BACHELOR THESIS

Filip Štrobl

Data logging and visualization for Mailtrain using IVIS

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	prof. RNDr. Tomáš Bureš, Ph.D.
Study programme:	Computer Science
Study branch:	Programming and software development

Prague 2023

I would like to express my gratitude to prof. RNDr. Tomáš Bureš, Ph.D. for supervising, providing advice, and having patience with me for the long, long time it took me to finally finish this thesis.

Title: Data logging and visualization for Mailtrain using IVIS

Author: Filip Štrobl

Department: Department of Distributed and Dependable Systems

Supervisor: prof. RNDr. Tomáš Bureš, Ph.D., Department of Distributed and Dependable Systems

Abstract: Mailtrain is a self-hosted, free and open-source newsletter application with advanced options for managing lists of subscribers, creating and sending e-mail campaigns, and managing multiple users with granular permissions and flexible sharing. The application lacks good options for analyzing and visualizing its data for the purposes of tracking performance or security. IVIS is a framework offering the data processing and visualizing tools Mailtrain needs, and the two projects share many key technologies. In this thesis, we extend Mailtrain so that it uses IVIS and its services for logging, visualization, and analysis of its data. An emphasis is given to the extensibility of both the data logged from Mailtrain and the ways it is visualized.

Keywords: newsletter data analysis visualization

Contents

1. Introduction

Mailtrain [15] is a self-hosted, free and open-source newsletter application, which is, at the time of writing, actively used in production. Its features include managing lists of subscribers, including their possible segmentation, and sending newsletter campaigns to them, either manually, or automatically through triggered or RSS campaigns. The application supports custom e-mail campaign templates, including MJML-based templates, and is relatively easy to setup using Docker and built-in Zone-MTA. It is well suited for larger and enterprise-level solutions, due to its support of multiple users with granular user permissions and flexible sharing, as well as its system of hierarchical namespaces.

However, the application lacks good features for logging, analyzing, and visualizing data of activity and performance of various entities, either for the purpose of analyzing performance of lists or campaigns or for auditing user activity to be able to detect possible mistakes and attacks. There exist some implementations in Mailtrain for this purpose, namely the campaign statistics, displaying various metrics of a campaign, and reports, which are custom data collection tasks programmed by users of Mailtrain. However, these features proved to be insufficient in terms of their convenience or flexibility, so a better solution is needed.

To make the processing and visualization of data easier, one could use an existing framework designed for this. The IVIS framework [19] is one such framework, capable both of processing data using its tasks and of visualizing data using its templates. Unlike most similar frameworks, IVIS focuses on flexibility, by making its visualizations and data processing jobs fully programmable through JavaScript and Python. The framework also provides various extensions to help its integration into other applications, like embedding visualizations from its web interface into other web pages.

These features alone would make IVIS a good candidate, but Mailtrain and IVIS also share many core technologies and principles, making IVIS naturally compatible and well integratable into Mailtrain. That said, the way in which IVIS' functionality needs to be integrated into Mailtrain has not been done before and certain IVIS features needed for it are yet to be fully implemented.

The main goal of this thesis is to create a system for logging and visualizing data in Mailtrain using IVIS. This entails:

1. making a system for logging data in Mailtrain which sends the data to IVIS,
2. designing the appropriate structure for the logged data in IVIS, as well as the visualizations (either by using existing ones or programming new ones),
3. making a system for displaying IVIS' visualizations of the logged data from the Mailtrain application.

The bulk of the work will be done on the code in Mailtrain's GitHub repository. Some new features will also have to be added into IVIS' repository, but care will be put into making the new features flexible enough for general use, and not just to serve Mailtrain.

The next chapter describes the existing projects to familiarize the reader with them before any deeper analysis. Mailtrain is described first in section 2.1, and

IVIS is described second in section 2.2. In both cases, the sections are split into an overview of the concepts, and an overview of the technologies used by the project. In Mailtrain's case, the overview of concepts is especially important, as the logged events to be implemented may encompass most of Mailtrain's functionality.

After the background, the Analysis chapter further specifies the requirements of this work. The aim is to provide a description of events to be logged and the accompanying visualizations, without implementing them yet. The first section 3.1 describes all events in Mailtrain's existing functionality, for which logging to IVIS should be implemented. The proposed events to be logged include complete descriptions of the data they should log. The second section 3.2 describes transformations of data in IVIS to make querying data easier. The final section 3.3 proposes visualizations for the logged data. Each visualization proposal contains a description of the logged data the visualization uses and its intended visual representation.

The Solution Architecture chapter describes the architecture of Mailtrain's and IVIS' integration. In section 4.1, the integration is first described as a whole, and problems such as communication between the two projects are addressed. The next two sections, 4.2 and 4.3, describe the architectures of the extensions from Mailtrain's and IVIS' side, which includes any new modules and their responsibilities. The final section 4.4 is dedicated to describing how the integration handles user management and authorization.

The Implementation chapter explains the technical details of the integration's implementation, structured by the newly implemented modules. Section 5.1 explains extensions done to the stand-alone IVIS project, which is still independent of Mailtrain. Sections 5.2, 5.3, 5.4, and 5.5 describe extensions done to the main runtime code in Mailtrain, including Mailtrain-specific extensions to IVIS. The final section 5.6 briefly describes changes done outside the main runtime code of the project. This includes things such as changes of code for the application's setup, modifications of the git repository, and changes to any documentation or configuration files.

The results of the work done in the thesis will be shown in the Evaluation chapter. The first section of this chapter, section 6.1, shows how the implementation was tested to ensure that it works correctly. Section 6.2 then shows how the resulting visualizations look by providing screenshots of them from the application using testing data.

2. Background

Before delving further into the work of the thesis, it's important to understand the main concepts and workings of both Mailtrain and the IVIS framework. To determine what data should be logged in Mailtrain and what visualizations it could use, we need quite a thorough understanding of Mailtrain's concepts. Similarly, if we are to use IVIS as a framework for logging and visualization of data, we should also understand its concepts, albeit not necessarily as thoroughly as Mailtrain. To successfully integrate the two projects, we also need to know their architectures and the technologies they use.

This chapter describes the concepts and architectures of both Mailtrain and IVIS. The descriptions cover every important concept and technology relevant to the goals of this thesis. Relevant details concerning existing implementation are explained in later chapters, where they are used for analysis or implementation.

2.1 Mailtrain

The most important piece of software to this thesis is Mailtrain [15]. As stated in the introduction, Mailtrain is a self-hosted newsletter application, which is fully free and open source, using the GPL-V3.0 license [6]. This section describes Mailtrain in more detail.

Mailtrain's main purpose is to help users create and manage newsletters. It helps by automatically managing subscribers in lists, including, for example, providing subscription forms to sign up, Messages are sent to subscribers using campaigns, which are created by users inside Mailtrain, possibly with the help of its various campaign content editors. The content of campaigns can also be defined more generally using templates, where multiple campaigns can use a single template. Campaigns may be sent manually to all subscribers of a list, or to only a segment of a list, consisting of subscribers who satisfy custom user-made rules. Campaigns may also be configured to be sent to a given subscriber automatically when a custom trigger is activated for the subscriber.

A Mailtrain instance can have multiple lists, campaigns, etc. An umbrella term for these objects is *entities*. Mailtrain supports having multiple users and allows managing access to entities by different users. It achieves this with the help of a hierarchical system of namespaces containing all of Mailtrain's entities. Users can share entities, or entire namespaces of entities, with other users. This makes Mailtrain capable of managing large, enterprise-level environments of many users with different permissions.

2.1.1 Concepts

Most of Mailtrain's functionality happens through entities. Due to this design, Mailtrain's concepts align quite well with its entity types, so many concepts in this section are described in terms of the corresponding entity type along with its responsibilities.

