with its own database is the heart of the system. They are responsible for user management as well as maintaining SAP Service Layer access tokens for both environments and connection pools to both Microsoft SQL database environments.

In contrast to the Proxy system, within our scope, the SAP Business One system is visualised in Figure 7.3 consisting of the following key components (simplified for clarity):

7.2.2.3 SAP Business One Service Layer

Running instance of SAP Business One Service Layer installed locally on the server with SAP Business One and the database. The Service Layer is accessible via HTTP on a given port.

7.2.2.4 SAP Database

Underlying database used by the SAP Business One instance. In our case, we are talking about a Microsoft SQL database.

7.2.3 Implementation

In the landscape of application development, especially when creating an application that serves as an API proxy, developers are presented with large array of options and tools. To ensure good maintainability, we have opted to remain within the JavaScript ecosystem by leveraging similar technologies used in the 3. This strategic choice not only leads to a more efficient development process, but also enhances existing expertise and resources by building on familiar technologies.

7.2.3.1 Technology Stack

- **Nginx** front-facing reverse proxy managing incomming traffic to the Proxy API.
- **Koa** makes up the backbone of the Proxy API as the backend framework. Its lightweight and middleware-orientated design allows for flexible and modular codebase.
- **PostgreSQL** is used to store SAP service layer access tokens as well as user credentials.
- `Objection`, `Knex` are tools configured to provide seamless integration between Koa backend and our database. Objection.js provides a simple to use ORM where glsknex serves as a query builder and database migration management tool.
- **Yarn** is chosen as a package manager and script executor.

7.2.3.2 Proxy API structure

The Koa backend is structured into several entities, middlewares, services, and actions that ensure modular and maintainable code. Objection Models used for Object Mapping entities to the Database.

- `User` basic model used for authentication and authorization. Storing username, password, and role (user/admin).
- `SAPToken` persistent singleton instance of a environment bias token having just token, environment, and expiration time.

Each middleware is designed to perform specific functions and, if needed, store additional data in the context passed to the next chained middleware or a final action.

- **Authentication middleware**
- **Authorization middleware**
- **SAP Environment middleware**
- **SAP Service Layer Login middleware**

Services usually perform CRUD operations on a database using a package Objection.js as ORM. They are most often called from actions; however, in some instances, they are called directly from one of our middlewares.

- **SAP service** provides a log-in to generate, store, and retrieve SAP Service Layer authentication tokens for both environments.
- **User service** provides simple CRUD and authentication methods for user objects.

At the end of the middleware chain there are actions. They serve as the main endpoint function. In our instance, we only need CRUD operations for manipulating and listing user objects, proxy for SAP Service Layer, and Microsoft (MS) SQL query method making use of pre-initialised singleton database connection pool for both environments.

7.2.3.3 Microsoft SQL connector

In order to allow for efficient SQL queries directly to the database, the Proxy API implements an endpoint that expects a query in the body. The query is then forwarded to pre-initialised MS SQL connection for either development or production database, based on the user's choice. The database connection pool is initialised on application start-up in order to minimise cold-starts. As a connector to the database, the `mssql` package for Node.js was used. However, as was later found, this package is not able to keep two distinct connection pools when calling the basic `login` method. It stored in cache only, although a different one was required. This was an issue since the requirement was to allow for connection to two different databases. However, this behaviour can be overcome by creating a `ConnectionPool` instance directly instead of relying on built-in login functions and caching it outside of the library.

7.2.3.4 SAP Service Layer Proxy

Then main part of the application is, as the name suggests, the Service Layer proxy itself. Using a static `AxiosInstance` request is passed with all necessary headers to the SAP Service Layer endpoint expecting a stream in the response. This approach, of course, creates some overhead by calling the API directly.

A series of performance tests were conducted to quantify the difference in response times. The goal was to evaluate efficiency of the Proxy API while handling authentication, authorization and data forwarding while operating remotely. As can be seen in figure 7.4, on average, direct calls to request the full `BusinessPartner` object from SAP Business One Service Layer were completed in approximately 85 milliseconds. In contrast, calls made through the Proxy API were executed in an average response time of approximately 268 milliseconds.

Although the Proxy API introduces an additional overhead, resulting in longer response times compared to direct Service Layer interactions, the increase is fairly consistent and within acceptable margins, given the added functionalities and remote location with public domain access which introduces considerable network latency.

Figure 7.4: Response time of direct SAP Service Layer call in contrast of Proxy API

7.2.3.5 Database model overview

Database model of the Proxy API is very minimalist and can be best seen in 7.5 Its main focus is to store user data in a safe way. In addition, it serves as a cache for SAP Service Layer tokens. This approach might initially appear to be an overextension, such as ”using a sledgehammer to crack a nut”. However, our user base might grow, and even for minimal usage of the production instance, it is necessary to provide reliable persistent storage.

Figure 7.5: DB diagram of database models

7.2.4 Deployment

Proxy API deployment was carried out on a VPS with Ubuntu 22.04.3, using Docker for containerisation. This approach streamlines the setup by separating the build process into distinct phases to minimise the container's footprint. The section further discusses continuous deployment via Docker Hub and the use of reverse proxy for secure internet access.

7.2.4.1 Overview

Probably the most significant part of the deployment itself is utilizing Docker, a powerful containerization platform that simplifies the process of building, shipping and running applications in different environments and contexts. The container was constructed using a multi-phase Dockerfile, which allows for a seamless streamlined setup by caching dependencies and separating the build process into several phases. This approach not only speeds up the build process, but it even significantly minimises the footprint of the container itself.

7.2.4.1.1 Dockerfile strategy

The Dockerfile was divided into multiple phases to optimize the build process

- **Dependency Caching:** The initial phase used `Node.js 18.16.1` in the `bullseye-slim` release as the base image to cache package.json file.
- **Build Process:** A temporary image created in the first phase is reused to install dependencies and build the Proxy API using esbuild, JavaScript bundler, and minifier.
- **Production Image:** The final image was prepared using the same Node.js base image as in the first phase. From this image, all development dependencies and source code were removed to ensure that only necessary components for running the application were included.

7.2.4.2 Continuous Deployment and Continuous Integration

The build and deployment process was completely automated. Automated build is done in an official Docker container registry, a Docker Hub, using GitHub integrations where all source code is stored. Every push to the master branch triggers a new build on the Docker Hub, ensuring that the latest version of the application was always available. Deployment on the Linux VPS is managed by Watchtower, an automated update tool that checks for new Docker images every 60 seconds and updates the running container accordingly. This setup allows for continuous deployment with minimal manual intervention. Furthermore, Slack notifications were integrated to provide immediate alerts on deployment status and system overall health, allowing quick responses to potential problems.

7.2.4.3 Accessing the application

To securely present the Proxy API to the Internet, Nginx was used as a reverse proxy configured with Certbot for automatic SSL certificate management. This setup ensures that all traffic to and from the Proxy API is encrypted. Using this configuration, secure and reliable gateway for accessing the Proxy API was achieved.

7.2.5 Data Sender

This module serves as an intermediary that covers the data flow between platform and SAP Business One through the Proxy API, as can be seen in 7.6 The application written in TypeScript running in a Node.js environment.

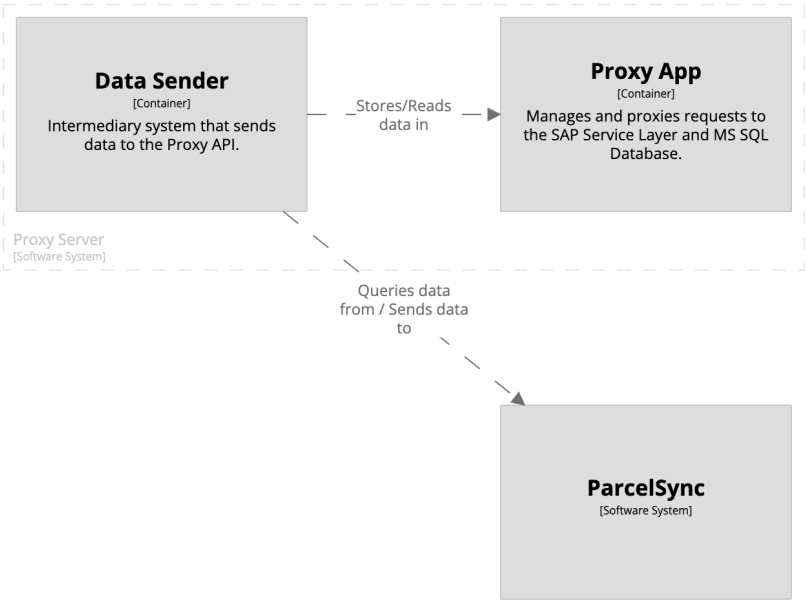


Figure 7.6: C4 Container diagram of communication with SAP

7.2.5.1 Design and Configuration

The Data Sender module integrates a scheduler that orchestrates task execution based on predefined schedules or commands. Time-schedule tasks are necessary to ensure that importing new shipments into platform or updating shipment statuses in SAP are performed efficiently and in expected times. Using the Node.js package `node-cron` for scheduling and the `yargs` library for simple CLI configuration.

7.2.5.2 Functionality

The main entry script acts as the core of the Data Sender. Initialising the application and setting up scheduled tasks. Each task is designed to address specific data synchronisation needs between platform and SAP Business One.

- **Order imports:** Divided by carriers - Packeta, PPL, Ceska Posta, this task targets the retrieval of ”unsent” orders reflecting data storage conventions in SAP Business One with option to ship shipment with multiple parcels (tracking numbers).
- **Shipping number imports:** Importing shipping numbers for recently dispatched orders back into SAP, ensuring that the data in SAP Business One stay up to date.

- **Parcel updates:** Focused on updating the information of the parcel in a specific time frame. This task updates shipment status, invoice numbers, and shipping cost in SAP, using the status mapping of the platform for efficiency.

7.2.5.3 Deployment strategy

Similar to the Proxy API 7.2.4, the Data Sender application is containerised, ensuring consistency and reliability across multiple environments. The CI/CD setup mirrors what was already described in 7.2.4.2, using automated builds and deployments to maintain up-to-date and secure operations. The container is designed to be always on utilising the scheduler to continuously check and run tasks.

8. Evaluation

Throughout this thesis, we have designed, implemented, and deployed a SaaS platform with a multi-tenant architecture aiming to fundamentally transform how eCommerce businesses interact with shipping carriers. This platform covers the need for business to manage the specifics associated with each shipping carrier by automating the communication process and hiding all unnecessary details. The platform's capabilities extend beyond mere data communication within expedition logistics; seeking into marketing corners with a tendency to present another possible marketing channel within post-purchase communication to enhance customer engagement. Key features include an automated mechanism for customer email notifications triggered by changes in parcel statuses based on shipping carriers and tracking page. Both are designed to serve as a post-purchase marketing channel with a seller's branding. Allowing businesses to maintain continuous engagement with their customers while reinforcing brand identity.

This chapter delves into the evaluation of the platform, assessing its operational efficacy and integration within a real-world business environment. The platform was integrated into a *company* that handles more than 100 parcels per day, providing a robust testing ground for all the implemented features. Integration was carried out on 14 March 2024. This evaluation focuses on several critical areas:

- The integration process with SAP Business One.
- Connecting with shipping carriers.
- Difficulty of training staff and problems that occurred during operation.
- The operational performance of the platform.

achieving the final goal of the project **G5**. By analysing these elements, we aim to validate the platform's design objectives and its potential to streamline logistics operations.

8.1 Evaluation environments

To ensure a comprehensive evaluation of the platform, the development and testing processes were conducted across three distinct environments: Local, Staging, and Production. Each environment played a specific role in the development life cycle, enabling incremental validation of features, performance testing, and secure deployment.

8.1.1 Local development environment

The local environment primarily serves as the initial testing ground for development. In this environment, we can quickly implement and test new features without the risk of affecting the live system. Leveraged Docker to run isolated instances of databases identical in structure to the production environment. This setup helped ensure that all database interactions were fully tested

under controlled conditions, minimising irregularities between local and production behaviour. In order to invoke functions and mimic server responses without connecting to the actual cloud service, the `serverless-offline` plugin is used to simulate AWS Lambda and API Gateway locally.

8.1.2 Staging environment

The staging environment mimics the production environment as closely as possible and served as the final step before full-scale deployment. It is hosted on AWS to simulate real world conditions using the same IaC tools as in production, ensuring that all configurations were replicated. This includes using AWS CloudFormation for resource orchestration and AWS Lambda to run backend services. The staging environment is publicly accessible on a `staging` subdomain serving all services such as the dashboard, tracking page, and backend.

8.1.3 Production environment

The production environment is where the platform fully operates and is accessible to the end users. The platform is automatically deployed into the production environment always after successful deployment to the staging utilising close configurations.

By maintaining these distinct environments, we are able to systematically deploy updates, ensuring that each feature got tested before being released to the public. This structured approach not only minimises disruptions to live services, but also ensures that end users received a reliable and secure product.

8.2 Production evaluation areas

This section outlines the evaluation of the platform after its integration into a live business environment. The assessment focuses on four critical areas: integration with SAP Business One, connectivity with shipping carriers, training and operational challenges faced by staff, and the overall operational performance and business impact of the platform. The first step was to create a user account and set up a project. In this project, all operators were invited, the public API key was generated, as well as the setup of the so-called shipper address which serves as a return address for project shipments.

8.2.1 Integration with SAP Business One

Exchanging data with SAP Business One was a key element in the integration of the platform. It required periodic exports of data from SAP to the platform and updates back into SAP with tracking numbers, the latest status, and delivery confirmation flags. To accomplish this, we used the SAP Service Layer Proxy and Data Sender, as detailed in Chapter 7. Storing data in the status of the parcel `metadata` retrieved from the PPL shipping carrier, such as invoice numbers, shipment costs, tolls and fuel taxes, proved beneficial. These data facilitated and accelerated monthly, mostly manual, shipping invoice processing tasks where the data provided on the invoice need to be checked and validated. The integration

process comprised three main phases: Integration works with three phases of data exchange:

1. **Exporting New Shipments:** In this initial phase, all packaged shipments are exported to the platform. This process runs every five minutes during operator working hours and picks shipments marked by the operator in the SAP Business One.

2. **Updating New Shipments in SAP:** This phase updates all packed orders that have been exported to the platform and successfully dispatched to the carrier with a new status `DATA_SENT` and the tracking number.

3. **Periodic Status Update:** Each night, the Data Sender queries all previously shipped parcels, with a time limit specific to each carrier, and updates their status in SAP - ideally to "delivered."

This structured approach ensures frequent synchronisation between the data held in the platform and SAP Business One. Figure 8.1 below illustrates the daily number of shipments processed since integration, highlighting an average of approximately 100 shipments per day. The flat spots on the plot presents weekends and bank holidays.

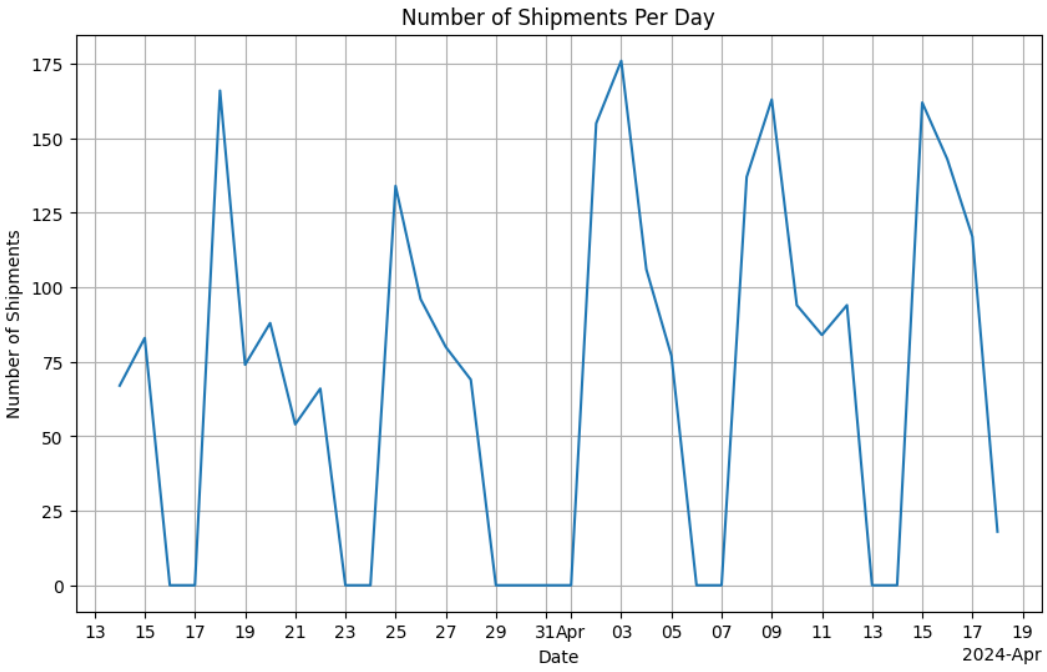


Figure 8.1: Number of shipments per day

Integration has been largely seamless. However, during the testing phase, we refined the database queries used to retrieve new shipments several times to meet previously unidentified requirements. Many of these adjustments stemmed from the need to handle shipments that did not have an associated invoice and only had reference to a packing list.

Furthermore, to provide insight into the workflow of operators and identify peak operational times, we analysed the distribution of shipments sent to the

platform within 30-minute intervals during a typical working day. As illustrated in Figure 8.2 below, the creation of shipments peaks around lunchtime, aligning closely with the pickup schedule of the shipping carrier between 13:30 and 14:30. With this data we can additionally optimise the scheduled tasks used for data exchange between SAP and our platform.