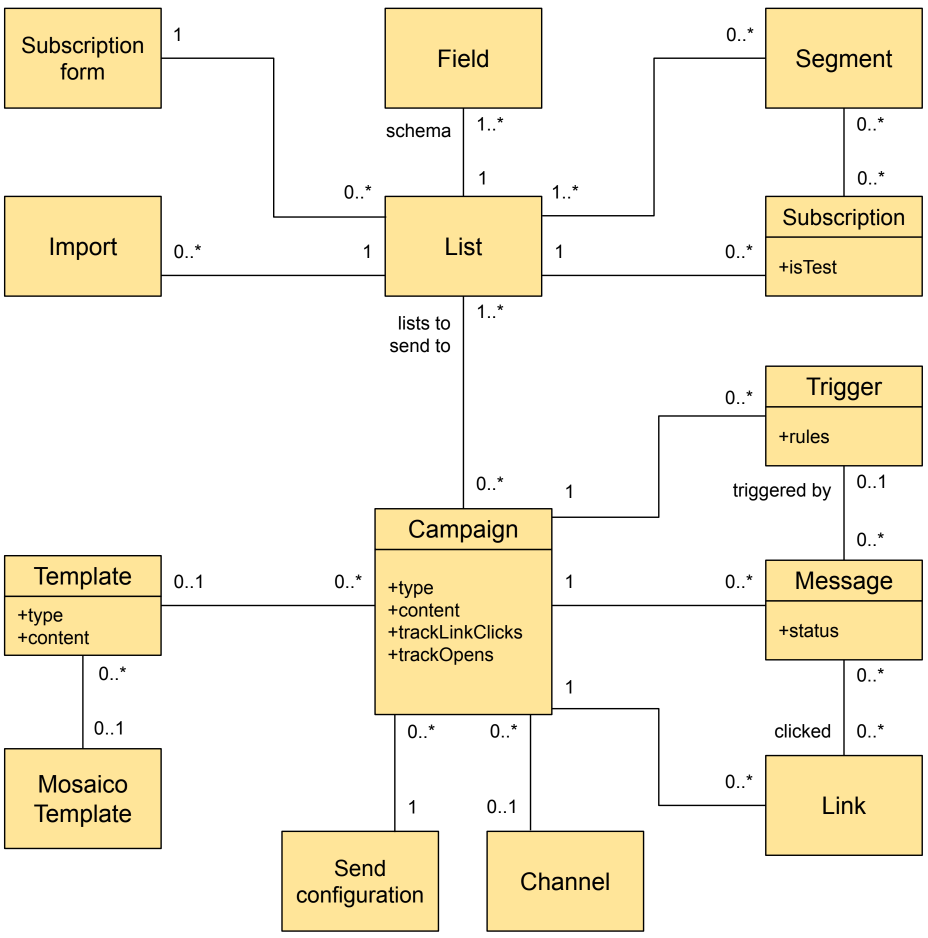


Figure 2.1: A class diagram of Mailtrain's most important entities

List

The central entity type in Mailtrain is a list, which is used to manage subscribers. A subscription form is another entity type, which allows custom content of a list's subscription form. Filling out subscription forms is the main way new subscribers are added to a list. The other way to add subscribers (apart from doing it by hand) is using imports. Imports are tasks that add all subscribers from some stored source (e.g. a CSV file) into a list. This may take a long time, so the imports are stored as entities and their progress is tracked. If extra information about a list's subscribers is needed, Mailtrain supports adding custom fields to a list, which is analogous to adding an extra column in a database table, where rows represent subscribers. Lists can contain segments, which include a portion of the list's subscribers satisfying a given rule set. Subscribers in a segment can then be treated separately, e.g. by sending an e-mail campaign only to subscribers in a given segment. Some subscribers of a list can also be marked as test users, which then allows them to receive testing messages.

Campaign

The second major entity type in Mailtrain is a campaign. Campaigns represent an e-mail with custom content and possibly attached files (either through e-mail or publicly accessible over HTTP) that gets sent to subscribers of a given list, or multiple lists. There are currently 3 types of campaigns: *regular*, which is sent to all target subscribers when launched, *RSS*, which is similar to a regular campaign, but is intended to be viewed through an RSS feed, and *triggered*, which is sent to a subscriber when they satisfy some given condition (a trigger). Mailtrain provides several editors for editing campaigns' content, such as the GrapesJS template designer, or, more importantly, the Mosaico template designer. Mailtrain also allows for the creation of templates, which multiple campaigns can use instead of making their content from scratch every time. Mosaico itself uses templates to customize the template editor, so Mailtrain also includes Mosaico templates as entities, distinct from regular templates. Both templates and Mosaico templates can also include files in their content. The user may send a test message of either a campaign or a template to one or several test users of a given list. Finally, a campaign may track subscribers opening e-mails or clicking on links in the campaign's content, but this can be opted out of in the campaign settings.

After a campaign is sent, all its sent messages are tracked. The bare minimum of tracked data is how many messages were sent or have failed to be sent, how many sent messages bounced back or were reported as spam, and which subscribers unsubscribed because of this campaign. If link click tracking or open tracking is enabled, Mailtrain also tracks the subscribers that opened the tracked message or the subscribers that clicked any of the message's links. In the case of a triggered campaign, its triggers may also be logged (i.e. which triggers have been triggered for which subscribers of the campaign's target lists).

The way campaigns are sent can be configured using send configurations. Send configurations also specify other mailer settings, such as mailer type or throttling, as well as bounce handling using a variable envelope return path (VERP). Different campaigns may use different send configurations.

Channel

To better organize campaigns and simplify campaign creation, Mailtrain supports channels of campaigns. Channels can specify some default parameters of their campaigns. Each campaign may belong to at most one channel.

Report

The next entity type is a report. Reports are tasks for analyzing and displaying data in Mailtrain. Reports are created first as report templates defining code with parameters and reports themselves then apply concrete parameters to the templates to use them. Eventually, user-made reports will likely be realized through IVIS, and these original ones will be removed. This is why they are not that important from the perspective of running the application.

Users, Namespaces and Entity Sharing

The two remaining entity types are used for administration: namespaces and users. Every entity described above belongs to a namespace, with the only exception being the root namespace. Based on role assignment of entities or namespaces, with roles defined in a configuration file, users have certain permissions to access other entities (including namespaces), either in their own namespace, or ones explicitly shared with them by other users (by assigning them a role).

Concepts Not Tied To Entities

There are several concepts left to describe which are not tied to entities and are instead global in nature. To avoid troublesome subscribers, Mailtrain includes a global blacklist to block given e-mail addresses. Mailtrain also has global settings, which can be edited by users with given global permissions. Finally, the application includes a user API, which can allow user-made programs to interact with Mailtrain.

2.1.2 Architecture

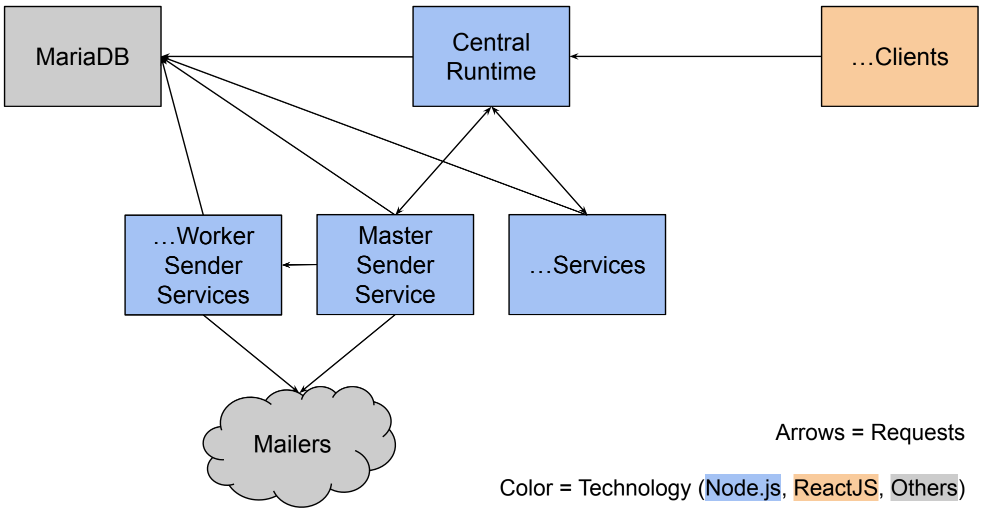


Figure 2.2: Runtime view of Mailtrain's architecture

Like the majority of other web applications, Mailtrain's architecture is split into a frontend and a backend. Both are written in the ES6 standard of JavaScript.