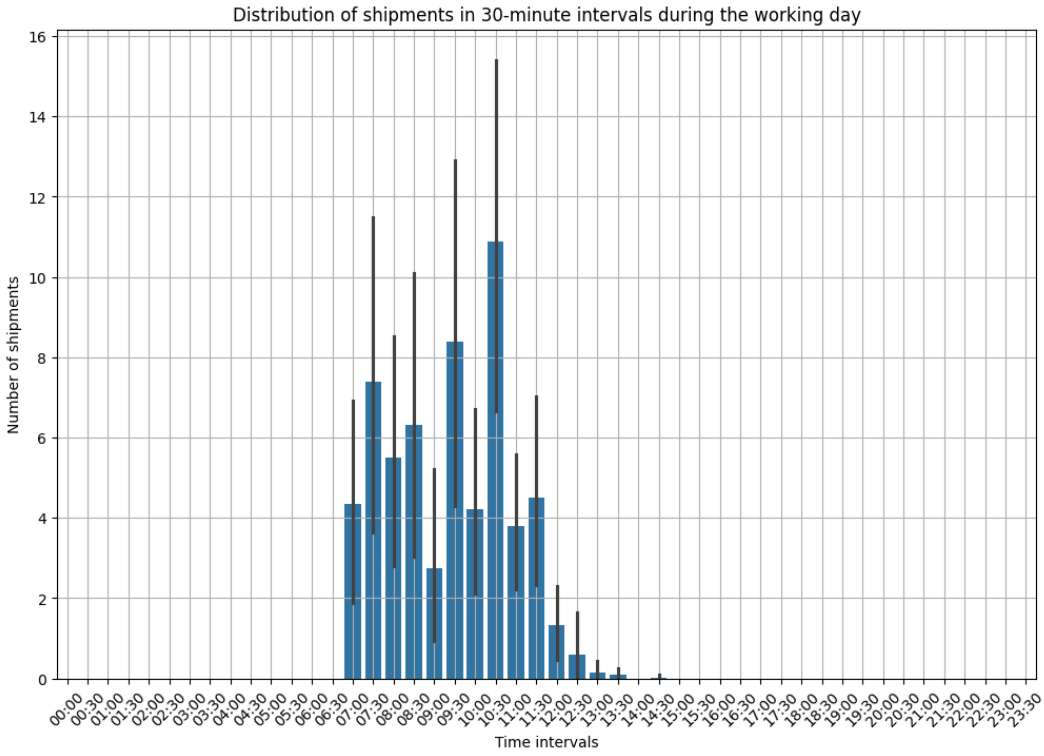


Figure 8.2: Distribution of shipments creation in 30-minutes intervals during the work day

8.2.2 Connecting with shipping carriers

Establishing a connection to all three implemented shipping carriers - Packeta, PPL, and Česká Pošta was a smooth process without any issues. Connecting packeta was straightforward; it involved simply copying the `API-Key` from Packeta's online administration interface. This password is all that was required to authenticate and interact with Packeta's API, making the integration process very simple and fast. For PPL, the credentials needed were a `ClientID` and a `Client Secret`, which had to be obtained directly from PPL's support team. Although this required waiting for the support team to provide the necessary credentials, the overall process went smoothly once the credentials were received. Integration with Česká Pošta was slightly more involved. Initially, it required contacting the sales representative of Česká Pošta to ensure that the proper permissions to generate access keys were established in the Česká Pošta client administration portal. After these permissions were in place, generating the `API Token` and obtaining `Secret` was straightforward. However, additional details such as the postal code of the post office, the customer's ID, and the contract number needed to be specified; these were promptly provided by the sales representative.

With all necessary credentials and configurations set, each carrier's integration was completed and saved under the respective project in platform. Integration of these carriers represents a critical step in the platform, allowing data transmission between the platform and the shipping carrier.

8.2.3 Training and operational challenges

Training the staff, so-called operators, to use the platform was relatively straightforward, thanks to the simplicity principle of the platform design. The primary interface feature is a main table where operators select rows and execute predefined actions with a single button click. This simplicity in design minimized the learning curve and helped quick adoption. Given the fact, that operators use desktop computers with a mouse and keyboard, there were no issues with layout responsibility or accessibility of the dashboard. Platform's data filtering and manipulation functionalities were also intuitive for the staff. Most operators already had basic knowledge of software like Excel and were familiar with the SAP Business One user interface, making adoption easier. Despite the ease of training, there were operational challenges during the initial phases of the deployment. Adjustments had to made within the shipments table, such as highlighting "today's" orders, setting a row colour for different shipment statuses. Given that the Packeta API is not among the fastest, which was mostly shown when generating package labels, the implementation of a simple loading bar was necessary. This loading bar was displayed to the user immediately after clicking the button, completely disabling it. Several changes were also made to the shipment detail user interface, where some fields were rearranged and added, as well as fixing a bug that made it impossible to edit the pickup point.

From the user perspective, the branded tracking page presents the status of the parcels and basic shipment data. However, users also have the option to go to the official shipping carrier tracking page. But the question is do they use it? We have implemented anonymous event-based user tracking.

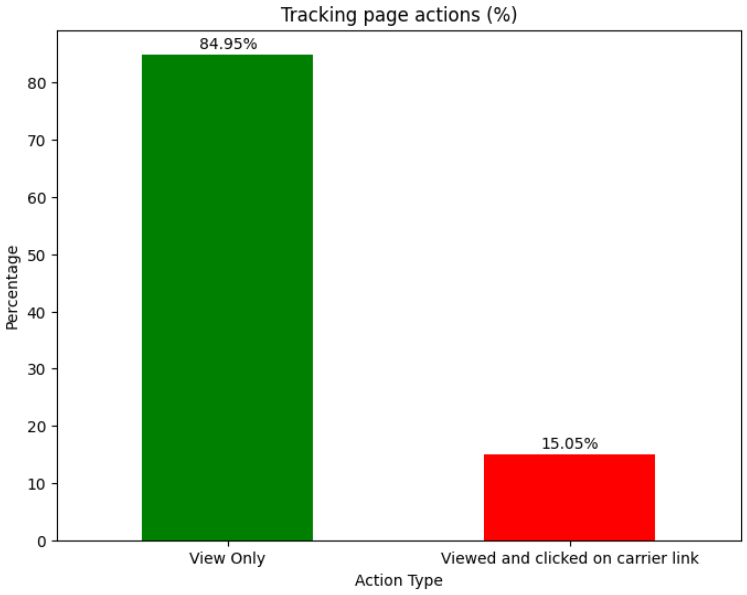


Figure 8.3: Tracking page actions

And, as we can see from the data on Figure 8.3, the significant portion of users is satisfied with what they see on our tracking page and do not need to continue to the carrier's official tracking page at all. Another interesting thing that arises from tracking page events data is the distribution of device, or to be more precise, screen type. We have decided to track three device types:

- `desktop`: Everything over approximately 992 px.
- `table`: Everything over approximately 768 px.
- `mobile`: The rest.

Although we expected that most of the users will open the tracking page from their phone, the data in Figure 8.4 show something different. Phone users are certainly not insignificant, but the desktop leads the way.

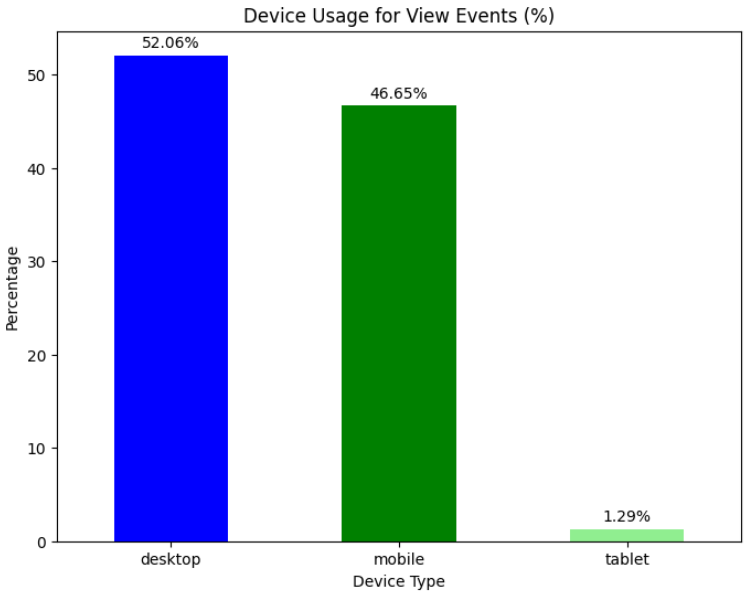


Figure 8.4: Device usage for view events (%)

8.2.4 Operational performance and business impact

The operational performance of the platform has been robust, supported by the detailed AWS CloudWatch monitoring. Analysis of the Lambda invocation duration in Figure 8.5 reveals that the average response time during peak periods can reach up to 4 seconds on average, which is still within the Lambda tolerance. So, just raising the timeout should be enough for these special cases described below. However, normally, this number is well below 250 miliseconds.

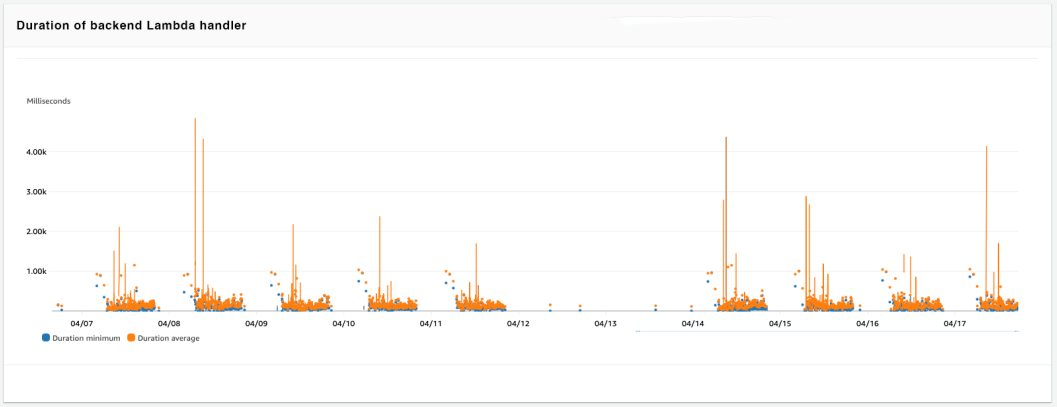


Figure 8.5: Duration of backend Lambda handler

The high average is typically associated with requests for shipping labels or operations involving the Packeta API, which tends to have slower response times due to the need to await responses with the Lambda function. Furthermore, the error and success rates monitored through AWS CloudWatch in Figure 8.6 indicate a very high availability of the system. The metrics show a minimal error rate, which underscores the robustness and reliability of the backend Lambda handler. This high success rate ensures that the system remains dependable under various operational conditions, providing a stable and efficient service to users.

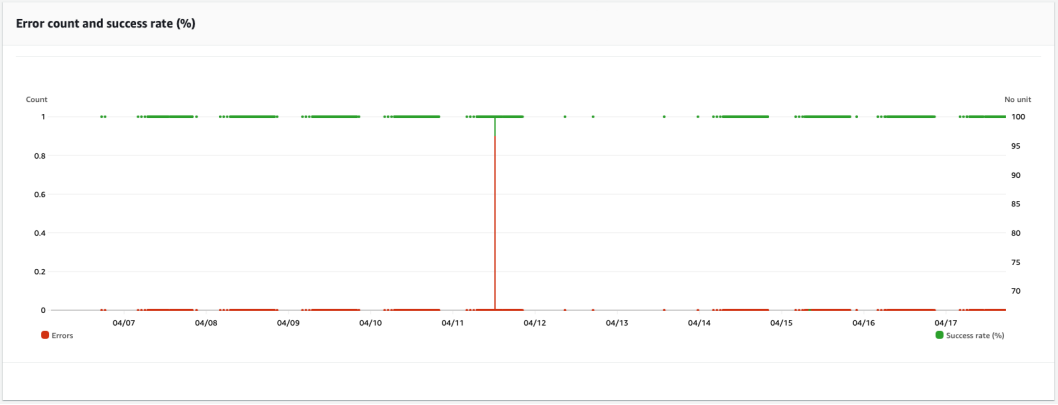


Figure 8.6: Error count and success rate (%) of backend Lambda handler

The business impact of the platform has been largely positive. Transitioning from an older, difficult to maintain system to this modern platform, has moved the company's logistic operations into a good direction. The previous system, while functional, suffered from poor architecture and limited accessibility, restricting usage to only a network within the company. Not only that, but operators could not anyhow edit the data in the old system. Meaning that they had to do all minor changes in the SAP Business One, making the whole job much more difficult. In contrast, the new platform offers flexibility and remote accessibility, allowing staff to interact with the system from any location using mobile devices.

This improvement in accessibility and user experience is complemented by the extensibility of the platform. The architecture of the new system is designed to facilitate integration with different carriers and allow for very straightforward

integrations with new carriers and updates within existing APIs. This capability is particularly valuable in today's dynamic business environment, where shipping conditions and costs can change frequently, making it unsuitable for the business. The platform design allows the business to quickly adapt to these changes by enabling a seamless transition to different carriers as needed, as long as the carrier implementation is present. Or, the business can request the implementation of a new carrier, which generally should be a complex problem given the platform carrier integration design. This adaptability not only provides operational flexibility, but also gives the company a competitive advantage in logistics management and enhances its ability to respond effectively to market changes and customer needs.

8.2.5 Achievement of project goals

At the beginning of the development of this platform, we set five main goals, as outlined in Project goals. These objectives were aiming to improve the expedition process of the eCommerce companies by automating interactions with shipping carriers, enhancing customer engagement through branded tracking page, and ensuring simple integration with existing systems. Here, we evaluate and reflect on how these goals were fulfilled through the deployment and real-world application of the platform.

G1: Streamline logistics operations: The platform has effectively simplified the process of dispatching orders to shipping carriers by automating data exchanges and minimizing manual intervention. This was achieved through a user-friendly dashboard that facilitates all data sending processes and label generation, thus streamlining logistics operations. Refer to Figure 8.7 for a view of the dashboard shipment list with filtering, Figure 8.8 for the shipment edit interface and Figure 8.9 for the detailed shipment view. Please note that the data shown in the examples are mocked, and not actual customer data.

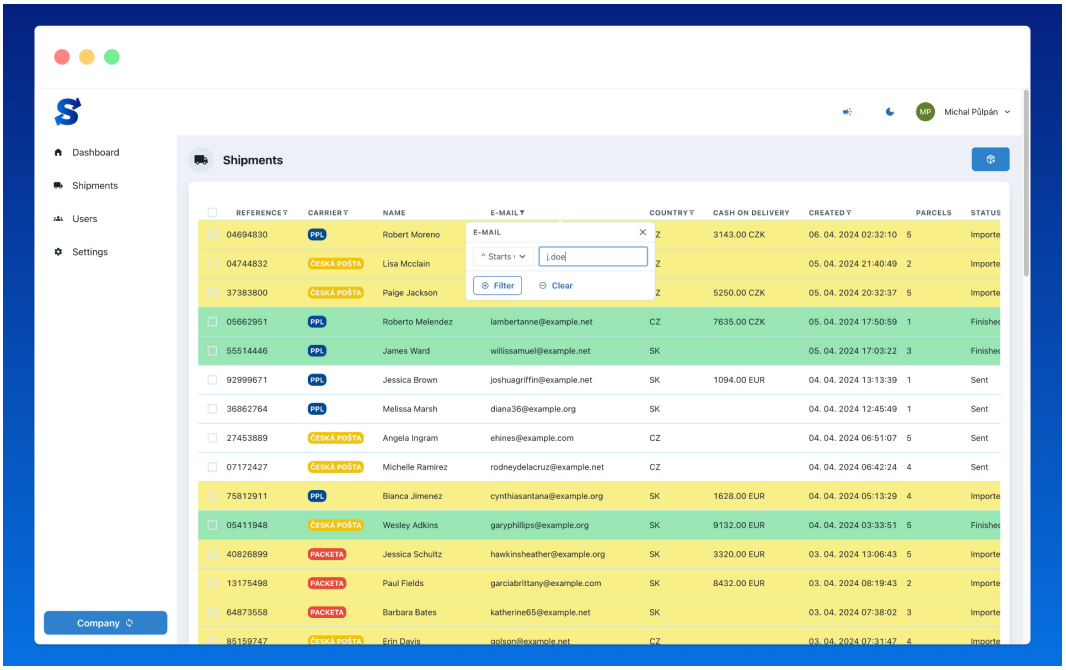


Figure 8.7: Dashboard shipment list with filtering

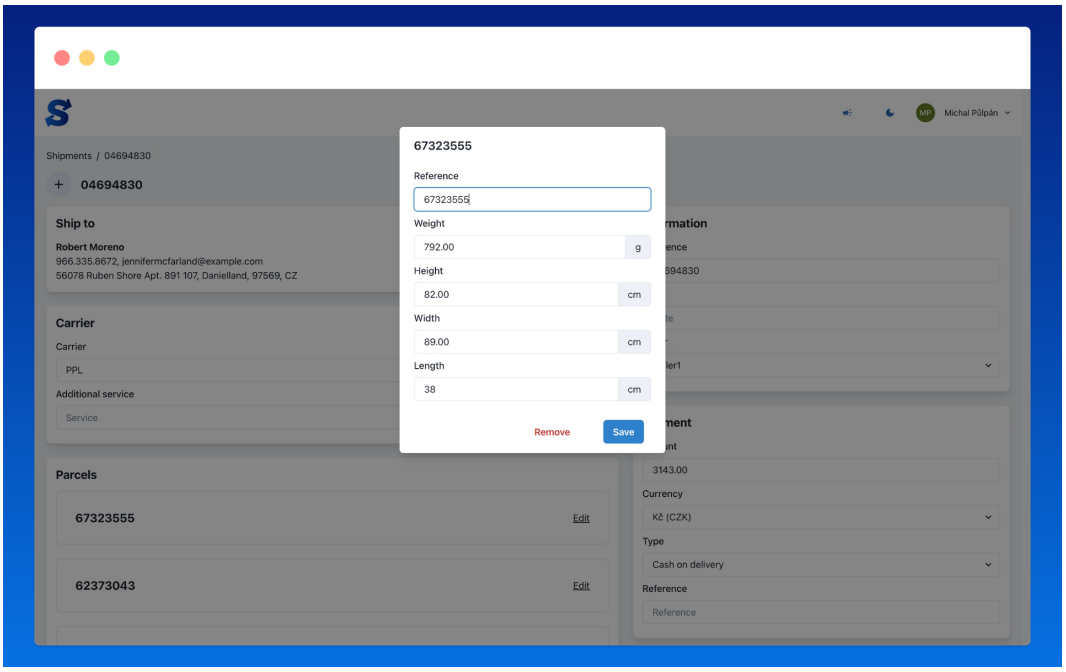


Figure 8.8: Dashboard shipment edit mode

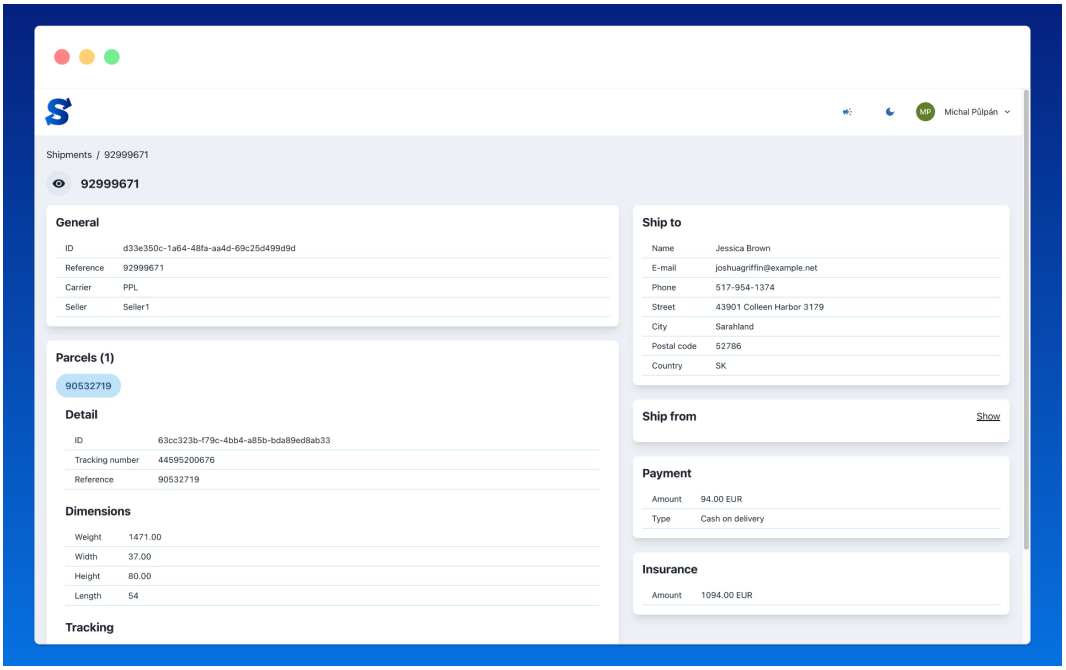


Figure 8.9: Dashboard shipment view mode (after being sent)

G2: Modern cloud based multi-tenant solution: Developed with a multi-tenant architecture, the platform supports multiple companies simultaneously. The platform uses Amazon Web Services as a deployment infrastructure. This structure with the use of a serverless deployment ensures that the platform can easily adapt to growing business needs while maintaining performance and data security.