Frontend

The client-side frontend runs in the web browser and forms the main interface for users' communication with Mailtrain. The main library used by the frontend is React [13], which utilizes the concept of Components assembled like HTML using JSX to create webpages entirely using JavaScript. Another notable library is Webpack [4], which bundles the code together for deployment.

To avoid cross-site scripting attacks and guarantee security, Mailtrain creates 3 URL endpoints that can be communicated with. The first is the *trusted* endpoint, which is the main endpoint for Mailtrain's UI when a user is logged in. The second is the *sandbox* endpoint, which is used to host template editors. The third one is the *public* endpoint, which is used to host content for subscribers, e.g. subscription management forms or files linked to an e-mail.

Backend

The backend runs on a server using the Ubuntu [14] or CentOS [17] operating system, or is deployed in a Docker [12] container. It is responsible for managing a database of application data, controlling the application logic, and communication with users and subscribers, which entails not only communication with the frontend, but also with e-mail servers of subscribers, and an API to be optionally used by users' own applications.

Most of Mailtrain's backend code runs within Node.js [8]. From a runtime perspective, the code is split by functionality into multiple Node.js processes. The central backend runtime component in Mailtrain is responsible for controlling all main actions in the application, listening for requests from clients, and communicating with the other Node.js processes. The other processes, which Mailtrain calls *services*, are usually responsible for doing computationally difficult tasks. One service in particular - the sender service, is responsible for sending campaigns, even spawns its own worker processes to make sending large campaigns possible.

Mailtrain uses MariaDB [7] as an SQL database, and communicates with it with the help of the Knex library [3]. Communication between any parent Node.js process and its service is done using the Child process [9] module from Node.js standard library, which is obtained when the service process is spawned. Communication with clients is realized using a RESTful HTTP API. The sender processes communicate with the intended mailers, to be able to send messages. There is built-in support for using Zone-MTA [5] as a mailer, but using generic SMTP is also supported, as well as Amazon SES [11].

2.2 IVIS Framework

IVIS [19] is the software we shall use to extend Mailtrain to make it capable of displaying visualizations of its data. This section describes the framework in more detail.

The IVIS framework is used for processing and visualizing data from various data sources. Its main features are flexibility and the ability to extend other domain-specific applications. The framework is sometimes advertised as a framework for data generated by Internet of Things (IoT) and Cyber-Physical Systems (CPS) environments, but its focus on flexibility allows it to accommodate the needs of Mailtrain well enough, despite Mailtrain not necessarily belonging to either of those categories. There exist multiple variants of the framework, the one this thesis uses and refers to as "IVIS" is known as IVIS Core.

2.2.1 Concepts

This subsection describes IVIS' concepts in more detail. If in need of more information, the IVIS Core GitHub repository also contains its own description of its concepts [18].

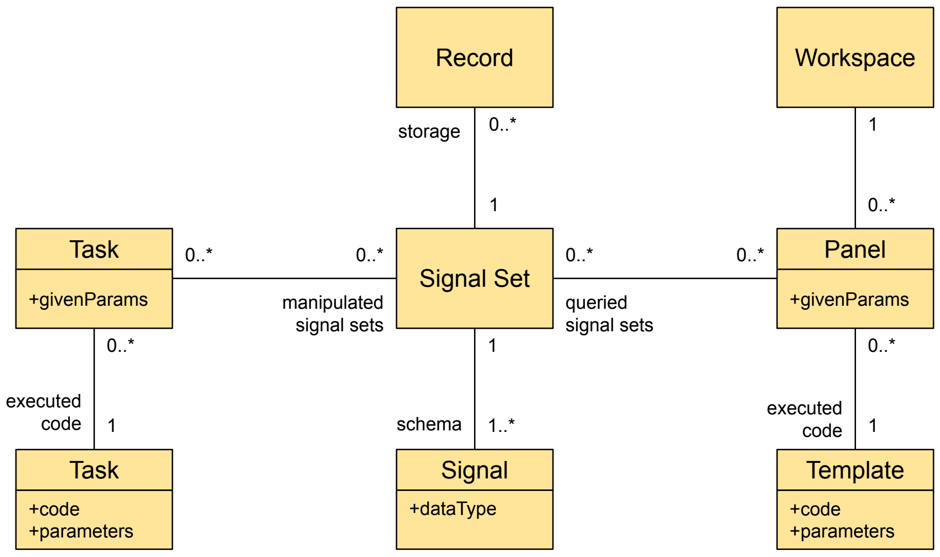


Figure 2.3: A class diagram of IVIS's entities

Data Organization

The data collected by IVIS are organized into a structure of *records* and *signal sets*. To use a typical example for IVIS, if we have a monitoring system of several sensors collecting data at the same time, a signal represents data from a single sensor. Each signal has a data type and must belong to some signal set. A signal set represents a set of one or multiple signals, and a record is a set of values for each signal of some signal set. A good analogy is to think about signal sets as database tables, signals as table columns, and records as table rows. In the typical example, data is gathered from real-life sensors, each providing signals measured in the real world, but IVIS also allows us to insert signal set records without any of the signals corresponding to a real-life sensor.

Data Processing

The data processing in IVIS is handled by *tasks* and *jobs*. A task defines the code to be run, along with any files to use, and task parameters. The code is programmed fully by the user, giving them complete control over the transformation of data. Currently, tasks support Python code with a few additional libraries, those being EnergyPlus, Numpy, and Pandas. A job is an instance of a task with given parameters, e.g. what signal sets it operates on, as well as trigger settings, meaning if the job should activate periodically, or activate when a certain signal set is updated.

Visualizations

A similar level of complete control is seen in IVIS' visualizations, which, unlike most other popular visualization frameworks, are programmed directly from JavaScript code. This method is less accessible than GUI editors, but ultimately more flexible and powerful. To make the process of creating new visualizations easier, IVIS contains pre-implemented common visualization elements in its code, which it calls *charts*. Charts are components of React code that can be used and built upon by user-made visualizations. Instantiating the visualizations' code in IVIS is realized using *templates*. Similarly to IVIS' tasks, templates define the visualization code and parameters. Templates are instantiated using *panels*, which specify the parameter values, e.g. what signal sets to visualize. Panels are organized using *workspaces*. Each panel must belong to a workspace.

As a standalone application, the main method of IVIS' user interface is a web interface for modifying data and viewing visualizations. As an extension of another application, IVIS also provides an option to modify its data through code and even allows embedding visualizations in another web window. The embedding functionality is very powerful, since unlike panels, one can visualize a template directly, without the need to create a panel or a workspace. In that case, the template's panel and workspace are only virtual.

Extending the Framework

To simplify the task of using IVIS as an extension to other applications, the framework provides an extension manager, which can be used to extend the desired functionality using custom code. Extending IVIS this way is done by creating a Node.js program, defining all needed extensions in the code, and then importing the IVIS code. The result is an IVIS core program that has all the required extensions. IVIS allows extending its code at pre-defined points, where each extension point is identified with a keyword. Here are the extensions most important to this thesis (without their keywords):

1. create extra routes, which means either server-side REST call listeners, or client-side web page content providers,
2. prepare extra services after IVIS starts,
3. define built-in templates and tasks, which can be used similarly to normal templates and tasks, but do not need to be stored in a database.

Administration

Finally, IVIS includes tools for administration, which are identical to the ones in Mailtrain, described in section 2.1.1. Every entity (task, job, template, etc.) belongs to a namespace, and IVIS users can share these entities manually, or automatically with their role.

The data stored by IVIS can be shared to the granularity of signals in a signal set, meaning access of users to IVIS' data can be controlled to the point where a user has access to only a subset of signals of each record in a given signal set. There is no way to restrict users' access to signal sets record-wise. If one wants to keep some records accessible and some records not, they have to be stored in separate signal sets.

2.2.2 Architecture

Mailtrain and IVIS are developed largely by the same people, and so the architectures of Mailtrain and IVIS are almost identical. Just like Mailtrain, IVIS is

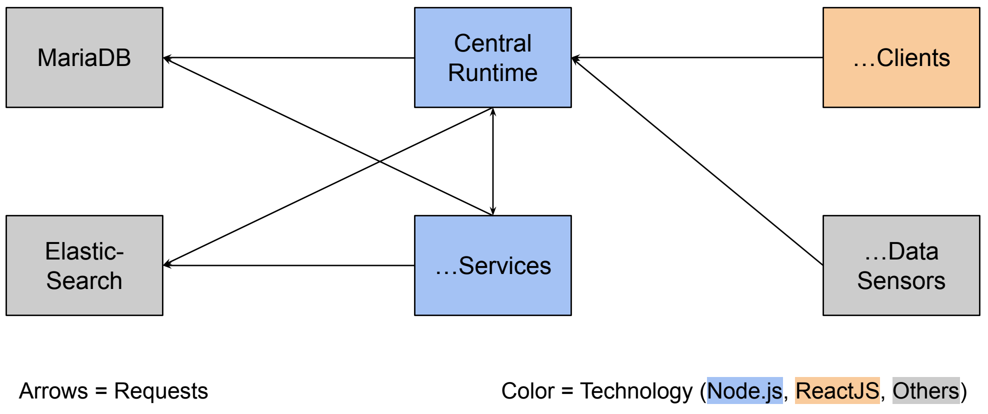


Figure 2.4: Runtime view of IVIS' architecture

also written in ES6 JavaScript and has an architecture split into a frontend and a backend. Since Mailtrain's architecture has been explained in section 2.1.2, this section only explains the architectural differences between the two projects.

Frontend Differences To Mailtrain

The frontend runs in a web browser and uses the React [13] library to build web pages that are then bundled using Webpack [4]. Unlike Mailtrain, IVIS aims to provide potentially complex visualizations, which is a difficult task without a proper library. The libraries IVIS uses to help build its visualizations are the various libraries by D3 (Data-Driven Documents) [1].

Like Mailtrain, IVIS splits access to it into three endpoints. The function of the *trusted* endpoint is identical to Mailtrain. The *sandbox* endpoint is used to host user-defined panels, which may be potentially unsafe. The third endpoint in IVIS is an *API* endpoint, which is meant to be used by applications working with IVIS' data.

Backend Differences To Mailtrain

The backend runs on a server using the Ubuntu [14] or CentOS [17] operating system. Just like Mailtrain, the backend Node.js [8] runtime is split into a central component and services. In the case of IVIS, no service spawns additional subprocesses. Since IVIS does not manage newsletters, it does not use mailers, and its communication with clients is primarily done through its web interface. IVIS however expects to be connected to sources of data, which it calls *sensors*. The sensors supply IVIS with the data they measure so that the data can be processed and visualized. The way the backend communicates with sensors can be configured in the code and is therefore not limited to any single technology.

IVIS also needs a way to quickly analyze its signal set data. For this purpose IVIS indexes its signal set data with Elasticsearch [16]. Elasticsearch is a distributed search engine with a REST API, capable of quick access, analysis, and aggregation of indexed data.

3. Analysis

This chapter analyzes Mailtrain and determines both the events worthy of logging, and the visualizations to be added, which are meant to utilize data mainly from the logged events. The descriptions of logged events include information about all their logged data. The descriptions of visualizations include what data they draw from and how they display it. The proposed events and visualizations also include a rationale behind why they are useful and worthy of being added. However, this chapter will mostly not describe any technical or implementation details, although it may touch on how the data should be organized in order to be queried effectively.

The events to be logged are described first. Described events are split into categories, where events from a single category can be stored together. The descriptions may be split further if there is notable difference between the event types, although this is only to make the description more clear, and does not affect implementation. The description of logged data for each event category is structured as a schema of a relational database table, hinting at how the events may be stored.

Before the section on visualizations, there is a section describing data transformations of the logged events. For some visualizations, querying the logged data as-is would be inefficient. Therefore, a section is dedicated to describing the necessary transformations of data to make using the proposed visualizations practical.

The final section describes the visualizations. This entails describing what data they use and how they display them.

3.1 Mailtrain Events to be Logged

This section describes an analysis of Mailtrain's functionality to find events worthy of logging to IVIS. While the logged data are used mainly by visualizations, logging the events is useful by itself, since if more visualizations will be made in the future, they will be able to use this data as well. Also, even without visualizations, IVIS allows manually viewing records without any custom visualizations, which ensures that logged events are accessible to users.

The described events are split into categories called event types. Logged events of a single event type are semantically close together, meaning they are likely to be queried together, and they usually also share a significant part of their data schemas. For these reasons, each event type describes a schema for logged data, which is shared by all of the logged events of this type. The shared schema allows them to be stored in the same place, and therefore easily queried together.

The schema of each event type is structured as a schema of a relational database table, because for IVIS to utilize logged data effectively, it needs to store the data into signal sets, which are structured this way. Due to this requirement, it is best to format the data like this from the start and avoid having to transform the data during logging. The data may also have rather high redundancy, which is not recommended for standard SQL databases. However, data is queried using ElasticSearch, which is not a standard SQL database, and IVIS itself does not yet have the functionality needed to query ElasticSearch with advanced query operators such as joins, so the redundancy in the schemas is somewhat justified by this.

When analyzing the possible events to be logged, some data were found that could be useful in some situations, but it was decided to not log them, mostly because it would be too inefficient or complicated (e.g. information about entity settings). If any future extensions decide to extend the data that is logged, in many cases new events may not need to be created, and instead it should be enough to simply extend the data logged by some already existing events.

3.1.1 Entity Activity Event Types

A part of the intended functionality this thesis aims to add to Mailtrain is the ability to track activity of the application's entities and global settings. This kind of activity is usually done by users of Mailtrain and include activities such as creations, modifications, and deletions of entities, or changing states of certain entities (e.g. launching a campaign). The goal of logging these activities is to allow administrators to track the activity of the application. Among other things, tracking the application's entity activity is useful to spot any disruptive activity, either as a result of an error or an intentional attack.

The purpose of tracking activity data is shared accross all of these events, which makes a case for assigning the same event type to all of them. However, in case of entity activity, various events tend to differ greatly in their logged data schemas, so storing all of them in a single table or signal set would lead to very complex schemas. Therefore, the entity activity events are further split into events based on entity type. That said, entities of some types always belong to a parent entity (e.g. every list field belongs to a list). The events of these entities are merged with the event type of their parent entity.

For the logged data to be useful, every record of entity activity should contain a timestamp of when the event happened, information about the type of activity that was done (e.g. that a campaign was launched), the ID of the user making this action, if the activity was caused by one (this user is called the *actor* of the activity), and the ID of the affected entity. Certain entity activity events may also log extra data in addition to the three base fields, such as the status of a campaign, but omitting one of the three fundamental fields is very rare, so unless explicitly mentioned, all logged entity activity events are assumed to contain these three fields. The following paragraphs describe the logged events in more detail.