G3: Create branded shipping customer experience: The platform enhances the post-purchase experience by allowing businesses to customise the branding of parcel tracking pages and email notifications. Customisation is done through an intuitive configuration interface within the dashboard. This covers standard operational processes into valuable marketing opportunities, creating a new marketing channel while increasing customer engagement and strengthening brand identity. Figure 8.10 shows the configuration page where businesses can set the branding for their tracking pages and email notifications. Figure 8.11 shows an example of an email notification for a package sent to an Austrian customer, illustrating how multiple branding layouts can be managed in a single account.

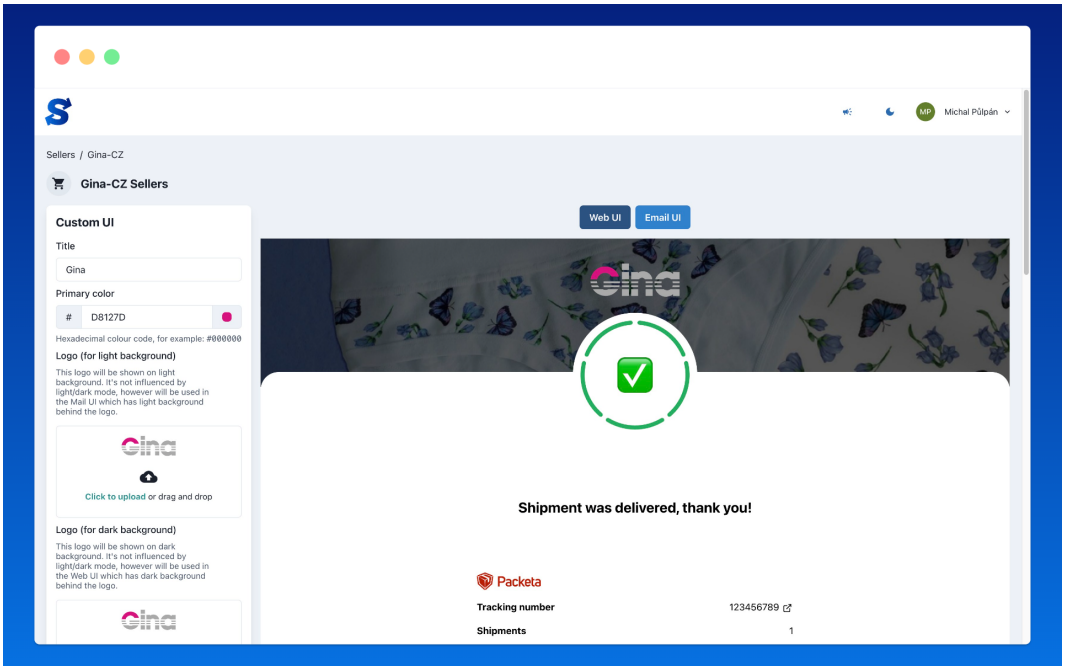


Figure 8.10: Dashboard tracking page/email notification branding configuration page

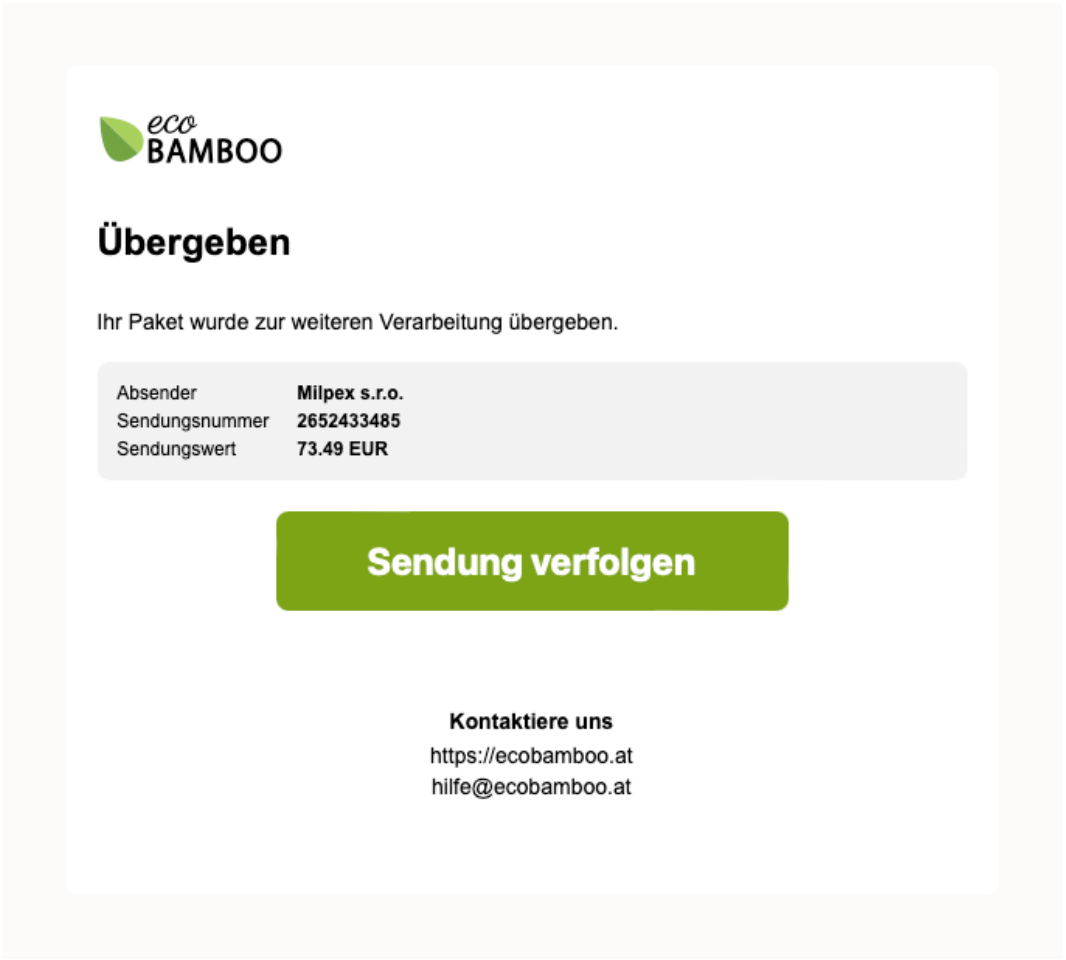


Figure 8.11: Email notification of shipped parcel for Austrian customer

G4: Integration with existing systems: Seamless integration with existing business systems, particularly SAP Business One, was an important aspect of the platform. The platform facilitates this through public API that offers the same services provided by the dashboard, including creating and modifying shipments, generating labels, retrieving filtered shipments, and much more. This allows for integration flexibility and the ability to automate processes externally from the platform interface. Figure 8.12 shows the online user documentation interface that helps users navigate and utilise the platform features effectively. Figure 8.13 displays the Swagger public API documentation, which is instrumental for developers looking to seamlessly integrate their systems with the platform.

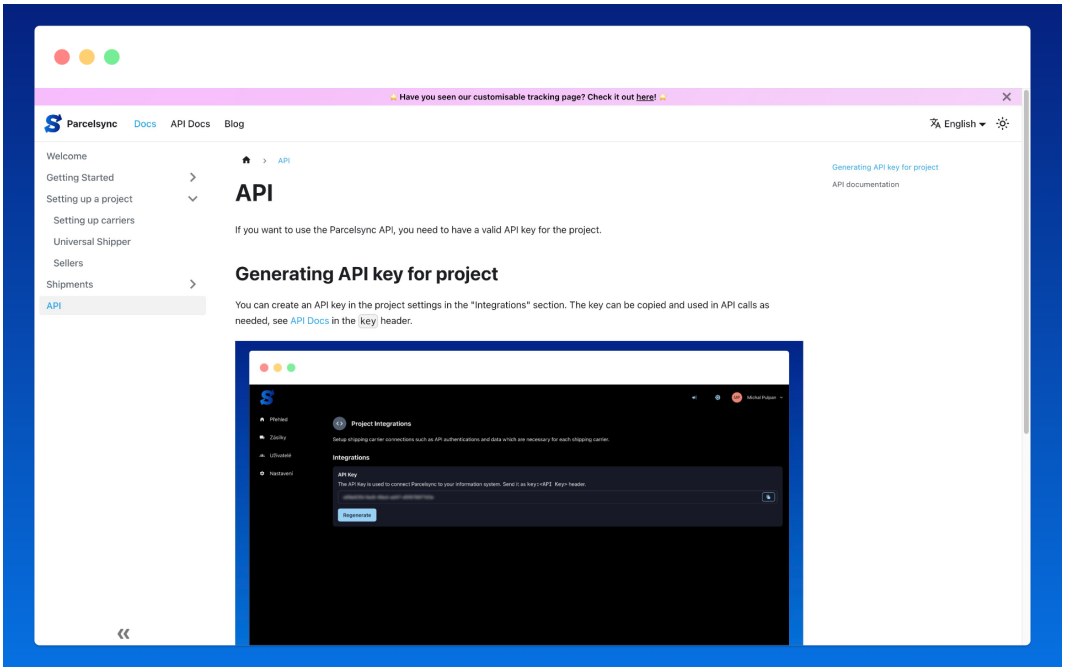


Figure 8.12: Online user documentation interface

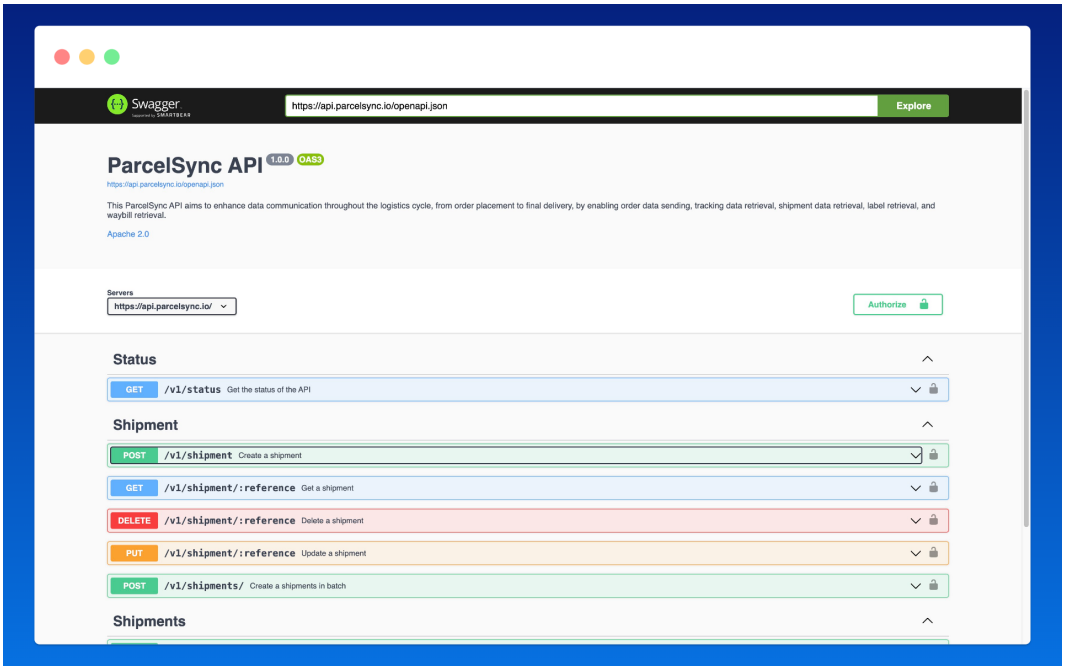


Figure 8.13: Swagger public API documentation

G5: Validate in a real-world setting with SAP Business One integration: The effectiveness of the platform has been thoroughly tested in a real eCommerce environment, which manages more than 100 shipments per day. This real-world testing, integrated with SAP Business One, demonstrated the robustness and operational reliability of the platform. Figure 8.14 showing the spatial distribution of the recipients of the shipment during the first month of use of the production.

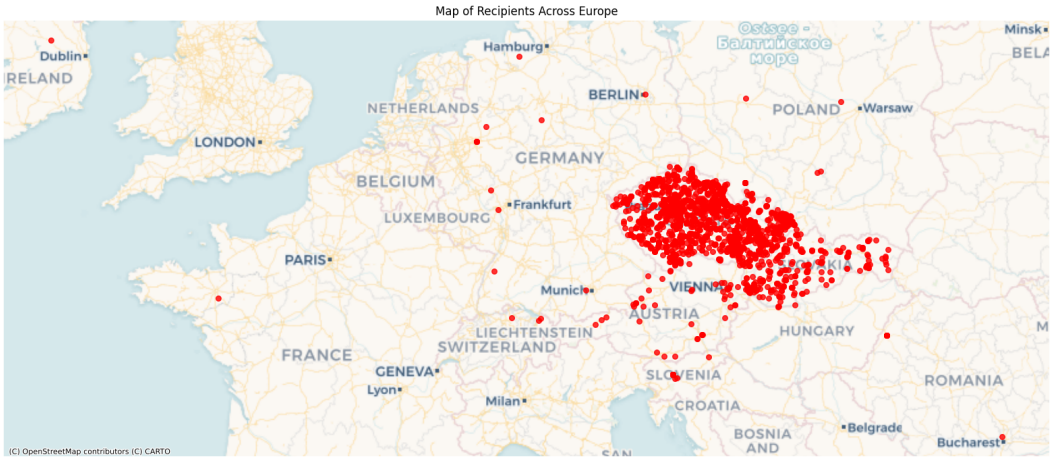


Figure 8.14: Distribution of recipients within Europe

This chapter has evaluated the operational efficacy and integration of the SaaS platform within a real-world business environment. During this evaluation, the platform has been shown to meet operational requirements by simplifying logistics processes, enhancing customer interaction through branded experiences, and integrating with existing business systems. By achieving these objectives, the

platform not only fulfils the set project goals but also establishes a foundation for future expansions and optimisations in eCommerce logistics. The platform provides a clear direction for continued innovation and improvement, ensuring that the platform can continue to deliver business value as market conditions evolve.

Conclusion

In this thesis, we explored the complexities of communication between shipping carriers and e-Commerce companies. Our initial research into existing solutions revealed a gap: Most platforms focus primarily on the operational aspects of expedition logistics without integrating any interface for the customer to whom the collected data about their parcel could be presented. This provided an opportunity to enhance the post-purchase experience through a customizable tracking page and email notification, serving as a possible new marketing channel.

The primary goal of this thesis was to develop a multi-tenant SaaS application that not only handles communication with shipping carriers, but also enhances the customer's engagement with the seller after the purchase. At the same time, we integrated the platform with SAP Business One through a communication proxy we created, which allowed us to exchange data. This integration was pivotal, as it allowed us to test the platform in a real warehouse environment, validating its functionality in a real-world setting.

The deployment of the platform on AWS uses modern standards of IaC and maintaining a CI/CD processes, highlights its readiness for industry use. By using AWS services, the platform benefits from availability, security, and performance, which are critical for handling sensitive business operations.

Evaluation

The platform has successfully achieved all the goals set out at the beginning of this thesis. The creation of a SaaS platform designed to streamline the communication between e-Commerce businesses and shipping carriers has proven to be a valuable asset in daily operations for both operators and customers.

The platform has met all functional and non-functional requirements identified during the analysis phase. We have implemented integrations with three major shipping carriers in the Czech republic - Česká Pošta, Packeta, and PPL. Furthermore, the platform is designed for easy integration with additional carriers, ensuring that the integration process is as seamless as possible and abstracts the complexities from the operators. To facilitate the integration of the platform within a company environment, two additional software solutions were developed. The main is to secure a proxy for SAP Business One Service Layer, which ensures the safe and reliable handling of requests to the ERP system. The second software handles the scheduling of synchronization tasks that enables continuous data exchange between the platform and SAP Business One. Despite the fact that it is out of the scope of this thesis, we will mention that the SAP Proxy API was used to connect one of the e-Commerce stores of the company in which the integration took place.

As of the date of this submission, the platform has successfully processed logistics data for more than 3,500 shipments and served more than 3,000 customers since its integration in mid-March 2024. This not only shows the success of the platform but also demonstrates the seamless integration and ability used by the operators. The successful deployment, integration, and functionality of the platform reflects its robustness and readiness to meet the evolving needs of

the e-Commerce sector, opening the doors for future enhancements and wider adoption.

Future work

Although the platform has successfully met its original objectives, it has potential for expansion and further development. Future enhancements can expand its usability and increase its value to a wider range of companies and their use cases.

Carriers

Expanding the range of integrated shipping carriers is a key focus for the future development of the platform. By integrating additional carries such as DPD, GLS, and others that operate within the Czech market, the platform will be able to meet more logistical needs and requirements. It is also important to keep in mind that the Czech market is not the only target market. Unfortunately, in this domain, often the carriers in question have a bias towards a given country, and therefore, for example, the API communication with Czech PPL owned by a German DHL is different.

Ready-made integrations with external systems

Ready-made integrations will simplify the adoption process for companies, making it simpler and more efficient. These integrations should involve automatic data transfers between ERP systems or e-Commerce platforms, thus reducing the barriers for potential users.

Domestic carriers for international shipments

The platform should improve its support for international shipments managed by domestic carriers that partner with foreign carriers. Often, a shipment sent through a domestic shipping carrier like Česká Pošta or PPL is actually delivered by a partner like DHL in a foreign country. The sellers usually also try to display the branding of delivery carriers in the checkout process on their e-Commerce store to make the entire shopping process more trustworthy and local. The platform needs to recognise and adapt to these co-operations by displaying the actual carrier's logo and brand information to the recipient. This will ensure transparency and maintain consistency in customer communication for the seller.

Pick-up point details

Providing detailed information about the pick-up points could improve the customer experience. If a shipment is sent to the pick-up, the platform could provide detailed information, location, and operating hours. Such features would make the tracking page more informative and valuable for recipients.

Customer's parcels overview

The platform has the potential to become a central hub for customers to view all their incoming and historical parcels. This feature would allow users to manage and track all their shipments in one place, regardless of the carrier or the sender. As the platform gains wider adoption, this functionality could become a valuable tool for recipients, enhancing their overall experience and interaction with the platform.