The vast majority of entities in Mailtrain can be manipulated with using three types of actions: *create*, *update*, and *delete*. Since all of these actions represent some events when the entities are manipulated with, they are therefore potentially worthy of logging. For brevity, we shall refer to these events as *CUD*.

If complete information about a change in an entity is needed, then details about the changes of the entity should be logged. However, entities often contain several fields of data, which vary greatly among entities of different types. Logging these details would require either an extra field of the log data for every field of the entity, or have only one field of the logged data which would contain all the data about the entity at once. The first of these options is too complex, and the second is not very useful, so in the end it was decided to only log a few key fields of entities instead of all of them.

Some of the actions listed below can be executed in multiple ways, most importantly by a user in a web interface, or through Mailtrain's API. Regardless of the source, the events should be logged, and the logged events account for all of these scenarios.

List Activity

The list entities should log the fundamental *CUD* events. In addition, since campaign messages are sent to subscribers of lists, and actions of subscribers to the campaign messages (e.g. unsubscribing) are potentially relevant to the lists themselves, a launch of a campaign should be logged for all lists targeted by that campaign. The data of this event should contain ID of the list, but also ID of the sent campaign.

It is also possible for Mailtrain users to create, update and remove subscriptions manually. These events should also be logged, and they should include the ID of the subscription in their data. However, subscriptions can also be changed by the subscribers or automatically, and these events should not be logged in entity activity.

- **logged data:** timestamp, actor, activity type, list ID, changed subscription ID (if any)
- **activity types:** *CUD* of a list, *CUD* of a subscription

List *fields* are also represented as entities in Mailtrain, but every field belongs to exactly one list, and does not make sense without a list. Therefore, rather than logging fields as their own type, every logged field event should also contain the ID of the field's list. A list's fields only define what data does the list store about its subscribers, and actual subscriber data are stored within the list's subscription tables. Therefore, we only need to log *CUD* events for lists' fields.

- **logged data:** timestamp, actor, activity type, list ID, field ID
- **activity types:** *CUD* of a field

Similarly to fields, *imports* also only make sense in context of a list, so they are logged as a part of the list event type, and their logged data should contain the ID of the import's list. Import runs are handled over longer time periods, where the state of an import run is tracked using an import's status database entry. For that reason, along with *CUD*, imports should also log a status change event. Events of status change should also contain the new status of the import.

- **logged data:** timestamp, actor, activity type, list ID, import ID, changed import status (if any)
- **activity types:** *CUD* of an import, import status change

Segments also always belong to exactly one list, so they are also logged under the list event type. Segments only define rules which a subset of a list's subscribers can satisfy, and otherwise do not have any state (the subscribers satisfying the segment's rules are queried on demand), so we only need to log their *CUD* events.

- **logged data:** timestamp, actor, activity type, list ID, segment ID
- **activity types:** *CUD* of a segment,

These are all the events logged under the list event type. Altogether they form this data schema:

- **logged data:** timestamp, actor, activity type, list ID, changed subscription ID (if any), field ID (if any), import ID (if any), changed import status (if any), segment ID (if any)
- **activity types:** *CUD* of a list, *CUD* of a subscription, *CUD* of a field, *CUD* of an import, import status change, *CUD* of a segment

Subscription Form Activity

Subscription forms are used by lists, but one form can be used by multiple lists, so the forms' log events are kept independent from lists' log events as their own event type. Other than lists using forms for subscription entry, there is no extra functionality to forms, so *CUD* events are all that needs to be logged about them.

- **logged data:** timestamp, actor, activity type, subscription form ID
- **activity types:** *CUD* of a subscription form,

Campaign Activity

Like other entities, campaigns should log their *CUD* events. A campaign can belong to at most one channel, so their current channel should be included in the *CUD* event data, if they belong to one. Also, since campaigns can have multiple different types that have different functionality, it may be useful to include the campaign type in *create* event logs.

Running a campaign in Mailtrain works similarly to running an import, i.e. tracking the campaign's (overall) state is done using its status database entry. For that reason, a status change event (with the new campaign status in its data) should also be logged.

A regular campaign can be reset, which discards data about its sent messages and returns the campaign to a state before it was sent. In practice, a campaign reset is used rarely, but should still be logged.

- **logged data:** timestamp, actor, activity type, campaign ID, campaign type (if the campaign was just now created), channel ID (if the campaign belongs to one), changed campaign status (if any)
- **activity types:** *CUD* of a campaign, campaign status change, campaign reset

A campaign can be set to track clicks of the *links* in its content. Therefore, knowing which links the campaign contains is important in terms of analyzing the campaign's statistics. In the existing code, links are not entities, and as such do not allow *CUD* events. Instead, links are registered for tracking when a campaign launches, and removed when it resets. Since we already log a campaign reset, we only need to log the link registration, when a new link is added to be tracked. This event should contain the link's ID, as well as its URL.

- **logged data:** timestamp, actor, activity type, campaign ID, link ID, link URL
- **activity types:** link registration

A user can *test-send* a campaign's message to a number of test users. Unlike sending regular campaign messages, this isn't included in the campaign's statistics, so it should also be differentiated from a regular campaign send event. A campaign message can be test-sent to one testing subscriber of a given list, or to all testing subscribers of a given list, so the logged test-send events should include the target list's ID, and if only a single subscriber is selected, the ID of that subscriber.

- **logged data:** timestamp, actor, activity type, campaign ID, list ID, subscriber ID (if any)
- **activity types:** campaign test-send

Triggers of a triggered campaign only define conditions that trigger sending the campaign's message. The triggering of triggers then happens based on subscribers' actions (i.e. not because of Mailtrain's users). Therefore the only loggable activity events for triggers are *CUD*, and since each trigger belongs to exactly one campaign, they should be logged with the ID of the owning campaign in the event's data.

- **logged data:** timestamp, actor, activity type, campaign ID, trigger ID
- **activity types:** *CUD* of a trigger

The campaign may contain *files* along with its content. These may differ by the method they are sent to the subscribers with the campaign: either they are directly attached to the e-mail (those are called attachments by the campaign), or they are publicly accessible via HTTP and linked from the campaign's content (those are simply called files by the campaign). The files are not fully fledged entities, e.g. they can only uploaded or removed, and not updated. They are also likely tied directly to the content of the campaign, and so logging them when the content itself is not logged probably won't yield useful information. For that reason, while the upload or removal of files should be logged, no concrete information about the manipulated files needs to be specified.

- **logged data:** timestamp, actor, activity type, campaign ID
- **activity types:** add attachments, remove attachments, add files, remove files

When put together, the events of a campaign form this schema:

- **logged data:** timestamp, actor, activity type, campaign ID, campaign type (if the campaign was just now created), channel ID (if the campaign belongs to one), changed campaign status (if any), link ID (if any), link URL (if any), list ID (if any), subscriber ID (if any), trigger ID (if any)
- **activity types:** *CUD* of a campaign, campaign status change, campaign reset, link registration, campaign test-send, *CUD* of a trigger, add attachments, remove attachments, add files, remove files

Template Activity

Templates define the content and files for campaigns. Similarly to campaigns, templates can be test-sent to test subscribers of a given list, so a logged event of a test-send should also contain the list's ID and the subscriber's ID if it is sent to only one subscriber. Templates may also link files, but unlike campaigns, they only define HTTP files, and not e-mail attachments. Like campaigns, the logged file manipulation events also do not need to include any details of the changed files. In summary, their logged events are *CUD*, test-sends, and upload or removal of files.

- **logged data:** timestamp, actor, activity type, template ID, list ID (if any), subscriber ID (if any)
- **activity types:** *CUD* of a template, template test-send, add files, remove files

Mosaico Template Activity

Mosaico templates are very similar to normal templates. One difference is that Mosaico templates cannot be test-sent. Another difference is that apart from content files, they can also upload Mosaico block thumbnail files. Those files are distinguished from the content files but otherwise logged in the same way. The logged events for Mosaico templates are *CUD* and upload or removal of either content files, or block thumbnail files.

- **logged data:** timestamp, actor, activity type, Mosaico template ID
- **activity types:** *CUD* of a Mosaico template, add files, remove files, add block thumbnail files, remove block thumbnail files

Send Configuration Activity

Since send configurations only define rules for sending e-mail messages, we only need to log *CUD* for them.

- **logged data:** timestamp, actor, activity type, send configuration ID
- **activity types:** *CUD* of a send configuration

Channel Activity

Channels only serve the function of grouping campaigns and providing default settings for new ones. However, the users might want to see what campaigns a channel contained at a given time, and so, in addition to *CUD*, a campaign being added to or removed from a channel shall also be logged. The campaign log data has to contain the IDs of the added or removed campaign.

- **logged data:** timestamp, actor, activity type, channel ID, campaign ID (if any)
- **activity types:** *CUD* of a channel, add campaign to the channel, remove campaign from the channel

Report Activity

Reports may be removed in the future, so logging their activity probably does not need to be very thorough, but until they are removed, it's at least good to log their basic manipulation events. Therefore, *CUD* events should be logged for both reports and report templates. The following schema describes report event data:

- **logged data:** timestamp, actor, activity type, report ID
- **activity types:** *CUD* of a report

And the following schema describes report template event data:

- **logged data:** timestamp, actor, activity type, report template ID
- **activity types:** *CUD* of a report template

Namespace Activity

Namespaces only provide hierarchical structure to other entities, so their only relevant log events are *CUD*.

- **logged data:** timestamp, actor, activity type, namespace ID
- **activity types:** *CUD* of a namespace,

User Activity

The only relevant log event for user entities, apart from *CUD*, is a password reset. Users themselves can do many more actions of course, but those are handled by the other entity activity event logs.

- **logged data:** timestamp, actor, activity type, user ID
- **activity types:** *CUD* of a user, password reset

Entity Sharing Activity

A key feature of Mailtrain's security system is entity and namespace sharing using roles. Users may share an entity with another user by assigning the user a role within that entity. To log this action, we need to know the entity, the assigned role, and the user the role is assigned to. Sharing entities works the same way for every entity, so all these events are logged in the same way. This results in the entity type being variable, so the entity is now defined by a pair of its type and its ID. Also, note that unlike most events, an activity type is not needed, since the only activity type possible is role assignment.

- **logged data:** timestamp, actor, entity type ID, entity ID, user ID, assigned role

Global Settings and Blacklist Activity

Finally, users may also manipulate certain settings which are not tied to an entity with an ID, so no entity ID is included in the following log data.

The first event type is the modification of the global settings. The events belonging to it only need the timestamp and actor ID as data.

- **logged data:** timestamp, actor

The next event type is the modification of the blacklist, i.e. add or remove an e-mail address from the blacklist. These events, alongside the timestamp and actor ID, also log the e-mail address being added or removed.

- **logged data:** timestamp, actor, activity type, e-mail address
- **activity types:** blacklist e-mail, un-blacklist e-mail

3.1.2 Tracker Event Types

While logging entity activity is useful for administration, the goal of this thesis is also to log and visualize various statistics of lists and their campaigns. This is very useful for owners and maintainers of lists, because it allows for the analysis of effectivity, performance, and reception of campaigns, long-term list performance, comparing the performance of campaigns, etc. These trackers of lists and campaigns log events related to subscribers of lists, and since lists may have many subscribers, the amount of data logged is likely to be much larger than that of entity activity logs.

There are some considerations regarding the European Union General Data Protection Regulation (GDPR) [20]. Certain data logged by the proposed trackers satisfies GDPR's definition of *personal data*, and as such, GDPR's regulations apply to this data. Unfortunately, advanced subscriber privacy management is beyond the scope of this thesis, so in practice, many obligations (for example, deleting the data based on a user's request) are left up to Mailtrain's users to be dealt with manually. However, if the users decide that they do not need subscribers' personal data, or that they do not want to deal with the obligations associated with GDPR, there shall be a configuration option in Mailtrain, which controls whether any sensitive data is included in event logs.

List Tracker

One of the two central entities in Mailtrain is a list. The function of lists is to manage subscriptions, and so the purpose of a list tracker is to track the lists' subscriptions.

There are several ways a subscription can be altered. New subscriptions can be added using the list's subscription form, a subscription may be updated to change the preferences of the user, the subscription may change status (e.g. when a user unsubscribes), and the subscription may be deleted by a periodic inactive subscription cleanup service. Subscriptions may also be created, updated, and deleted manually. While the manual subscription manipulations are already logged in entity activity, both cases need to be logged in a list tracker, since the two logs serve different purposes, with the goal of the entity activity log being mostly the ability to track the user who did it.

Every list tracker log needs to contain a timestamp, the ID of the logged subscription and the ID of the subscription's list, and the type of activity that was logged.

When a subscription is created, updated, or deleted, some information about the subscription should also be logged. The two important variables are the subscription's e-mail address and whether the subscription is marked as a test subscription. In terms of GDPR, the e-mail address of the subscription is sensitive information, so it should not be logged when logging sensitive data is not configured to be on. Still, we may want to identify a subscription based on some known e-mail address, even if we cannot store it directly. Luckily, this is achievable if we simply also log a hash of the e-mail address.

In all list tracker events, including a subscription status change, it's important to log the status of the subscription. Additionally, given a time interval, it's useful to be able to know the difference between counts of subscriptions with a given status over that interval. There are more than 2 possible values of subscription status, so with the log data proposed so far, it would be difficult to compute this difference from the logs. To make this easier, an extra variable is added to the log containing the previous status of the logged subscription if it had one. Also, for this method to work well, when a subscription is deleted, its deletion event log should contain its last status in the previous subscription status and not its current subscription status. With this, computing the difference between counts subscriptions with a given status over a time interval is simple: it is the difference between logs containing the status as a current subscription status and logs containing the status as a previous subscription status.

With all that taken into account, we get the following schema:

- **logged data:** timestamp, activity type, list ID, subscription ID, e-mail address (if sensitive data is logged), e-mail address hash, whether the subscription is a test subscription, subscription status (if any), previous subscription status (if any)
- **activity types:** *CUD* of a subscription, subscription status change

Lists can also be divided into segments, and there was some thought put into how subscriptions belonging to segments would be logged. It would be somewhat problematic, because the inclusion of some rules, such as a subscriber not opening an e-mail for a given time can cause a subscriber to start belonging to a segment at an arbitrary point in time. Mailtrain currently only checks which subscribers belong to a segment on demand (e.g. when a campaign is being sent), but logging subscriber segment inclusion over time would require periodic checking, which would be ineffective if there were many lists. For that reason, logging inclusion of subscriptions in segments is not implemented in this thesis.

Campaign Tracker

The second of the two central entities in Mailtrain is a campaign. The campaigns in Mailtrain serve the function of sending messages with given content to subscribers. The purpose of a campaign tracker is to track campaigns' messages and the subscribers' reactions to them. A campaign message is identified by the ID of its campaign, an ID of the target subscription, and the ID of the subscription's list, so each log should contain these values, as well as a timestamp and the type of event that was logged. The following paragraphs describe logged campaign tracker events.

We want to know all the intended recipients of a campaign. Every intended recipient should be sent a message, but in rare cases, sending the message may fail, so the set of recipients of a campaign is the union of successfully sent messages and failed messages. This means we should log both a success and a failure to send a message. Messages can also be test-sent, but these messages do not contribute to the total sent messages. While test-sent messages should also be logged, they should be distinguished from those sent after a campaign launches.