Glossary

`AWS Lambda Step Functions` serverless orchestration service - Step functions documentations. 109

`Joi` JavaScript schema description language and validation library - Official page. 100

`Joi` Create and modify PDF documents in any JavaScript environment - Official page. 103

`Puppeteer` Node.js library which provides a high-level API to control Chrome/Chromium - GitHub repository. 44

`jsPDF` Client-side JavaScript PDF generation library - GitHub project repository. 44

`mssql` Microsoft SQL Server client for Node.js- Project GitHub. 60

`pdfjs` A PDF generation library targeting both the server- and client-side - GitHub project repository. 44, 103

`react-router-dom` Library for client-routing in React - Official page. 45

`serverless-offline` Serverless plugin emulating AWS Lambda and API Gateway locally - Serverless Offline documentations. 67

Amazon Cognito Service helps implement customer identity and access management (CIAM) into web and mobile applications - Official page. 43

AWS Certificate Manager Service to provision, manage, and deploy public and private SSL/TLS certificates within AWS - Official page. 46, 48

AWS Cloud Development Kit (CDK) software development framework used to model and provision cloud application resources - Official page. 47, 49, 51, 52

AWS Cloud Front Is a content delivery network (CDN) service built for high performance, security, and developer convenience - Official page. 46

AWS CloudFormation IaC service - Official page. 2, 46, 49, 52, 67

AWS CloudWatch Monitoring tool of AWS services - Official page. 71, 72

AWS Command Line Interface (CLI) Unified tool to manage AWS services directly from the command line - Official page. 51

AWS Elastic Container Service (ECS) Managed container orchestration service - Official page. 50

AWS Identity and Access Management (IAM) Permission and role management for AWS resources - Official page. 49, 52, 123

AWS Lambda Amazon serverless solution - Official page. 2, 46–48, 50, 51, 67

AWS Relational Database Service (RDS) A relational database service - Official page. 48

AWS Route 53 Scalable and highly available Domain Name System (DNS) service - Official page. 46, 48

AWS S3 Amazon S3 - Official page. 2, 46, 48, 52

AWS SDK Simplifies use of AWS services with higher level of abstraction in JavaScript - Official page. 108

AWS Simple Email Service Amazon SES is a cloud-based email service that can integrate into any application for high volume email automation - Official page. 44, 46, 108

AWS Virtual Private Cloud (VPC) A service to logically isolate AWS resources - Official page. 48

Axios Promise based HTTP client for the browser and node.js - Axios documentation. 116

Axios Library for building interactive command line tools in JavaScript - Official page. 117

AxiosInstance Reusable instance of axios configuration - Axios documentation of AxiosInstance. 61, 116

Azure Functions Microsoft serverless solution - Official page. 51

Azure Kubernetes Service Official page - Azure Kubernetes Service. 51

Certbot Tool for Let's Encrypt certificate management - Official page. 63

Chromium Chromium is an open-source browser project - Official page. 44

Django High-level Python web framework - Django official page. 33

Docker Platform to build, share and run container applications - Official page. 50, 62

Docker Hub A container registry for Docker images - Official page. 63

Dockerfile A text-based file with no file extension that contains a script of instructions for the container - Docker documentation (Build the app's image). 62

Doctosaurus An optimized site generator in React - Official page. 109

esbuild JavaScript bundler for the web - Official page. 62

ESLint Linting utility for JavasScript and TypeScript. More details can be found in ESLint official page. 21, 51

Express Web framework for Node.js. Express official page. 35

Flask Micro web framework for Python - Flask official page. 33

GitHub Actions Tool for task automation by GitHub - Official page. 51, 52

Google Cloud Platform Official page - Google Cloud. 2, 50

Google Kubernetes Engine Kubernetes service - Google Kubernetes Engine (GKE) official page. 50, 51

Knex.js SQL query builder for PostgreSQL, CockroachDB, MSSQL, MySQL, MariaDB, SQLite3, Better-SQLite3, Oracle, and Amazon Redshift designed to be flexible, portable, and fun to use. Knex.js official page. 37, 105, 112

Koa Web framework for Node.js. Koa official page. 2, 31, 35–37, 47, 59, 60

Microsoft Azure Official page - Microsoft Azure. 2, 51

Nginx HTTP and reverse proxy server - Official page. 59, 63

Node.js JavaScript runtime environment Node.js official page. 35, 37

Objection.js An SQL-friendly ORM for Node.js - Objection.js. 37, 43, 59, 60, 112

PostgreSQL Open source object-relational database system PostgreSQL official page. 31, 37, 59

project An entity within the platform meant to group data together.. 19, 20, 40

React JavaScript frontend library React official page. 2, 31, 34, 35, 46

SAP Business One ERP software - Official page. 2, 17, 18, 54–59, 63, 64, 88

SAP Business One Data Interface API (DI API) SAP Business One API for consuming SAP Business One data- Working with SAP Business DI API. 2, 54, 55

SAP Business One Service Layer SAP Business One API - Working with SAP Business One Service Layer. 2, 3, 55–57, 59, 61, 80

seller An entity within the platform associated to the project defining the communication layout - branding, types of notifications and general information.. 19, 20

Serverless framework Framework developed for building applications on AWS Lambda - Serverless framework official page. 37, 47, 49, 52

Svelte JavaScript web framework - Svelte. 35

Vercel Platform as a service company - Vercel official page. 50

Versino SAP Business One supplier and integrator - Official page. 55

Vue.js JavaScript web framework for developing SPA - Angular.js. 35

Vue.js JavaScript framework for building user interfaces - Vue.js. 35

Watchtower Tool for automating Docker container base image updates - Official page. 63

Yarn Node.js package manager - Official page. 59

Bibliography

1. CZECH ASSOCIATION FOR ELECTRONIC COMMERCE (ed.). *E-commerce Study 2023 - Hlavní textová část* [online]. [visited on 2024-03-24]. Available from: https://www.apek.cz/archiv-dokumentu?filters_document%5Bcategories%5D%5B0%5D=1.
2. OGUNMOLA, Gabriel; KUMAR, Vikas. E-Commerce Research Models: A systematic review and Identification of the Determinants to Success. *International Journal of Business Information Systems*. 2023, vol. 43, p. 2023. Available from DOI: 10.1504/IJBIS.2020.10044532.
3. SOMMERVILLE, Ian. *Software Engineering*. 9th ed. Addison-Wesley, 2010.
4. YONKEU, Steve. *Understanding event driven architecture* [https://dev.to/yokwejuste/understanding-event-driven-architecture-110o]. 2024. [visited on 2024-04-20].
5. YONKEU, Steve. *Top Best Software Architecture Pattern TO Choose* [www.medium.com/@darshanaslp/top-best-software-architecture-pattern-to-choose-d44c20b37652]. 2021. [visited on 2024-04-20].
6. STACK OVERFLOW. *Stack Overflow Developer Survey 2022* [online]. 2022. [visited on 2024-04-03]. Available from: https://survey.stackoverflow.co/2022/#most-popular-technologies-language-prof.
7. STACK OVERFLOW. *Stack Overflow Developer Survey 2023* [online]. 2023. [visited on 2024-04-03]. Available from: https://survey.stackoverflow.co/2023/#most-popular-technologies-language-prof.
8. WORTHAM, Steve (ed.). *Performance Benchmarking: Bun vs. C# vs. Go vs. Node.js vs. Python* [online]. [N.d.]. [visited on 2024-04-03]. Available from: https://www.wwt.com/blog/performance-benchmarking-bun-vs-c-vs-go-vs-nodejs-vs-python.
9. REACT. *React Documentation* [online]. [N.d.]. [visited on 2024-04-04]. Available from: https://react.dev/learn/.
10. REACT. *Writing Markup with JSX* [online]. [N.d.]. [visited on 2024-04-04]. Available from: https://react.dev/learn/writing-markup-with-jsx.
11. VUE.JS. *Vue.js Documentation* [online]. [N.d.]. [visited on 2024-04-04]. Available from: https://vuejs.org/guide.
12. KOA. *Koa Documentation* [online]. [N.d.]. [visited on 2024-04-04]. Available from: https://koajs.com.
13. KREBS, Rouven; MOMM, Christof; KOUNEV, Samuel. Architectural Concerns in Multi-Tenant SaaS Applications. In: 2012.
14. CODECRAFT. *Multi-Tenancy In Software Architecture [Updated-2023]* [https://medium.com/codex/multi-tenancy-1c31c181cc41]. 2023. [visited on 2024-04-05].

List of Figures

List of Abbreviations

APEK Czech Association for Electronic Commerce
ERP Enterprise Resource Planning
PPC Pay-per-click
CRUD Create, Read, Update, Delete
ORM Object–relational mapping
SaaS Software as a Service
AWS Amazon Web Services
CLI Command-Line Interface
CRA Create React App
DOM Document Object Model
MVC Model View Controller pattern
DBMS Database Management System
UI User Interface
CI Continuous Integration
CD Continuous Deployment
MS Microsoft
B2B Business to Business
B2C Business to Customer
SOAP Simple Object Access Protocol
REST Representational state transfer
IaC Infrastructure as Code
RDS Relational Database Service
SES Simple Email Service
SPA Single Page Application
JWT JSON Web Token
SSL Secure Sockets Layer
TLS Transport Layer Security
DNS Domain Name Server
S3 Simple Storage Service
CDN Content Delivery Network
VPS Virtual Private Server
GKE Google Kubernetes Engine
monorepo Monolithic repository
multi-repo multiple repositories
DRY Don't repeat yourself

A. Programming Documentation - Platform

This document serves as a programming documentation for the platform. Covers the mono-repo setup that houses both the client-side React applications and the server-side API services. This document is intended to guide developers through system setup, project structure, feature integration, and daily tasks.

The first Section A.1 will present the complete structure of the platform, including the description of the components. Then, in the Section A.2 we will go through and describe the coding conventions that are recommended to follow when working on the platform. The third A.3 Section will provide brief overview of the technical design of the platform. Next, the fourth Section A.4 will introduce the backend of the platform, including segmentation of project parts and technical details such as authorization, authentication, data filtering and general communication with shipping carriers. In the fifth Section A.5 we will present frontends with several decision and key elements influencing the project. Next, in the Section A.6, we will delve into a way how to integrate new features, such as shipping carriers, but also essential things such as adding new environment variable. In the Section A.7 several important infrastructure points from programmers perspective will be presented. And lastly, in the Section A.8, some information regarding user documentation will be provided.

A.1 Project structure

The platform uses a monorepo architecture to house both frontend and backend components under a single repository. This approach simplifies dependency management, and enhances re-usability across the codebase. The monorepo includes:

- **Clients:** Containing all React frontend applications and dashboard user documentation.
- **Services:** Housing the API backend service.
- **Infrastructure:** Definitions and configurations of the deployment and operational infrastructure.

A.1.0.1 Clients

The clients section currently contains all three frontend projects:

```
clients
├── docs/
├── tracking/
└── web/
```

Here, the `docs` project is a Doctosaurus user documentation. The `tracking` and `web` are React applications, first for parcel recipients and second is dashboard for the actual users of the platform.

A.1.0.2 Services

The Services folder contains only one service, `api` - Koa backend.

```
services
└── api/
```

A.1.0.3 Infrastructure

The Infrastructure folder contains all the necessary setups for the deployment in AWS.

```
infrastructure
├── apps/
├── constructs/
└── stacks/
```

The `apps` folder contains the deployment entry point file. It is where all `stacks` are initialised for the requested environment. In the `constructs` we can find several AWS Constructs. These are classes inherited from `Construct` from package `constructs` defining so-called "piece of system state". In our case, the constructs are used to define the database, the gateways, the policies, and the S3 bucket for the user uploaded assets. And lastly, `stacks` is where the actual services are defined. Using data from `aws-cdk-lib.Stack`, they define actual CloudFormation stacks. Having said that, there is a stack for api, dashboard (webapp), tracking page, documentation, and a specific stack for certificates.

A.1.1 Package management

The platform uses Yarn as its package manager. Yarn Workspaces are utilised to link together frontend and backend packages, allowing share dependencies installed at the root level optimising disk usage and installation process. The dependencies for individual projects within the monorepo are declared in their `package.json` files. The root `package.json` includes scripts and dependencies applied throughout the repository.

A.2 Coding convention

A.2.1 Style guide

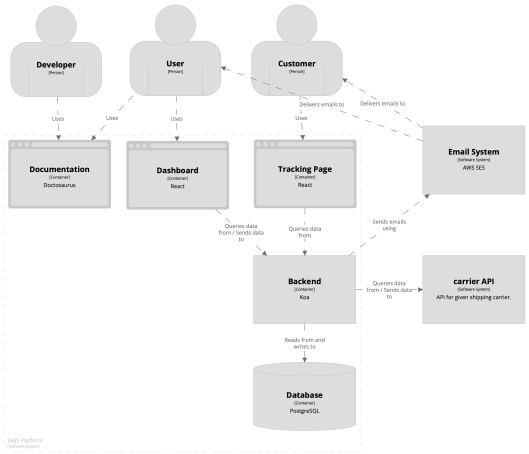
It is recommended to adhere to the Airbnb JavaScript and React style guide for both backend and frontend code. Including consistent use of modern JavaScript syntax and readable formatting. Please use linter regularly, each workspace within the repository contains `eslint` which can be run by `yarn check:lint`.

A.2.2 File naming

It is recommended to use `kebaba-case` for file names. However, in the React frontend codebase, `PascalCase` is usually used except for the layout components.

A.3 Technical design

The platform overview can be seen on Figure A.1.



[Container] SaaS Platform

Figure A.1: C4 overview of the architecture

A.4 Backend

The backend service serves as the central centre for processing all logistic operations, interfacing with external carriers, and managing user interactions through the API. Developed using Node.js and the Koa framework, it provides scalable server-side logic. The project is in the `services/api/` directory. The platform is built around several key components in the following order:

1. **Router:** Is main entry of the API where all endpoints are defined.
2. **Middlewares:** Are chain-able methods initiated at the beginning of endpoint request or at the end to modify response. They handle authorization, authentication, schema validation, parsing pagination, or filter requests.
3. **Actions:** Actions present the ”called” method of the endpoint after request middleware.
4. **Entities:** Objects stored in the connected database.
5. **Services:** Manage the database interactions.
6. **Modules:** Defined carrier integrations with unified interface.

The backend elevates the dependency injection container using Microsoft library `tsyringe`.

A.4.1 Database connection

The database connection reference is stored as a singleton variable at the Lambda handler entry-point (`src/lambda-handlers/api.ts`). This should ensure that no more connections than one are initiated, hence minimising the database load and user connections.

A.4.2 Database schema

Due to the complexity of the database schema, it will be described in following sections:

- Projects
- Users
- Shipments

A.4.2.1 Projects

Projects are the main building block of the platform. They are used as the point of differentiation of data between tenants. However, each tenant can have access to or own multiple projects (based on their plan). For the database diagram, refer to Figure A.2. The project includes a reference to its owner, as well as a name and renewable API key. Multiple users can have access to the project and users can be invited into the project. Each project can contain individual settings of the shipping carrier API (`project_shippingcarrier`) and multiple sellers (`project_sellers`). Furthermore, the project contains its global shipper (`shipper`) reference meant for each shipment in the table.

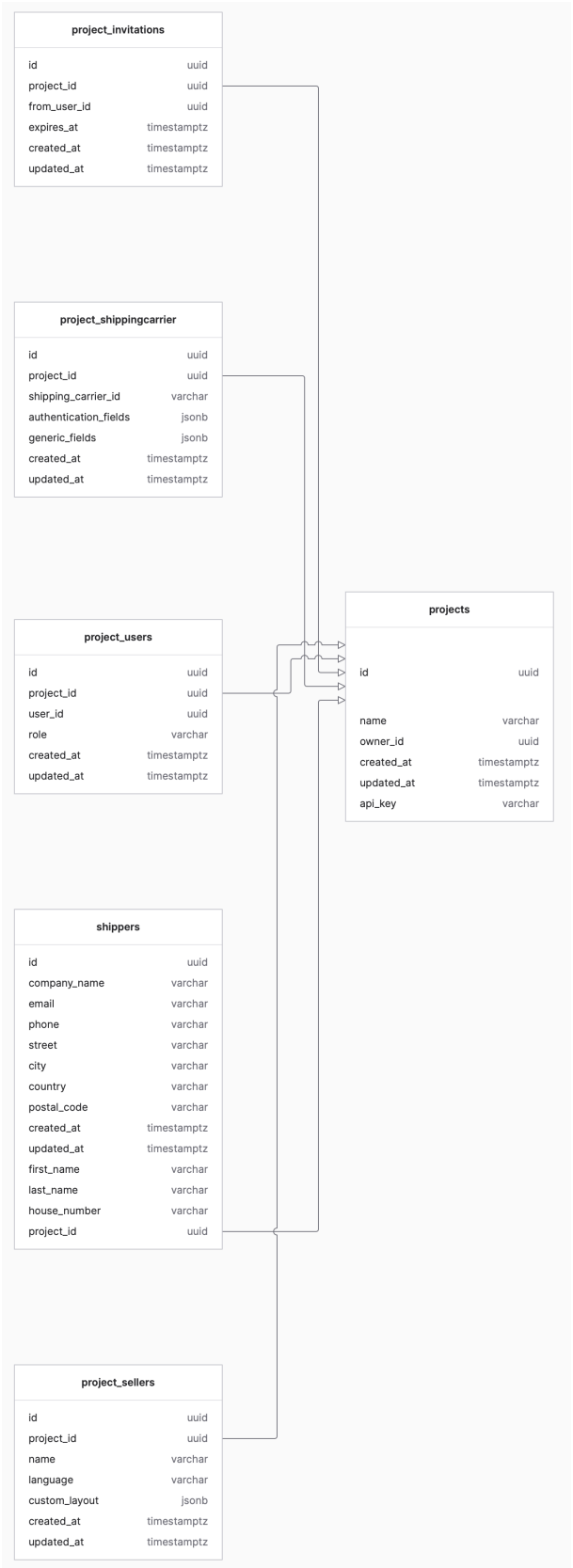


Figure A.2: Database schema diagram of `Projects` related tables

A.4.2.2 Users

The user is an object authenticated and authorised with each request. For the database diagram, please refer to Figure A.3. If a user owns a project, they have to have a plan defined.

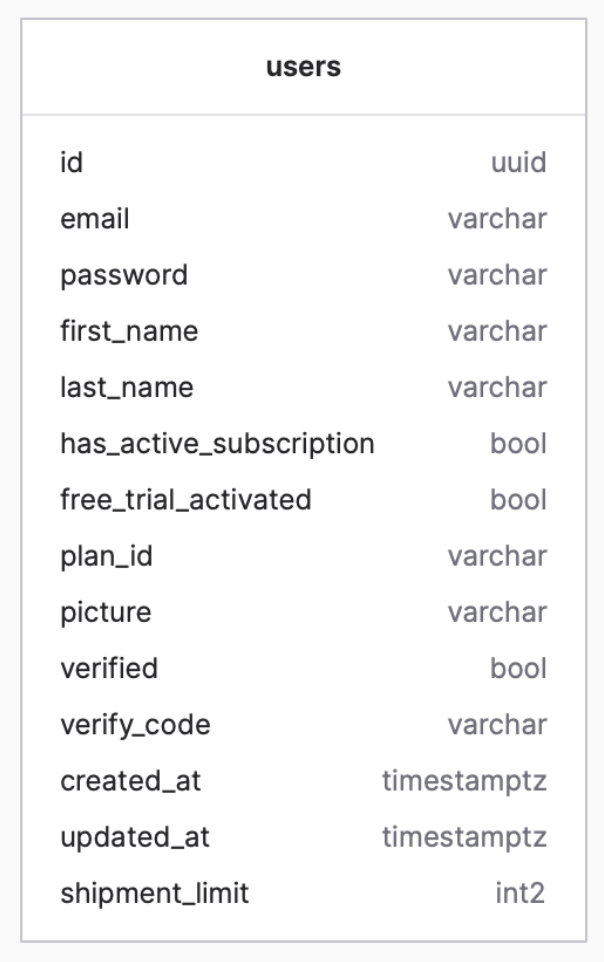


Figure A.3: Database schema diagram of `Users` related tables

A.4.2.3 Shipments

The shipment is the object with which the user comes into contact the most and also the most important part from a business perspective. For the database diagram, please refer to Figure A.4. Shipment is a complex object constructed from:

- **Parcels:** Representing the actual shipped box with a label and tracking number. Shipment can have multiple parcels.
  - **Parcel status:** Status of each parcel with metadata provided by carrier.
    - **Parcel status type:** A unified type of parcel status with user visible translated textual data. It can take the following values:
      - `EXCEPTION`

        - DATA_SENT
        - ACCEPTED_BY_CARRIER
        - HANDED_OVER
        - IN_TRANSIT
        - OUT_FOR_DELIVERY
        - STORED_FOR_PICKUP
        - PICKED_UP
        - DELIVERED
        - DELIVERY_FAILED
        - RETURNED
        - CANCELED

- **Shipper:** Universal shipper of the project.
- **Recipient:** The recipient sent to the carrier with the address or pickup point and contact information.
- **Customs, commodities:** For a customs clearance, the user needs to provide information about the parcel.
- **Insurance, payment:** To define a insurance value with currency, as well as payment type and amount.