- **logged data:** timestamp, activity type, campaign ID, list ID, subscription ID
- **activity types:** message sent, message failed to send, message test-sent

In the case of triggered campaigns, it's also useful to know when a certain trigger was triggered. The log of a trigger event should include the ID of the trigger.

- **logged data:** timestamp, activity type, campaign ID, list ID, subscription ID, trigger ID
- **activity types:** trigger triggered

After a message is sent, it may still encounter events worth logging. In case the e-mail message is not deliverable, it may bounce back, which can be detected by Mailtrain. It's also possible for a spam complaint about the message to be sent back by the e-mail server, which Mailtrain can also detect. The user may also choose to unsubscribe using the link inside the message. All three of these events should be logged.

- **logged data:** timestamp, activity type, campaign ID, list ID, subscription ID
- **activity types:** message bounced back, message received spam complaint, subscriber unsubscribed using this message

When interacting with the content of the message, subscribers may be tracked when they open an image (this is achieved using a tracking image with no content), or click a link in the message's content. Both events are to be logged, with link clicks also including the clicked link's ID with the logged data. By default, both opening a message and clicking a link are only logged the first time they are tracked for each message, because logging them repeatedly would require large amounts of disk space, and also because knowing whether a subscriber opened a message or clicked on a link at least once is useful enough on its own. If Mailtrain's users wish, the application shall allow turning on repeated logging of these events in its configuration. A third log-worthy event is the first time a subscriber clicks on any of the links in a message's content. This gives general information about the subscribers that clicked on at least one link of the message.

Loading the content of the tracking image or links makes the subscriber connect to the Mailtrain server through HTTP, from which Mailtrain can deduce the time of clicking the link, the user's IP address, device type, and approximate geolocation (which is translated into a country). These data may be included when logging either a message opening, a specific link click, or a general link click. However, the IP address, device type, and country are all sensitive information, so they should only be logged when logging sensitive data is enabled in the configuration.

- **logged data:** timestamp, activity type, campaign ID, list ID, subscription ID, link ID (if the activity type is link clicked) IP address (if sensitive data is logged), device type (if sensitive data is logged), country (if sensitive data is logged)
- **activity types:** message opened, link clicked, any link clicked for the first time

Altogether this gives us this schema:

- **logged data:** timestamp, activity type, campaign ID, list ID, subscription ID, link ID (if the activity type is link clicked) IP address (if sensitive data is logged), device type (if sensitive data is logged), country (if sensitive data is logged)
- **activity types:** message sent, message failed to send, message test-sent, message bounced back, message received spam complaint, subscriber unsubscribed using this message, message opened, link clicked, any link clicked for the first time

3.2 Logged Event Data Transformations

The event data as described in the previous section have a set structure. That structure makes some queries of the data by visualizations more effective, but other queries less so. The goal of this section is to propose a few transformations of the logged event data. In IVIS' terms, a data transformation is meant to process the data from one or several signal sets into a new signal set for the transformed data, without modifying the original signal sets' data.

These transformations are mainly important for the visualizations to be able to query the data faster and more efficiently. That is why the transformations' utility will be omitted for now and will be left to be explained in section 3.3, which proposes the visualizations.

3.2.1 List Subscriptions

The first data transformation is meant to create a signal set displaying subscription counts of a list over time. This is quite difficult to do so far but can be done using list activity and list tracker event data.

At a list's creation, its subscription count is zero. The list's creation time can be found in list activity, or simply assumed that it is at some point before the first list tracker event. Then, given a point in time, the subscription count of the list at that point is the difference between the count of the list's list tracker events with subscription status 'subscribed' and the count of the list's list tracker events with previous subscription status 'subscribed'.

This would be computationally expensive if it would have to be done on demand for many time points, but there are possible optimizations. The first one is that the chart does not need to be granular to the level of single subscriptions, the values over time may be aggregated into buckets of short time intervals, like a minute. The second one is that the transformed data only needs to be computed once, and then saved to the transformed signal set, where it can be looked up. This means the schema of this transformed signal set would be simple:

- **logged data:** list ID, timestamp, subscription count

If a record displaying subscription counts at a given point exists, then to get the next record at another time point, one only needs to compute the difference of the aforementioned events between the two time points to get the difference of subscription counts in that time, and then add this difference to the subscription count of the previous record to get the new subscription count. This way the data can be computed relatively easily over time.

3.2.2 Campaign Messages

The second data transformation is very similar to the List Subscriptions transformation, only this time it is meant to display cumulative counts of various events of campaigns. Assuming the output of this transformation is a new signal set, its schema contains these fields:

- **logged data:** campaign ID, timestamp, number of failed messages, number of sent messages, number of opened messages, number of bounced messages, number of spam complaints, number of unsubscribes, number of subscribers who clicked on at least one link number of link clicks *(this last one means multiple data fields, one for each known link)*

The variable amount of fields due to link click fields will be left aside for now. Assuming we know the required links and want to know the values of a record given point in time and a previous record, we can calculate them by taking the values of the previous record, and adding to them the counts of the corresponding campaign's tracker events, which occurred between the two time points. This can be done even for links, as campaign tracker entries for a clicked link contain the link's ID. Also, unlike the previous transformation, no subtraction is done, as a cumulative amount of events cannot diminish with time. And like in the previous transformation, if there is no previous record, we can assume that the previous values are zero and take into account all of the campaign's tracker events before the given time point.

To briefly touch on the problem with links, since different campaigns have different links, the variable link click count would be pretty much impossible to implement in a relational database. This suggests that this data would be split into multiple tables by their campaign, which would solve this problem. This is a more technical problem, however, so it is beyond the scope of this chapter. It is further explained in Chapter 4.

3.2.3 Channel Campaigns

The third data transformation aims to create a signal set with information about channels' campaigns. Channel activity events log when a campaign is added to or removed from a channel, but there isn't a simple way to deduce from that data which campaigns belong to a channel at the current time.

The solution is to create a signal set containing records corresponding to the current campaigns in channels. Each record would represent a campaign belonging to a channel. This would mean that the records would have to be actively updated, added, or removed when a campaign's inclusion in a channel changes.

The campaign entries should also contain some overview of their current statistics. The data fields of interest would be the event types of the schema in the campaign messages transformation, as described in section 3.2.2, except the link clicks, which would make creating a schema impossible, as this time, all of a channel's records should be in a single signal set. Each record should also contain a timestamp of the time the campaign was created, so that the campaign records can be sorted time-wise. This leads to the following schema:

- **logged data:** channel ID, campaign ID, timestamp of campaign creation, number of failed messages, number of sent messages, number of opened messages, number of bounced messages, number of spam complaints, number of unsubscribes, number of subscribers who clicked on at least one link

3.3 Proposed Visualizations

The following section describes the proposed visualizations to be added to Mailtrain. The source of data for these visualizations should be logged events or results of some data transformation described in the previous section.