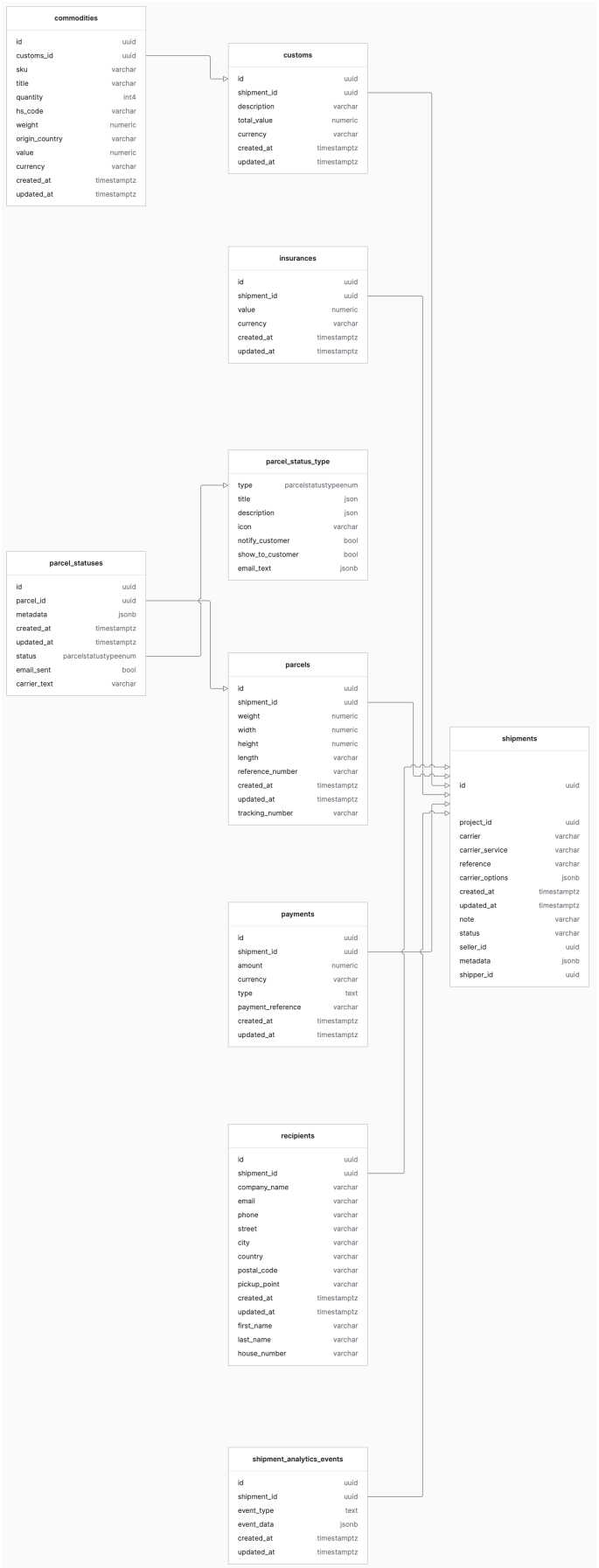


Figure A.4: Database schema diagram of `Shipments` related tables

A.4.3 Endpoints

The backend is structured as an API utilising REST principles. Each endpoint corresponds to a specific function within the platform, allowing CRUD operations on resources such as shipments, users, projects, and operations calling carrier APIs. All endpoints with related middlewares and actions are defined in `router.ts`.

A.4.4 Authentication and authorisation

User authentication and authorization is implemented with respect to the best practices of Koa as a middleware. With this convenient approach, we can "decorate" each route of the API with `authenticationMiddleware` and, if needed, `authorizationMiddleware` handling the crucial logic of retrieving user credentials, verifying them and passing them into the `tsyringe` container for other middlewares, actions or services to utilise the user object.

A.4.4.1 Authentication flow

The authentication is based on token authentication.

1. **Login request:** Users submit their credentials via `/authentication/signin`
2. **Credential verification:** Credentials are verified against the database entries using hashed passwords.
3. **Token generation:** Upon successful authentication, an access and refresh JSON Web Token (JWT) is generated and returned to the user.

A.4.4.2 Session management

- **Token storage:** JWTs are stored client-side and included in the HTTP authorization header (with `Bearer` prefix) for requests.
- **Token expiry and refresh mechanism:** Tokens have finite lifetime, after which user must either refresh the token or re-authenticate.

A.4.4.3 Authorization

Authorisation is solely based on an authenticated user. Each user can have a single role, being either:

- Owner
- Admin
- Member

A.4.5 Request body validation

Validating request body on backend is a necessity. Improve the integrity of the data and the security of the entire system. For this task, a `Joi` library was chosen. It allows us to simply describe the schema of request body and validate against it.

For an endpoint requiring request body validation it is necessary to define the schema. For a larger schema, we recommend that you define it in separate file from the file containing the action. Then the schema is utilized in the `schemaValidationMiddleware`.

For example, a schema describing request body where either a list of shipment references or objects containing tracking numbers with carrier name is provided, will look like this:

```
1  import Joi, { Schema } from 'joi';
2
3  const schema: Schema = Joi.object({
4    shipments: Joi.array().items({
5      reference: Joi.string().required(),
6    }),
7    trackingIds: Joi.object().pattern(
8      Joi.string(), // Key as string (carrier name)
9      Joi.array().items(Joi.string()) // Value as array of strings
10   ),
11 }).xor('shipments', 'trackingIds'); // Ensure either shipments or
       trackingIds is provided, but not both
```

Listing A.1: Joi schema example

In order to comply with the DRY principle, `Joi` schemas are also used to generate OpenAPI specification for the public API. Each public API should export `specification` object which, if exists, should contain the `Joi` schema converted to `swagger` format using `joiToSwagger` method.

A.4.6 Public API

So-called "public" methods rely strictly on `publicApiMiddleware`. This middleware controls provided `key` in request headers and attempts to resolve a project related to that value. Public API methods are prefixed with `v1` in the path, and it is necessary to always document them. Thanks to that we can differentiate whether the request is coming from the user interface or external integration. This is especially useful when working with shipments. The shipment alteration uses the UI-first approach, meaning that all shipments created or modified within the user interface cannot be modified through the API. This is because the integration will usually be done as a periodic data sender which will repeatedly send shipment data until they are marked as "SENT". Hence, overwriting user's change which we do not want.

A.4.7 OpenAPI schema generation

After touching on both the topics of request schema description and public API methods, it is a good time to describe the generation of the OpenAPI specification and a Swagger UI. The initialisation of an instance of the `OpenAPI`

class is carried out while setting up the server in the Lambda handler. Each public API endpoint must be registered in the `OpenAPI` class constructor using the `registerSpec` function with parameters of the endpoint path and its specification. Since the `OpenAPI` class gets the reference to the Koa router instance as a parameter, we also define the routes `/openapi.json` and `/docs` routes. The Swagger UI is exported from `openapiUI` method containing the actual Swagger UI HTML with our OpenAPI specification retrieved from `/openapi.json`.

A.4.8 Data filtering

Backend handles and provides a lot of data, for this reason it is necessary to provide convenience of data filtering while using the REST API methods provided by backend. For this purpose `filterParsingMiddleware` was defined. This middleware handles the task of parsing query parameters which serve for data filtering. This filtering syntax is built around this parameter format: `?filter=[{FIELD}|{OPERATOR}]={VALUE}`
Where:

- `FIELD` can be arbitrary field from the response
- `OPERATOR` can be of two types - unary operator or multi-value operator :
  - **Unary:**
    - `EQUALS`
    - `CONTAINS`
    - `STARTS_WITH`
    - `ENDS_WITH`
  - **Multi-value:**
    - `IN`
    - `BETWEEN`
- `VALUE` can be either a single value when paired with unary operator, or multiple values separated by comma, if multi-value operator is used.

These filters are parsed and stored as an array of objects in format `{[FIELD]: {VALUE, OPERATOR}}` in the Koa context as the `filters`. From this object a database query is then constructed in a service method to which is the object passed as a parameter.

A.4.9 Data pagination

Handling pagination on backend is essential task with a growing dataset. For this purpose, `paginationMiddleware` with supportive types and methods such as `responsePagination`. This ensures a unified format of query parameters for both `page` and `pageSize` parameters, as well as a response format utilising few of the RESTful principles that return the paginated data set with links to retrieve other data. The response format is:

```
type ResponsePagination = {
  page: number;
  pageSize: number;
  offset: number;
  dataLength: number;
  total: number;
  links: {
    first: string;
    previous: string | null;
    next: string | null;
    last: string;
  };
};
```

Listing A.2: Response pagination type

A.4.10 Carrier communication

Integration of shipping carrier APIs is build around abstraction, trying to unify carriers as much as possible to provide seamless user experience without caring about specific details related to each carrier. All carrier implementations are stored in `src/modules/carrier` where each carrier implementation inherits from `AbstractCarrierModule`. This abstract class requires from each implementation to contain several public methods which are then used for sending the data, generating files such as labels and waybills, as well as updating or retrieving parcel statuses.

Each carrier is then registered in a dedicated service `carrier-service` supporting the logic of hiding the details from user experience and, hence, ideally, treating every carrier, from user perspective, as if it was just one.

However, given the fact that each carrier treats their API development differently, there are always some specifics that we cannot fully hide. They will be described in the following section.

A.4.10.1 Packeta

Let us start relatively lightly. The Packeta API generally does not contain any special features, and the function structure more or less copies the functions in the abstract class. However, what needed to be modified is the inability to send multiple parcels in a single shipment. This was done by breaking the shipment into several requests. This brings us to the next and final point; unfortunately, Packeta does not allow batch shipping of shipments and everything is sent one at a time. This means a rather lengthy sending of a large amount of data, but due to the relatively fast response time, this does not limit us to the maximum timeout of Lambda functions.

A.4.10.2 Česká Pošta

For the integration with Česká Pošta, our approach leverages asynchronous data transfer techniques to manage parcel shipping. Unlike synchronous operations, parcels are not immediately confirmed upon dispatch. Instead, the pickup

details are stored within the shipment metadata and must be retrieved through a separate mechanism. This integration utilises something like a "step function" approach in the Lambda architecture, which allows for asynchronous processing of shipments. This method is particularly advantageous for handling operations that do not provide immediate results, such as waiting for pickup confirmations or processing delayed status updates. The response retrieved from the request method is stored in the database queue, which is then picked up by separate "pick-up" function retrieving the final response.

One notable limitation with the Česká Pošta API is that the status updates provided only include the date, not the time. This lack of granularity can complicate logistics tracking when precise timing is essential. Additionally, we have encountered several status codes that are not documented in the official Česká Pošta API documentation. Such discrepancies suggest that the API might undergo updates or changes that are not immediately reflected in the documentation, posing a challenge to maintaining accurate integration.

A.4.10.3 PPL

The integration with PPL similarly employs an asynchronous approach to data handling, much like the implementation for Česká Pošta. An essential feature of the PPL integration is the handling of shipping labels. Labels are generated at the PPL API side during the data transfer process and are crucial for the physical shipping of parcels. To accommodate the needs of users who may not immediately retrieve these labels, they are stored as an URL reference the shipment metadata and can be accessed at any point within 20 days after the data transfer. This solution ensures that labels are available whenever needed during the period. The only downside is that the labels are returned one by one from the PPL. So when requesting labels from multiple parcels, it is necessary to obtain each label separately as a JPEG and then construct a single PDF with 4 labels per page using `Joi`. The method responsible for this operation is `_mergeLabels` defined in `pplModule`.

A.4.11 Generating PDF waybills

Generating PDF waybills meant as a list of shipped parcels with courier's signature is a necessary feature for any shipping operation, providing a document that can serve as a reliable backup in case of discrepancies or issues during the shipping process. Waybills are generated as A4-sized PDFs using the `pdfjs` library. The process of generating these documents is designed to be flexible and to accommodate various needs and specifications required by different carriers and regulations. Specifically for Packeta, the waybills include a uniquely generated barcode from the Packeta API. This barcode facilitates a faster and more efficient parcel pickup process, as the carrier scans it to verify and process the shipment. The inclusion of a barcode streamlines the logistics chain, minimising errors and speeding up the time spent in the warehouse.

A.5 Frontends

There are currently three different frontends of two types in the platform. The first is the user documentation generated by Doctosaurus. It is located in the `clients/docs` folder and contains two language versions, Czech and English. We will not go into the documentation any further since it follows the official documentation of Doctosaurus.

Let us take a look at the second type of frontend - React. The platform contains two React frontends - `clients/web` serving dashboard and `clients/tracking` for displaying tracking information to the recipient. In this section, we will describe only the React frontends.

A.5.1 Overview

Both React frontends are of similar structure and conventions respecting the typical React project structure with `src` directory. Both projects use functional components with hooks for data fetching and processing.

A.5.2 State management

Elevating Context API principles: passing data through components is done mostly using providers defined within the `src/providers` folder. This enables to toggle the loading mode, as well as to manage currently selected project in order to differentiate tenants.

A.5.3 Routing

Routing is achieved using `react-router-dom` package where every protected project route is defined by the project ID in the path. Each of these protected paths is wrapped in the `AuthenticatedRoute` component that controls the validity of the user on every request.

A.5.4 Data fetching

The platform employs a structured approach to data retrieval across its frontend applications, centralizing the logic within custom React hooks to enhance modularity and reusability. Each hook is designed to interact with specific backend endpoints, handling various operations such as listing, creating, updating, and deleting projects.

Data fetching operations utilize the `useApiActions` custom hook to perform API calls. This setup abstracts the complexities of HTTP requests and state management, providing a clear and straightforward API for frontend components to interact with the backend. The `executeApiAction` function, central to the mentioned hook, manages the life cycle of API requests. It prepares and sends HTTP requests using the `ky` library configured with essential hooks for error handling and token refresh logic if needed for protected routes. Upon completion of an API call, feedback is provided through toast notifications.

A.6 Integrating new features

This section provides guidelines for the integration of new features into the platform. Covers the addition of new logistics carriers, the management of environment variables, the handling of frontend metadata, and the implementation of localisation strategies.

A.6.1 Adding carriers

Integrating a new carrier into the platform involves several key steps to ensure a seamless interaction between the platform and the carrier's services. This integration typically includes API communication, data parsing, and updating the UI to reflect the new carrier's options.

Steps to integrate a new carrier:

1. **Create carrier type:** Create new value in `ShippingCarrierId` enumeration in `config/types.ts`.
2. **Define carrier type:** In the central platform configuration `app.config.ts` create a entry in the `shippingCarriers` array containing the newly defined carrier with all the necessary authentication and generic fields.
3. **Defining carrier module:** To create the implementation of the carrier API, it is necessary to create a new file in `src/modules/carrier` in the backend that contains a class that implements `AbstractCarrierModule`.
4. **Register the carrier:** In `src/services/carrier-service.ts` on backend in the method `_createCarrierModule` register the newly created module under the `ShippingCarrierId` value.

With these steps involved, a new carrier should be integrated into the platform.

A.6.2 Database migrations

A database migration refers to version control of a database schema. Database migrations produce incremental changes to the schema. Sometimes, these changes can be reversible. Migrations can create a table, add a column, remove it, rename it, or change the type.

All migrations are stored in the `lib/db/migrations.ts` file within the `api` directory. Each migration is inside an array that fully respects the format expected by Knex.js library specified in the Knex.js documentation. The migration might look like this:

```
1 ...
2 {
3     name: '16_create_shipment_analytics_events_table',
4     up: async (knex: Knex) => {
5       await knex.schema.createTable('shipment_analytics_events',
    (table) => {
6         table.uuid('id').defaultTo(knex.raw('uuid_generate_v4()')
    ).primary();
7         table
```

```
8             .uuid('shipmentId')
9             .notNullable()
10            .index()
11            .references('id')
12            .inTable('shipments')
13            .onUpdate('CASCADE')
14            .onDelete('CASCADE');
15          table.enum('eventType', ['view', 'click']).notNullable().
    index().defaultTo('view');
16          table.jsonb('eventData').notNullable();
17          table.timestamps(true, true);
18        });
19        return knex;
20      },
21      down: async (knex: Knex) => {
22        await knex.schema.dropTable('shipment_analytics_events');
23        return knex;
24      },
25    },
26  ...
```

Listing A.3: Example of a migration from `migrations.ts`

Running migrations in both local and AWS environment is very straightforward. We can utilise the following commands which are environment-biased and will run migrations only within the specified environment as long as the database is accessible. Go to the `services/api` directory and run:

- `yarn migrate:up:<environment>`: Runs the first unapplied migration in the list.
- `yarn migrate:down:<environment>`: Reverts the last migration applied.
- `yarn migrate:latest:<environment>`: Runs all unapplied migrations.

Where the environment can be one of: `local | staging | production`.
However, in AWS environments, we recommend leaving migrations for the GitHub action triggered on every push to the `main` branch.

A.6.3 Adding new environment variables

Adding new environment variable to the whole platform is a very straightforward process.

1. Extend the enumeration `EnvironmentVariable` located in `config/types.ts` by the newly added value. If the value should be optional, modify the `OptionalVariables` type.
2. In `app.config.ts` locate `environments.environmentVariables` and add a new entry loaded using the `getEnvVar` method.
3. For local development, add value to the `.env` file, for production and staging purposes, add the `StringParameter` to the `api-stack`.

A.6.4 Passing metadata to frontend

Metadata provided by the backend are fetched on every load from the frontend. With this mechanism, we can provide the necessary data to the frontend before every page load. Bare in mind, that response of this request must be quick - hence, ideally providing just static data. The process is straightforward - only modify the
`src/actions/metadata/app-config.ts` action on backend.

A.6.5 React Frontend localisation

Localisation of both React client applications is done using `i18next` library. All translation files are stored within the project in `public/locales` folders. If a new translation key is added, simply run `yarn generate:locale` within the workspace, and the keys will be generated or modified.

A.7 Infrastructure

This section of the documentation delves into the infrastructure setup that supports the platform's operational and programming decisions. A primary focus is placed on explaining how certain infrastructure choices are closely tied to development practices and user features, such as the direct upload of static assets from the client to the cloud.

A.7.1 Static Asset upload from client

The platform provides the ability to upload static assets directly from the client to an AWS S3 bucket using presigned URLs, which allows secure, direct browser uploads without exposing server credentials. This method optimises the upload process by reducing the server load and network traffic that would otherwise be required if the server had to act as an intermediary.

The backend setup involves a specific route and action handler that generates these presigned URLs. The action, defined in the `generateSignedUrlAction`, prepares a URL that allows a client to put a file directly into an S3 bucket under a specified path that includes the project ID and the file name. The `generateSignedUrlAction` performs the following steps:

1. **Validation:** Ensures required parameters are provided and valid using a Joi schema.

2. **Presigned URL Generation:** Uses the AWS SDK's S3RequestPresigner to generate a presigned URL that allows the client to upload a file with public read access. The URL is valid for a short period (e.g., 60 seconds) to enhance security.

3. **Response:** Sends the generated URL back to the client, which can be used to upload the file directly to S3

On the frontend, the process to upload a file involves:

1. **Requesting the Presigned URL:** The frontend makes a POST request to the backend to fetch the pre-signed URL, providing the file name as part of the request.

2. **File Upload:** Using the received URL, the frontend performs a PUT request using fetch API to upload the file directly to the S3 bucket. The frontend handles upload progress and completion status.

This setup is beneficial for use cases where clients need to upload images or documents related to project sellers, such as banners, logos, or other relevant files. The use of AWS S3 ensures scalable and secure storage, while presigned URLs keep AWS credentials safe and provide a method to control access rights on a per-use basis.

A.7.2 Sending e-mails

The functionality for sending emails is defined in `communication-service.ts` within `sendEmail` method. Using AWS Simple Email Service to handle outgoing emails. This method is responsible for constructing and sending an email through AWS Simple Email Service. It is designed to be flexible, supporting both plain text and HTML email bodies, as well as additional email parameters such as CC, BCC, and reply-to addresses. The method accepts the following parameters to construct the email:

- **email (string)**: The recipient's email address.
- **subject (string)**: The subject line of the email.
- **textBody (string)**: The plain text body of the email. This is the fallback option for clients that do not support HTML.
- **fromOrganization (string)**: The email address of the sender. By default, this is set to the environment variable `ORGANIZATION`, which should be configured in the system environment settings.
- **htmlBody (string)**: An optional parameter for the HTML body of the email. If provided, it allows the email to include HTML formatting.
- **replyTo (string)**: An optional parameter specifies the reply-to email address.
- **bccAddresses (array of strings)**: An optional array of BCC (blind carbon copy) addresses for sending the email to additional recipients without disclosing their identities to other recipients.

The method initialises a connection to AWS Simple Email Service using AWS SDK. The AWS Simple Email Service client is configured with the necessary credentials and region information, which are obtained from the environment settings of the platform. Once the client is initialised, the `sendEmail` method constructs the email parameters into the format required by AWS Simple Email Service and sends the email using the `SendEmailCommand` from the AWS SDK.

A.7.3 Time consuming functions

In the platform architecture, certain operations, such as updating shipment statuses with carriers like Česká Pošta, are time-consuming and require handling that goes beyond the typical execution time limits of AWS Lambda functions. To address this, the system uses a two-part method involving request and pickup processes that efficiently manage these operations within Lambda's constraints. The process is divided into two distinct Lambda functions:

1. **Request Function:** This function is responsible for initiating requests to the carrier's API (e.g., Česká Pošta) for shipment status updates. It handles communication with the external API and then saves the immediate response, which often includes a queue or task ID, in a PostgreSQL database queue. This method allows the function to complete within a short runtime, adhering to Lambda's execution limits.

2. **Pickup function:** After the request function stores the task ID in the database, the pickup function takes over. Scheduled or triggered after a predefined interval, this function retrieves the task ID from the queue and makes a subsequent API call to fetch the actual status update. Once the data are received, it processes and integrates the information into the platform's operational flow, updating shipment statuses accordingly.

A.7.3.1 Scalability with Step Functions

With the platform's user base expanding and the volume of shipments growing, the existing method of handling status updates through separate Lambda functions might not scale efficiently. Transitioning the current setup to `AWS Lambda Step Functions` could offer a more robust solution by orchestrating these operations more dynamically and resiliently.

A.8 User documentation

User documentation for the platform's dashboard user interface is created using Doctosaurus, a dedicated documentation generator that handles the creation of structured and navigable content. The documentation files are located within the `clients/docs` directory of the monorepo, ensuring easy access and version control along with the codebase. The documentation is available in two languages: Czech and English. Any changes to the user interface, modifications to existing functionalities, or the introduction of new features that directly impact how users interact with the platform must be promptly and accurately documented. This practice ensures that the documentation remains a reliable and up-to-date resource for end-users, helping them to effectively utilise the dashboard and understand its full range of capabilities.

B. Programming Documentation - SAP Business One ServiceLayer Proxy with Database Connector

The SAP Proxy serves as an intermediary between client applications and SAP Business One ServiceLayer ensuring secure and efficient data exchanges. Its primary function is to authenticate, authorise, and process requests. It is designed to operate in both development and production environments, utilising a connector to both development and production SAP underlying MS SQL databases and SAP ServiceLayer tokens.

B.1 Workflow

Requests are received at various endpoints, processed by middlewares for security checks, and then routed to the appropriate actions, utilising services managing database operations or interactions with SAP ServiceLayer or MS SQL database connection. Responses are generated based on the outcome of these interactions and sent back to the client.

B.2 Overview

The SAP Proxy holds active connections to both environments of the SAP ServiceLayer as well as the Microsoft SQL database, which underlies the SAP. With this proxy, it is possible to call requests on both environments just by specifying route segment `.../<env>/...`. The initialisation is performed by convention in `src/index.ts` which creates an instance of the Koa.js application by calling the `createServer` method from `/src/server.ts`. Here, the initialisation occurs. Create a new Koa instance with the necessary settings, router, and initialising the Microsoft SQL connection. The SAP Proxy is built around several key components in the following order:

1. **Router:** Is pretty much the entry-file where all endpoints are defined.
2. **Middlewares:** Are chain-able methods initiated at the beginning of endpoint request or at the end to modify response. They handle authorization, authentization, and even environment retrieval.
3. **Actions:** Actions present the ”called” method of the endpoint after request middleware.
4. **Entities:** Objects stored in the connected database.
5. **Services:** Manage database interactions, user data processing, and connectivity with SAP ServiceLayer and Microsoft SQL database.

The folder structure of the program is meant to group components of similar purpose together:

```
config
├── types.ts
└── utils.ts
scripts
└── build.js
src
├── actions/
├── constants
│   └── error-names.ts
├── entities
│   ├── sap-tokens.ts
│   └── user.ts
├── errors
│   └── service-error.ts
├── lib
│   ├── db
│   │   ├── knex.ts
│   │   ├── knexfile.ts
│   │   ├── migration-source.ts
│   │   ├── migrations.ts
│   │   ├── seed-source.ts
│   │   └── seeds.ts
│   └── mssql
│       └── connection.ts
├── middlewares/
├── services/
├── types/
├── utils/
├── router.ts
└── server.ts
```

We will go through the most important directories and files.

B.2.0.1 MSSQLConnection

Since one of the requirements is to initialise both development and production environments at the same time, it was necessary to eliminate the cold starts caused by logging into the database and request time. For this purpose, the class `MSSQLConnection` (located in the file `src/lib/mssql/connection.ts`) was created with a static mapping of connections `<environment, MSSQLConnection>` containing a pool of connections for each environment.

Connection is done through the `mssql` package creating `mssql.sql.ConnectionPool(config)` with config passed into it. As it turned out during development, it is not possible to simply initialise multiple connections calling `mssql.sql.connect(config)`. In this way, the package silently denies creating other connections than the first one initialised. For different connection configurations, it always returns the reference to the first connection pool initialised. Hence why it is necessary to initialise `ConnectionPool` directly, instead

of calling the `connection` wrapper.

If one connection fails for some reason during the request, a new one is initiated. This may slow down the data retrieval but will ensure that, as long as the database is reachable, the proxy will try to connect. The pool retrieval from anywhere within the program is done by calling `getInstance(env)` on `MSSQLConnection` where `env` parameter is `dev` or `prod`.

B.2.1 Router

The file representing router `src/router.ts` defines endpoints and HTTP methods with middleware chains together with the result action.

B.2.2 Middlewares

Middlewares are chained methods within the router for each endpoint. The SAP Proxy defines middlewares for:

- Schema validation
- Authentication
- Authorization
- Environment resolver
- SAP ServiceLayer login

B.2.3 Actions

Actions present the actual method called by the endpoint. The usual workflow of these methods defined in `src/actions` is to retrieve a request validated or authorised by the middleware chain with data passed within the context (such as body, environment or instance of the ServiceLayer), and call some service B.2.5. The service usually returns some data which are then passed into the response.

B.2.4 Entities

Entities defined in SAP Proxy are TypeScript classes that extend `Model` from Objection.js and are directly mapped to the database using modules Knex.js and Objection.js.

B.2.4.1 `SAPToken`

Entity used for caching the login token for SAP ServiceLayer. It's model directly mapped to the database which hold data such as:

- `token`
- `environment`
- `expiry`

This ensures that whenever user calls the SAP ServiceLayer Proxy, we retrieve the token from the database or refresh the token in SAP ServiceLayer based on the environment and expiration.

B.2.4.2 `User`

User entity presents the database model mapping stored for authentication and authorization of the SAP ServiceLayer Proxy. Containing fields such as:

- `username`
- `password`
- `role`
- `email`

B.2.5 Services

Services represent the place where data communication between the application and the database or the SAP ServiceLayer is performed.

B.2.5.1 `sap-service`

This service represents connector to both the SAP ServiceLayer and `SAPToken` entity in the database. It is located at `src/services/sap-service.ts`. The service methods for the SAP ServiceLayer login for token retrieval, token caching in the database for each environment, and proxy call from method `proxy` in `SAPService` while returning `Stream` object.

B.2.5.2 `user-service`

User service performs database operations on User Entity. Containing methods such as `signup`, `changePassword`, `authenticate` and much more.

B.3 Error handling and logging

This section describes the practices used in the SAP Proxy to manage errors and log activities.

B.3.1 Error handling

The application uses a central error handling mechanism to capture and process errors and exceptions in a uniform way. Koa Middleware is used to wrap all endpoint handlers. This middleware catches any uncaught exceptions thrown during the request life cycle. Once caught, these exceptions are passed to a centralised error handling function which determines the type of error and the appropriate response to send back to the client.

B.3.1.1 HTTP status codes

Depending on the type of error, the handler sets the HTTP status code appropriately. For example, validation errors might return a 400 Bad Request, authentication errors a 401 unauthorised, and internal server errors a 500 Internal Server Error.

B.3.1.2 Error responses

All error responses are formatted in a consistent structure, which includes an error code, a human-readable message, and optional additional details that could help debugging.

B.3.2 Logging

The logs include the timestamp, error type, endpoint involved, and a stack trace for severe errors. The application uses logging framework `Winston` with appropriate configuration to differentiate log levels (info, warn, error) and to output formats.

C. Programming Documentation - `Data-sender`

The `Data-sender` program is designed to facilitate data communication between the 'SAP Proxy' and the main platform's public API as shown on Figure C.1 where `Data-sender` is highlighted in red. It handles the exchange of order and tracking information, ensuring that data flow is synchronised to-date across systems. This program acts as a middleware that not only transmits data to and from SAP, but also formats the data according to the requirements of the main platform.

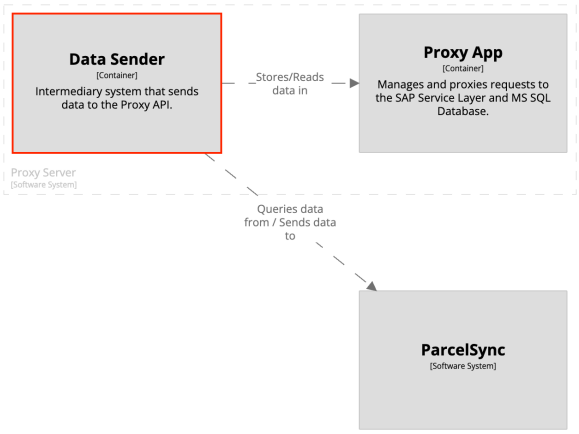


Figure C.1: C4 Container diagram of `Data-sender` context

C.1 Data flow

The data flow through `Data-sender` can be generalized into two main points:

- **From SAP to platform:** New and existing order details are loaded from SAP and sent to the platform. This includes orders that need platform processing and updates for existing (non-shipped) orders with modifications.
- **From platform to SAP:** Completed orders, including those shipped and delivered, are updated back in SAP with details such as tracking number, actual status, delivery date, and carrier specific data such as invoice number, weight, and name of the actual signed recipient.

C.2 Overview

The `Data-sender` is designed with two entry-points:

1. **Scheduler:** Manages the timing of data exchange tasks, ensuring that they are executed at appropriate intervals.

2. **CLI:** Allows manual triggering and execution of specific scripts.

So, the program can run in scheduled mode as well as in nonscheduled mode just by triggering the command. The folder structure of the program is very straight forward and defined the module separation:

```
src
├── tasks
│   ├── ceskaposta
│   │   ├── new-orders.ts
│   │   ├── update-new-orders.ts
│   │   └── update-old-orders.ts
│   ├── packeta
│   │   ├── new-orders.ts
│   │   ├── update-new-orders.ts
│   │   └── update-old-orders.ts
│   └── ppl
│       ├── new-orders.ts
│       ├── update-new-orders.ts
│       └── update-old-orders.ts
├── types
│   └── task.ts
├── utils
│   ├── parcelsyncApi.ts
│   └── sapApi.ts
├── logger.ts
└── setup.ts
```

As we can see from the structure, in `src/tasks` there is a lot of redundancy in tasks. The reason for this is that the company implementation of SAP handles each carrier slightly differently. Due to that, the tracking numbers for the carriers are inserted into different columns and the data is fetched differently. Hence why, the cleanest and well maintainable approach was to implement this completely separately as isolated scripts.

C.3 API fetchers

In order to call SAP Proxy and the public API of the platform, two instances of Axios were created in `src/utils/`. Each of them calls different endpoints and handles authentication a little differently.

- **`parcelsyncApi.ts`:** Defines AxiosInstance to call the API of the platform with support scripts to generate seller identification and defines several groups of parcel statuses to help with recognition whether the parcel was delivered, etc.

- **`sapApi.ts`:** Defines `AxiosInstance` to call SAP Proxy.

C.4 Scheduler and CLI

All tasks available to schedule and call via CLI are imported in `src/setup.ts` which is called from the entry `src/index.ts`. The `setup.ts` defined a mapping for each task between the method and the standard Cron schedule definition.

Each task is then bound to the Axios module object as a command to run either from CLI or as a single command triggering the scheduled mode.

C.5 Carrier specific modules

For each carrier implemented and used, there is a separate module that handles data retrieval and alteration.

Each carrier implements three main methods:

- **`new-orders`:** Queries data in SAP via SAP Proxy, transforms the data into expected format and sends to the platform. This task implements the data query with specifics for each carrier.
- **`update-new-orders`:** Retrieves packages from the platform up to 1 day old and alters the tracking number, tracking link, status name and status date in SAP via SAP Proxy.
- **`update-old-orders`:** Retrieves parcels from the platform that are up to 20 days (`ceskaposta`, `packeta`) day old (for `ppl`, it is 35 days, more on that in Section C.5.3) and alters the tracking number, tracking link, status name and status date in SAP through SAP Proxy.

C.5.1 `ceskaposta`

Implements the three standard methods as mentioned in C.5.

C.5.2 `packeta`

Implements the three standard methods as mentioned in C.5.

C.5.3 `ppl`

Implements the three standard methods as mentioned in C.5, however, with some specifics to the PPL carrier.

C.5.3.1 Multiple parcels in one shipment

One specific feature of the PPL is that its API allows multi-parcel shipments. This means that we can group the data retrieved from SAP using a grouping parameter (Invoice number) and send those shipments to the platform with multiple parcels.

C.5.3.2 Retrieval of Invoice number and price of the service

Since PPL provides much more data than the two carriers mentioned above, it is possible to retrieve the PPL invoice number and the granulated cost of the shipping service. The platform returns these additional data obtained from tracking in `metadata` field of the status. This can be parsed and inserted into the appropriate SAP columns in the shipment object. Because these data are all returned on the first day of the next calendar month, it is necessary to fetch more much older shipments from the platform (hence the 35-day parameter).

D. Administration Manual - Platform

D.1 Local development

Local development is essential to test new features and debugging. This section outlines the requirements and steps to set up your local development environment for the platform, ensuring that all components function together seamlessly.

D.1.1 Prerequisites

Before setting up the local development environment, ensure that you have the following prerequisites installed:

- **Node.js $\geq$ 16**
- **Docker**
- **Yarn**

D.1.2 Running backend and frontend services

1. Once you have `yarn` and `Node.js` installed - preferably by using some package manager like `asdf`.
2. Run `yarn install` to install all the dependencies needed.
3. Set up a public S3 bucket to upload files from seller configurations.
4. Create a `.env` file and place it in the root folder. The file should contain the following:

```
JWT_SECRET_KEY=<key>
JWT_ACCESS_LIFESPAN_SECONDS=3600
JWT_REFRESH_LIFESPAN_SECONDS=604800
AWS_ACCESS_KEY_ID=<aws_access_key_id>
AWS_SECRET_ACCESS_KEY=<aws_secret_access_key>
AWS_SECRET_KEY_ID=<secret_key_id>
PARCELSYNC_AWS_ACCESS_KEY_ID=<aws_access_key_id>
PARCELSYNC_AWS_SECRET_KEY_ID=<aws_secret_access_key>
FROM_EMAIL=<from_email>
AWS_ACCOUNT=<aws_account>
AWS_S3_ASSETS_BUCKET=<s3_bucket_url>
```

Listing D.1: Platform local environment configuration

5. Run `yarn start:dev` and all three services (both frontends and API) should start up with database.

The `aws_access_key_id` variable appears twice in the environment settings due to differing requirements for its usage. Firstly, it is required in its conventional

format as recognised by AWS services. Secondly, a distinct instance of this variable is necessary because in the `generate-signed-url.ts` the call of AWS S3 cannot adhere to the AWS naming conventions and hence requires a separate declaration of the `aws_access_key_id` variable in the environment configuration.

D.1.3 Running database

Navigate to the `services/api` directory containing the `docker-compose.yml` file which includes the service definition for the database. Run the following command to start the service: `yarn dependencies:start`. This command will pull the necessary images and create container for your database and start the service on the background.

D.2 Administration Manual - AWS Infrastructure

This section provides a comprehensive guide to managing the platform's AWS infrastructure. Although the implementation and integration processes mainly leverage IaC for efficiency and consistency, there are instances where manual intervention is required. Accessing logs, managing database credentials, and performing specific tweaks often require direct interaction with the AWS Management Console. This section outlines essential administrative tasks and highlights adjustments and configurations that were manually implemented during the deployment phase, providing information about how to effectively navigate and manage the AWS environment.

Please note that as long as there is an option, all infrastructure should be deployed within eu-central region.

D.2.1 Lambda

The backend service fully utilises *AWS Lambda* for the deployment and related services. This section will describe managing of the Lambda handlers of both backend service and scheduled tasks.

D.2.1.1 Accessing logs

Logs of the Lambda handlers are accessible within the AWS console in *CloudWatch* section. Here it is possible to list all the log groups within whole account where are all the deployed services as well as Lambda handlers used as scheduled tasks and those ran while deployment (migrations).

To access the logs at a specific time, click the desired log group (usually labelled production environment see Figure D.1) and scroll through the log stream (see Figure D.2) to find the time and event you want (see Figure D.3).

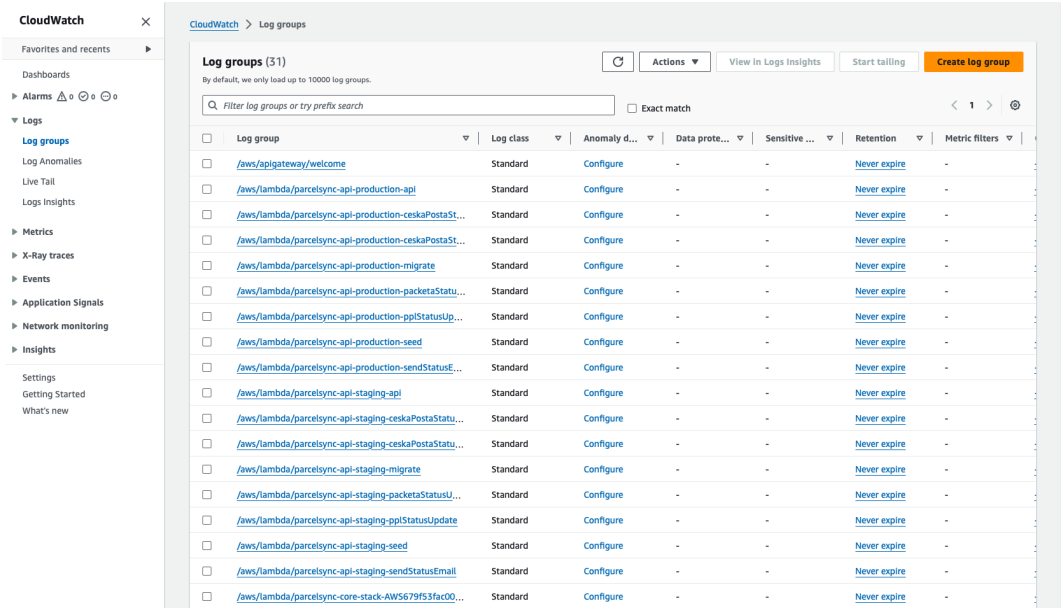


Figure D.1: AWS CloudWatch

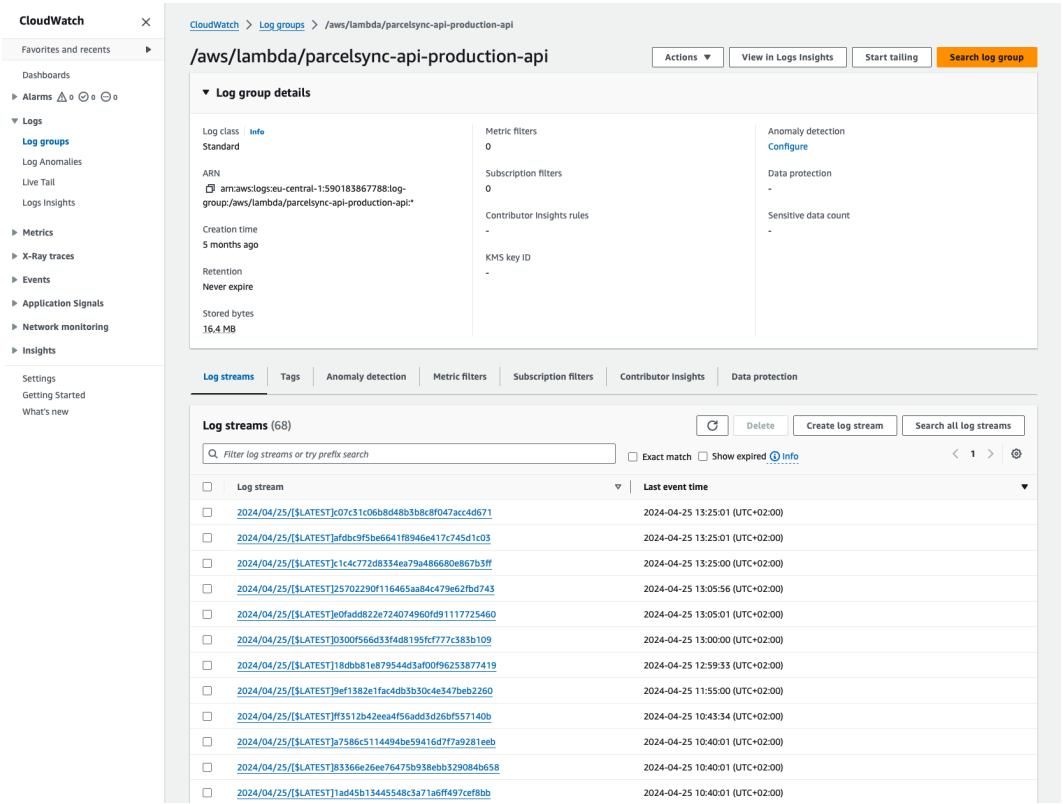


Figure D.2: AWS CloudWatch Log Group detail

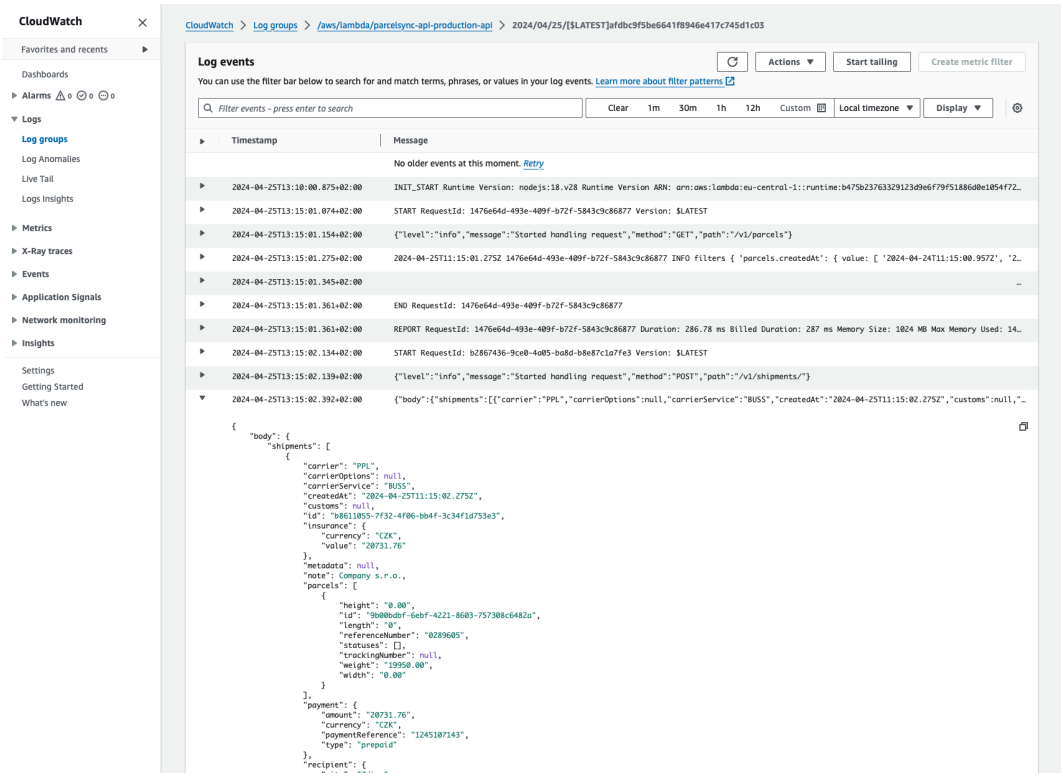


Figure D.3: AWS CloudWatch Log Group event detail

D.2.1.2 Scheduled tasks

Scheduled tasks can be found in the list of all Lambda functions. They are conveniently named and are used to retrieve shipment status from supported carriers and send tracking emails to customers. Each of these tasks has *EventBridge* attached with configuration of the event trigger (cron-like definition), see Figure D.4 We currently use these Lambda functions as scheduled tasks:

- `packetaStatusUpdate`: For Packeta parcel status retrieval.
- `pplStatusUpdate`: For Packeta parcel status retrieval.
- `ceskaPostaStatusRequest`: To request parcel statuses within given timeframe.
- `ceskaPostaStatusPickup`: To pickup requested parcel statuses and update them in the database.
- `sendStatusEmail`: To send-out status e-mails based on seller settings (allowed statuses).

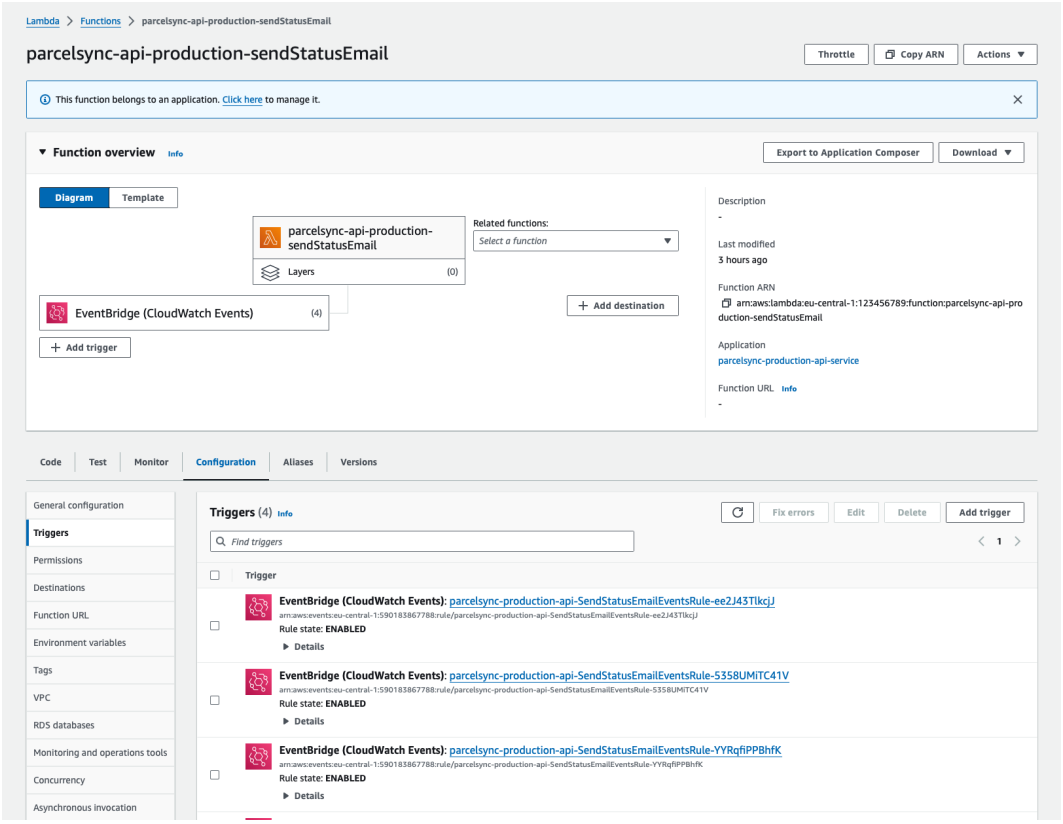


Figure D.4: AWS Lambda EventBridge definition

D.2.1.3 Lambda handler functions

Lambda handler functions are the entry point methods which are executed when the `lambda` function is started. We define the Lambda handler function for all scheduled tasks listed in D.2.1.2 as well as for the backend service. Each lambda function can be triggered manually; however, it is recommended to do so only with the scheduled tasks and tasks that run migrations or seeding the database. You can create the URL of the function in the details of every Lambda function. This will trigger the function to run, but it is recommended to setup the URL trigger only with AWS Identity and Access Management (IAM) authentication.

Each Lambda function is limited by the predefined timeout of each function. If the function is time-consuming, we might consider raising the timeout limit from anywhere between 1 second and 15 minutes. However, bear in mind that the pricing model of the Lambda function relies on billing the execution time. All our methods are configured with defualt one minute timeouts, except for `packetaStatusUpdate`. Due to the large number of Packet packages and the latency of their API responses, we had to extend the timeout to 2 minutes and 30 seconds. This should be enough because, during the mornings, when the Packeta API tends to be a bit slower, we can update about 600 parcels in approximately 110 seconds. As soon as the maximum possible time is no longer sufficient, it will be necessary to reconsider the design of the method and decompose it, for example, into Lambda step functions or to use the predefined queue logic used, for example, by the Česká Pošta shipment update.

D.2.2 Database

Database on the AWS is deployed in two instances - staging and production. Both databases are instances of *Amazon RDS* service setup with the PostgreSQL engine. The chosen instance type is `db.t4g.small` `https://aws.amazon.com/rds/instance-types/` which was chosen since after testing the `db.t4g.micro` could not handle a load of 100 sequential Shipment inserts with related objects (to form a complete object) at the same time.

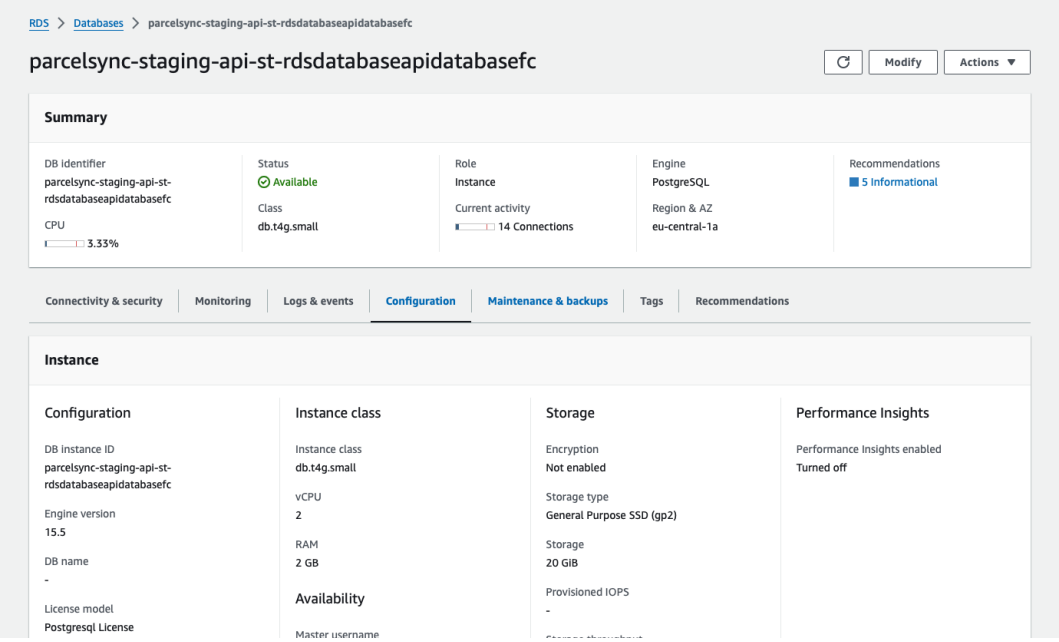


Figure D.5: AWS RDS instance detail

D.2.2.1 Accessing credentials

In order to log into the database from a desktop viewer, it is necessary to obtain the credentials from the console. The database credentials of *Amazon RDS* within the AWS console can be obtained in the following way:

1. First, log into the AWS console.
2. From the *Services* menu select *RDS*.
3. From the database list, select either the staging or production database.
4. In the *Connectivity & security* tab can be obtained **Endpoint** and **Port** (see Figure D.5.
5. In the *Configuration* tab, we can see **Master username**.
6. To obtain the password, from the *Services* menu select *System Manager*.
7. In the *Parameter Store* section search for `DB_PASSWORD` and choose the expected environment.
8. In detail, we can show the decrypted value of the password.

D.2.2.1.1 Sequential inserts, database pool

If there are many sequential inserts at the same time from the public API, there is a small chance that all connection slots will be used.

This happened during the testing, and the following was performed:

1. Updated `knex` library used for database connection.
2. Modified database connection reference to singleton in the lambda API handler.
3. Upgraded the database instance type to `db.t4g.small`.
4. Changed `max_connections` parameter in the PostgreSQL database to 1500.

D.2.3 S3

S3 buckets are used to host static exports of frontend applications which are:

- User documentation
- Tracking page
- Dashboard

Each of these buckets is served through the *AWS CloudFront* distributions that handle routing. This *CloudFront* distribution used for the user documentation had to be manually adjusted, more about that in Section D.2.3.1.

However, dedicated S3 buckets are used to store static assets, for example user-uploaded images for the custom layout of the tracking page and notification emails. More on that later in Section D.2.3.2.

D.2.3.1 Documentation deployment (locale redirection)

As mentioned previously, each bucket with direct routing from Route 53 uses the *AWS CloudFront* distribution. Due to the nature of the user documentation or, more accurately, the routing of non-primary languages in a given application, it was necessary to adjust the *CloudFront* distribution to ensure that other locales can be served without throwing error 404. Because, the Czech documentation is served on a non-prefixed path in URI such as `/docs/welcome`, English documentation is served on prefixed path like `/en/docs/welcome`.

This meant creating a simple function within the *AWS CloudFront* that (in the background) attaches the `index.html` file to the end of each URI.

1. In the *AWS CloudFront* console select the *Functions* section.
2. Now, if not created, create a new function as shown in Listings 2 named `indexhtml-appender` with JavaScript runtime.

```
1 function handler(event) {
2     var request = event.request;
3     var uri = request.uri;
4
5     if (uri.endsWith('/')) {
```

```
6           request.uri += 'index.html';
7       } else if (!uri.includes('.')) {
8           request.uri += '/index.html';
9       }
10
11      return request;
12  }
```

Listing D.2: AWS CloudFront function to append `index.html` to each URI

3. Publish the function and associate it with distribution of documentation.

This function will ensure that the requests of other languages other than the primary one are resolved correctly from the S3 bucket directories.

D.2.3.2 Setting up permissions for assets storage (enable ACLs)

S3 used for storing static assets (uploaded from frontend via backend) require manual adjustments in the permission settings and object ownership after being freshly deployed.

1. Navigate to the newly deployed AWS S3 bucket used to store public assets.
2. Go to the *Permissions* tab and click on **Edit** button in section *Block public access (bucket settings)*
3. Here uncheck all the check-boxes listed (*Block all public access* and save changes as in the Figure D.6.
4. Now within the *Permissions* tab, scroll down to *Object Ownership* and click the **Edit** button.
5. Here select **ACLs enabled** and in the *Bucket Ownership* section select **Bucket owner preferred**. Your setting should look like in Figure D.7 and click *Save changes*.

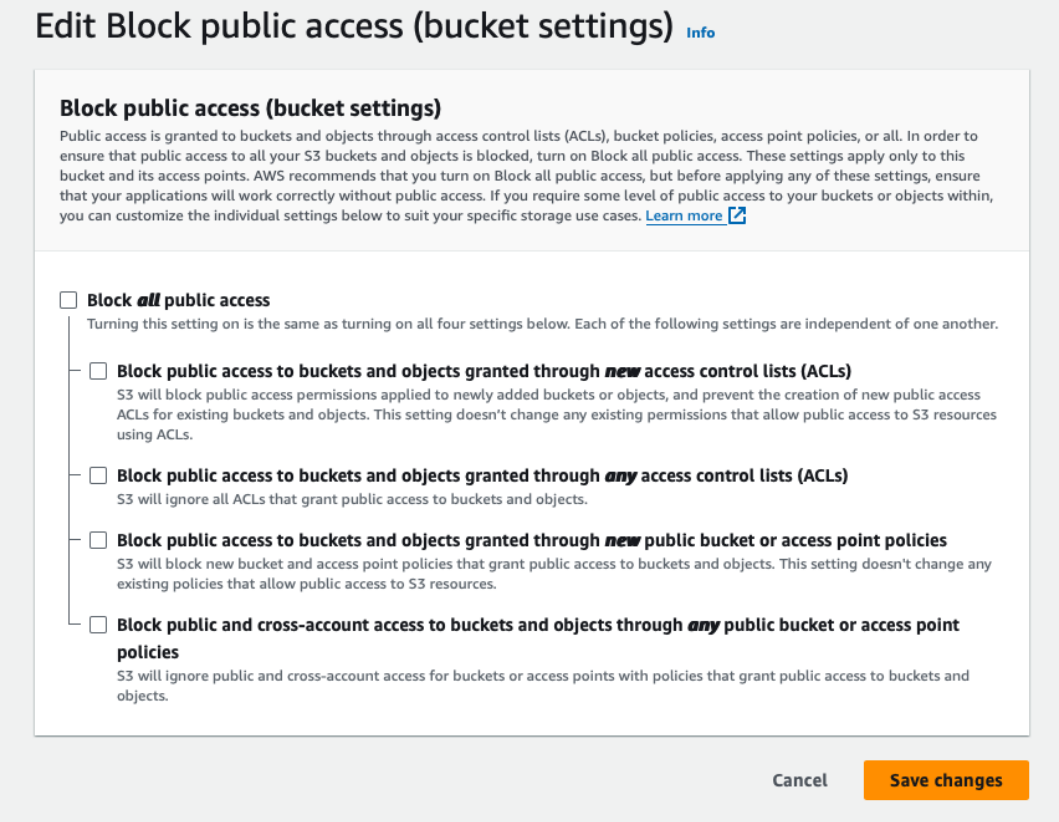


Figure D.6: AWS S3 bucket Block public access settings

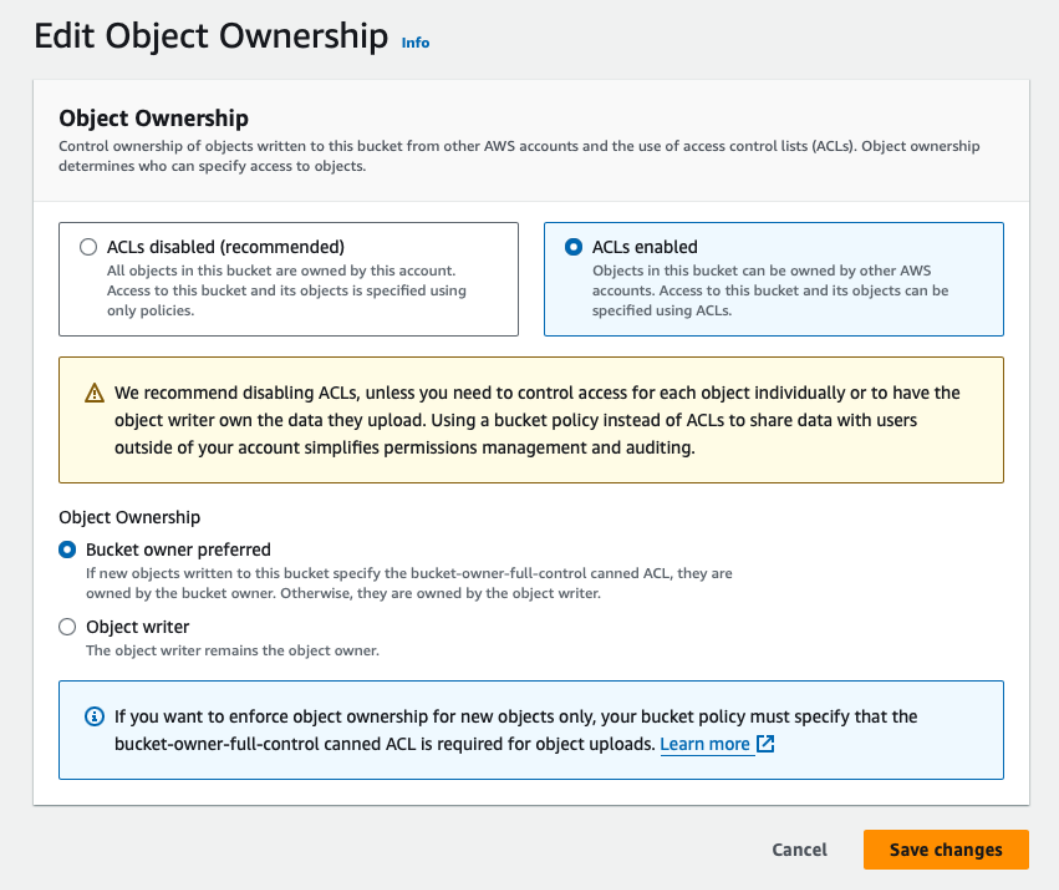


Figure D.7: AWS S3 bucket object ownership settings

D.2.4 Email sender

As a email sender service *AWS Simple Email Service (SES)* is used. The account, where the platform is hosted, was moved from the *AWS SES* sandbox. This process took approximately 2 days, and email examples and communication style had to be presented to the AWS support. Now the account has a sending quota of 50 000 messages per day with a maximum send rate of 14 messages per second.

D.2.5 Deployment to the new AWS account

Thanks to the complex IaC setup within the platform, deployment to the Amazon Web Services is straight forward. As a prerequisite, you need to have platform repository localy, running Node.js $\geq$ 16 and have packages installed using `yarn install` command. Create a new clean AWS account using instructions how to create AWS account with MFA authentication. Then follow the instructions how to create IAM user and use this configuration:

- Set password for the IAM user.
- Generate Access Key and Secret to allow programatic access.
- In the ”Set permissions” section, click on ”Attach existing policies directly” and select `AdministratorAccess`.

- Skip the ”Add tags”

Next time you log in to the AWS console, it is recommended to use created IAM.

Now create entry in the `/.aws/config` file and append following configuration to the file:

```
1 [account-name]
2 region=eu-central-1
3 output=json
```

Listing D.3: `/.aws/config`

Now open `/.aws/credentials` and using values generated while creating IAM user append the following:

```
1 [account-name]
2 aws_access_key_id=<ACCES_KEY>
3 aws_secret_access_key=<SECRET_KEY>
```

Listing D.4: `/.aws/credentials`

In order to test the credentials in the CLI, run following command:

```
1 AWS_PROFILE=account-name aws sts get-caller-identity
```

Listing D.5: Command to test AWS identity

This should output JSON with information about the IAM user.

Now, with everything setup locally, it is time to set up the AWS infrastructure. We will need to setup IAM roles to allow GitHub actions to deploy resources and setup a hosted zone in the Route53 and verify Amazon SES e-mail. Run the following command:

```
1 AWS_PROFILE=account-name yarn setup:aws
```

Listing D.6: Command to setup AWS infrastructure

Once the task if finished, it will output four name servers. Add this list of name servers to the domain NS records and wait until the name servers are propagated.

```
1 AWS_PROFILE=account-name yarn deploy:aws
```

Listing D.7: Command to deploy to the AWS

With the services deployed, we need to deploy our applications in the S3 subsets. For this, it is best to trigger the GitHub deployment action. Whether in the GitHub repository or by pushing some code to the `main` branch.

Now continue with Locale redirection on the documentation described in Section D.2.3.1, enabling ACLs for assets storage as in Section D.2.3.2, and requesting for the SES production access described in Section D.2.4.

E. Administration Manual - SAP Business One ServiceLayer Proxy with Database Connector

This manual outlines the deployment and administration procedures for the SAP Business One ServiceLayer Proxy integrated with a database connector. Details how to establish and manage the service within a Docker environment, using Nginx as a reverse proxy and Certbot for SSL certificate management. The following sections will guide you through the necessary steps to configure and maintain the system effectively, ensuring secure and optimal operation.

E.1 Prerequisites

Before proceeding with the installation of SAP Business One ServiceLayer Proxy, it is important to ensure that the deployment node meets the necessary requirements. Both the SAP ServiceLayer service and the Microsoft SQL database should be reachable from the server network if not exposed publicly. However, it is worth noting that, in order to minimise communication latency, it is good to keep the servers geographically and network-wise as close as possible.

Please, note that the deployment node of the Proxy should be reachable from public network. Ideally, there should be a domain name routed to the server.

The minimal system requirements of the service come primarily from the requirements of the Docker environment. For more information on Docker installation and requirements, please refer to the official Docker documentation `https://docs.docker.com/`. For the containers Ngninx and Certbot, we will ideally need at least 5GB of free space.

E.2 Deployment

The deployment node is a VPS hosted locally within the company. The server is running Ubuntu Server 22.04.3 with Docker version 24.0.7.

E.2.1 PostgreSQL database

Utilised PostgreSQL database container is the official image of PostgreSQL version 16.0 (`postgres:16.0`).

E.2.2 Proxy Service

The service container is automatically built as a private image on Docker Hub `michalpulpan/milpex-sap-api:latest`.

E.2.2.1 Environment variables

To ensure both SAP ServiceLayer and MS SQL connections, it is necessary to run the service with a set of environment variables located in `/srv/sap-api/env`

```
DB_HOST=postgres_host
DB_PORT=postgres_port
DB_USERNAME=postgres_username
DB_PASSWORD=postgres_password

APP_DOMAIN=production_platform_url

SAP_SERVICE_LAYER_URL=https://SAP_URL:PORT/b1s/v1
SAP_SERVICE_LAYER_PROD_USERNAME=sap_prod_username
SAP_SERVICE_LAYER_PROD_PASSWORD=sap_prod_password
SAP_SERVICE_LAYER_PROD_DB=sap_prod_db
SAP_SERVICE_LAYER_DEV_USERNAME=sap_dev_username
SAP_SERVICE_LAYER_DEV_PASSWORD=sap_dev_password
SAP_SERVICE_LAYER_DEV_DB=SBO-sap_dev_db

MSSQL_DEV_DB=mssql_dev_db
MSSQL_DEV_DB_USERNAME=mssql_dev_username
MSSQL_DEV_DB_PASSWORD=mssql_dev_password
MSSQL_DEV_DB_HOST=mssql_dev_host
MSSQL_DEV_DB_PORT=mssql_dev_post

MSSQL_PROD_DB=mssql_prod_db
MSSQL_PROD_DB_USERNAME=mssql_prod_username
MSSQL_PROD_DB_PASSWORD=mssql_prod_password
MSSQL_PROD_DB_HOST=mssql_prod_host
MSSQL_PROD_DB_PORT=mssql_prod_port
```

Listing E.1: SAP Business One ServiceLayer Proxy with database connector environment variables setup

E.3 Docker Compose

In order to orchestrate and simplify the deployment process of both services, it is recommended to use the Docker Compose tool. The `docker-compose.yml` configuration on the deployment node F.2 is specified in `/srv/sap-api`. The example `docker-compose.yml` is:

```
version: '3.9'

services:
  api:
    container_name: sapb1-middleware-api
    image: michalpulpan/milpex-sap-api:latest
    ports:
      - '3333:3000'
    depends_on:
      - database
    env_file:
      - .env
    profiles:
      - prod
```

```
    restart: always
    logging:
      driver: json-file
      options:
        max-size: "5m"
        max-file: "10"
  database:
    container_name: sapb1-middleware-postgres-database
    command: postgres -c 'max_connections=300'
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: postgres
    restart: always
    image: postgres:16.0
    volumes:
      - database_volume:/var/lib/postgresql/data:Z
    ports:
      - '5432:5432'
    profiles:
      - prod
volumes:
  database_volume:
    name: database_volume
    external: true
```

Listing E.2: SAP Business One ServiceLayer Proxy with database connector `docker-compose.yml`

E.3.1 `Watchtower`

`Watchtower` is used to pull and rerun newly build images on the Docker Hub. The `Watchtower` is ran as a docker container from a Compose file specified in `/srv/monitoring/docker-compose.yml`.

```
version: '3.9'
services:
  watchtower:
    image: containrrr/watchtower
    restart: always
    env_file:
      - watchtower/.env
    volumes:
      - /home/user/.docker/config.json:/config.json
      - /var/run/docker.sock:/var/run/docker.sock
```

Listing E.3: Watchtower `docker-compose.yml`

`Watchtower` is defined to automatically send Slack notifications using notification library named `shoutrrr https://containrrr.dev/projects/shoutrrr/` module when new version is pulled. The Slack notification is defined in the `.env` file located in `/srv/monitoring/watchtower/`:

```
WATCHTOWER_LABEL_ENABLE=1
WATCHTOWER_NOTIFICATIONS="shoutrrr"
WATCHTOWER_NOTIFICATION_URL="slack://token:token@channel/"
```

Listing E.4: Watchtower environment variables

In order to define other communication channel, please refer to the `shoutrrr` documentation `https://containrrr.dev/projects/shoutrrr/`

E.3.2 Working with the containers

To start/stop or update containers, it is recommended to use Docker Compose.

E.3.2.1 Start the containers

In order to start the containers in background, go to the folder `/srv/sap-api` and run `docker compose --profile prod up -d`

E.3.2.2 Stop the containers

In order to stop the containers in background, go to the folder `/srv/sap-api` and run `docker compose stop`

E.3.2.3 Update the service

In order to update the API service, go to the folder `/srv/sap-api` and run `docker compose pull api && docker compose up --profile prod -d`. The specified command will pull the latest container from the Docker Hub and start the services again.

However, note that containers should update automatically if there is a new image.

E.4 Reverse-proxy

A reverse-proxy used for deployment is Nginx version 1.18.0. Since there is a domain routed directly onto the server, we can use Nginx for internal routing as a reverse-proxy. The standard Nginx definition is found by convention in `sites-available` within the Nginx specific folder `/etc/nginx`. The route is then symlinked to the `/etc/nginx/sites-enabled` folder using the following command:

```
1 sudo ln -s /etc/nginx/sites-available/domain /etc/nginx/sites-
    enabled/
```

Listing E.5: Command to create symbolic link by Nginx convention

E.4.1 SSL certificate

For automatically SSL certificate renewal from Let's Encrypt, `Certbot` was installed and configured. To obtain a certificate for the newly created route, run:

```
1 sudo certbot --nginx -d example.com -d www.example.com
```

Listing E.6: Certbot command to obtain SSL certificate

F. Administration Manual - Data-sender

In this manual will be described the deployment process of the Data-sender between the platform and SAP ServiceLayer Proxy. The deployment process contains running the service in a Docker environment in scheduled mode.

F.1 Prerequisites

Before proceeding with the installation of Data-sender, it is important to ensure that the deployment node meets the necessary requirements. Both the public API of the platform and the SAP ServiceLayer Proxy should be accessible from the server network if not exposed publicly. However, it is worth noting that, in order to minimise communication latency, it is good to keep the servers geographically and network-wise as close as possible.

The minimal system requirements of the service come primarily from the requirements of the Docker environment. For more information on Docker installation and requirements, please refer to the official Docker documentation https://docs.docker.com/.

F.2 Deployment

The deployment node is a VPS hosted locally within the company. The server is running Ubuntu Server 22.04.3 with Docker version 24.0.7.

F.2.1 Data-sender service

The service container is automatically built as a private image in the Docker Hub michalpulpan/milpex-sap-data-sender:latest.

F.2.1.1 Environment variables

To ensure the connection to the SAP ServiceLayer Proxy and the platform, it is necessary to run the service with a set of environment variables located in srvsap-api/.data_sender.env in Listings F.2.1.1.

```
ENVIRONMENT=production

SAP_PROXY_BASE_URL=https://api.company.com
SAP_PROXY_USERNAME=username
SAP_PROXY_PASSWORD=password


PARCELSYNC_BASE_URL=https://api.parcelsync.io
PARCELSYNC_API_KEY=key
```

Listing F.1: Data sender environment configuration

F.3 Docker Compose

In order to orchestrate and simplify the deployment process of both services, it is recommended to use the Docker Compose tool. The `docker-compose.yml` configuration on the deployment node F.2 is specified in `/srv/sap-api/` directory.

The example `docker-compose.yml` is in Listings F.3.

```
version: '3.9'

services:
  data-sender:
    container_name: sapb1-data-sender
    image: michalpulpan/milpex-sap-data-sender:latest
    env_file:
      - .data_sender.env
    depends_on:
      - api
    restart: always
    profiles:
      - prod
    logging:
      driver: json-file
      options:
        max-size: "5m"
        max-file: "10"
```

Listing F.2: Data-sender `docker-compose.yml`

F.3.1 Watchtower

`Watchtower` is used to pull and rerun newly build images on the Docker Hub. The `Watchtower` is ran as a docker container from a Compose file specified in `/srv/monitoring/docker-compose.yml` in Listings F.3.1.

```
version: '3.9'
services:
  watchtower:
    image: containrrr/watchtower
    restart: always
    env_file:
      - watchtower/.env
    volumes:
      - /home/user/.docker/config.json:/config.json
      - /var/run/docker.sock:/var/run/docker.sock
```

Listing F.3: Watchtower `docker-compose.yml`

`Watchtower` is defined to automatically send Slack notifications using notification library named `shoutrrr https://containrrr.dev/projects/shoutrrr/` module when new version is pulled. The Slack notification is defined in the `.env` file located in `/srv/monitoring/watchtower/`:

```
WATCHTOWER_LABEL_ENABLE=1
WATCHTOWER_NOTIFICATIONS="shoutrrr"
WATCHTOWER_NOTIFICATION_URL="slack://token:token@channel/"
```

Listing F.4: Watchtower environment variables

In order to define other communication channel, please refer to the `shoutrrr` documentation `https://containrrr.dev/projects/shoutrrr/`

F.3.2 Working with the containers

To start/stop or update containers, it is recommended to use Docker Compose.

F.3.2.1 Start the containers

In order to start the containers in background, go to the folder `/srv/sap-api` and run `docker compose --profile prod up -d`

F.3.2.2 Stop the containers

In order to stop the containers in background, go to the folder `/srv/sap-api` and run `docker compose stop`

F.3.2.3 Update the service

In order to update the API service, go to the folder `/srv/sap-api` and run `docker compose pull api && docker compose up --profile prod -d`. The specified command will pull the latest container from the Docker Hub and start the services again. Note that containers should update automatically within few minutes if there is a new image.

G. User documentation - Platform (Dashboard)

This guide designed to help users navigate and use the dashboard and API effectively is located at `https://help.parcelsync.io`. Covering the functions and features integrated into the dashboard, providing step-by-step instructions and helpful tips to ensure that users can fully leverage the tools available to them.