3.3.1 Entity Activity

The last section defined events of entity activity, which are logged when Mailtrain's entities are manipulated. As stated before, the purpose of having these

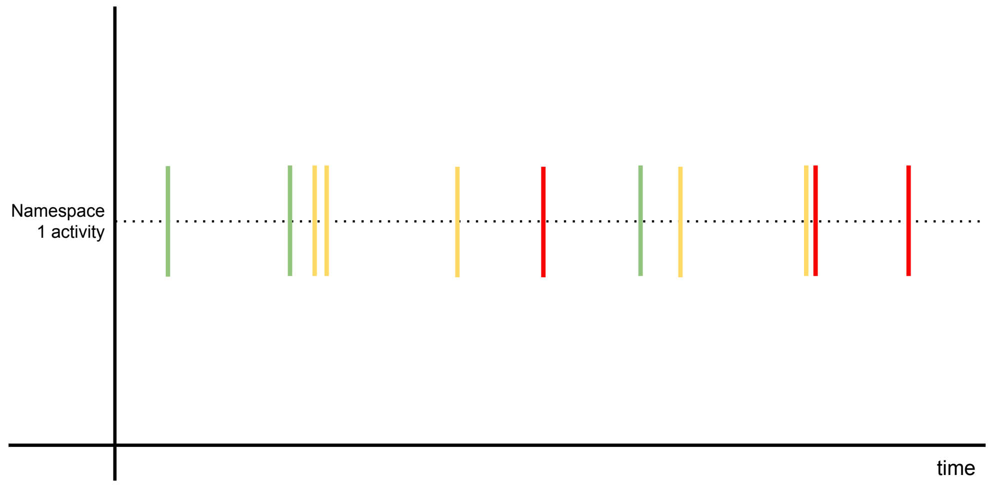


Figure 3.1: A draft of the entity activity visualization

events is to be able to search if any of the entities have been manipulated improperly, either because of an error or an attack.

There are two common ways of searching for these kinds of activities. The first is the activity of a specific entity, where we suspect some known entity of being improperly manipulated, so we view the activity done to that entity and look for events that might have caused it. The second is the activity of a user, where we suspect some user in Mailtrain may be malicious (for example, their account may have been taken over by an outside attacker), and we want to see all the events that the user caused over some time.

Both of these methods can use the same visualization. Events occur over time, so the visualization should display a timeline, and the events may be displayed as points on the timeline. Hovering over a point with the cursor should display a tooltip to learn more about the details of the event the point represents. This means displaying what type of event it is, the exact time the event was logged, the user who caused it to happen, and all other information relevant to the event.

However, there should be a distinction in the controls. The chart of activity of a specific entity should have a selection of the entity, which probably means an entity type selector, and a selector of a specific entity so that the visualization would show only the events relevant to it. These selectors may even be optional, which would allow displaying events of multiple entities at the same time. To somewhat differentiate different activities, ones from different entities or different entity types would be distinguished by color. On the other hand, the user activity visualization would only contain a user selector, to filter the events based on which user caused them to happen.

3.3.2 List Subscriptions

For maintainers of a list, it is useful to see the list's performance over time. This allows them to judge both the short-term and long-term effects of their actions on the list. Knowing the state of the list over various time periods can help address possible long-term issues the list may have. This is why each list should have a

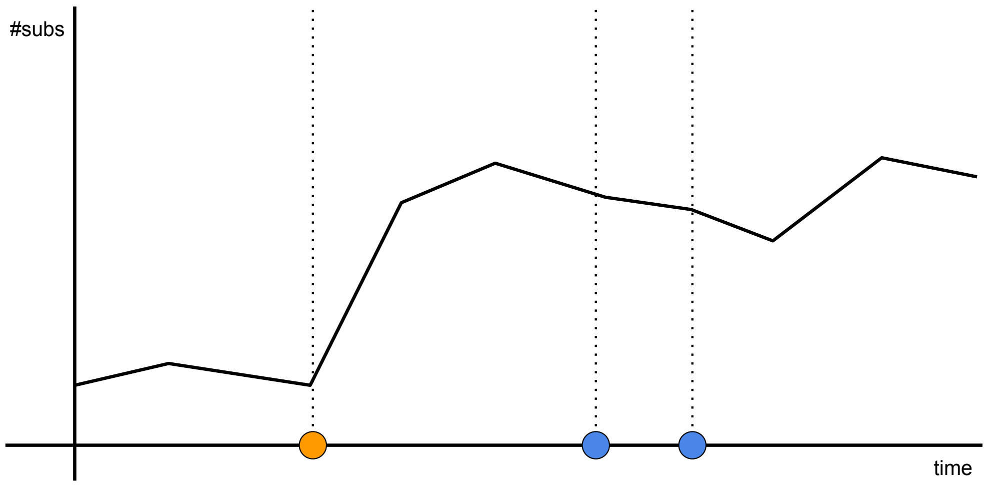


Figure 3.2: A draft of the list subscriptions visualization

visualization displaying its performance over an interval of time.

With that in mind, the main metric to judge how well a list is doing, with regard only to data from the list, is the list's subscriber count. For this reason, the base of the list subscriptions visualization for a given list is a line chart displaying subscriber count over time. The visualization can obtain this data from the list subscription transformed data, described in section 3.2.1.

However, simply displaying subscriber count over time without any context does not communicate much information. The list's subscriber count is affected by various actions to the list, e.g. sending a campaign, or importing subscribers. Fortunately, all of these actions are already logged as list activity events, which were described in section 3.1.1. With them, we can grant additional context to the subscriber count line chart by also displaying points on the timeline signifying that an event took place. These events need to be filtered to only include events relevant to the list. Like the entity activity visualization, users may hover over the events to show a tooltip providing more information about the event.

3.3.3 Campaign Messages

Similarly to the list subscriptions visualization, a campaign messages visualization should display the performance of a selected campaign over time, and since a campaign defines the content of a message that is to be sent to subscribers, tracking its performance means tracking the events of its messages. In the case of regular campaigns, the information it provides is likely more short-term, due to campaigns only serving to send a single batch of messages.

The visualization has the form of a line chart, but with multiple lines, each representing a different event type. Each event line shows the cumulative count of events of a given type at a given time. The amount of event types is rather numerous, so the event lines may be split into three categories.

The first category contains cumulative counts of three activity types. Two of them are sent messages and failed messages, which show how many messages were intended to be sent. The third one is opened messages, which shows how

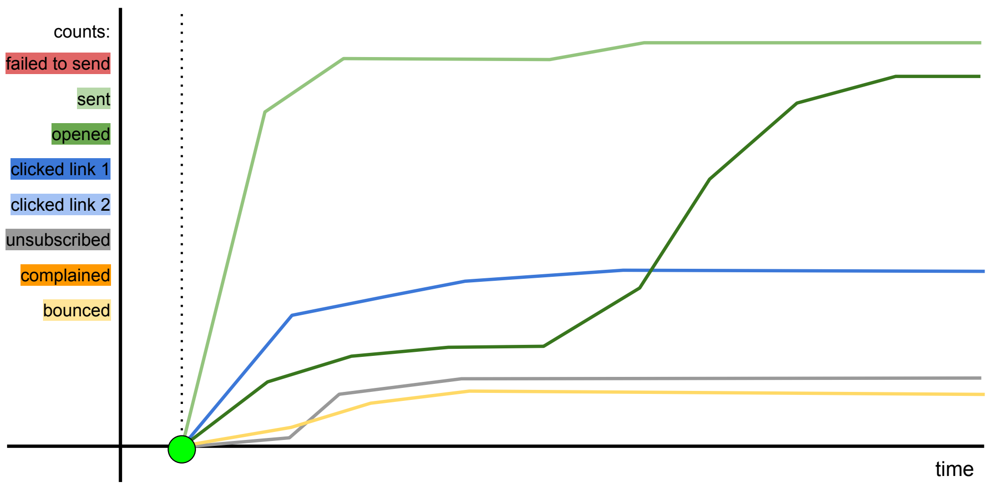


Figure 3.3: A draft of the campaign messages visualization

many people got to see the message.

The second category also contains three activity types, those being logged events of e-mail bounce-backs, complaints, and unsubscribes this campaign has received.

The last category shows link clicks. Since a campaign can have any number of links, the amount of event lines in this category is not known ahead of time.

The three categories may manifest as three distinct visualizations, or there may be only one visualization, but the lines should have toggleable visibility. Both of these options are viable.

Just like the list subscriptions visualization, the campaign messages line chart also displays events of the campaign's activity, defined in section 3.1.1, to provide additional context to the campaign's performance. The data for lines is obtained from the transformed campaign messages records, as defined in section 3.2.2.

3.3.4 Campaign Overview

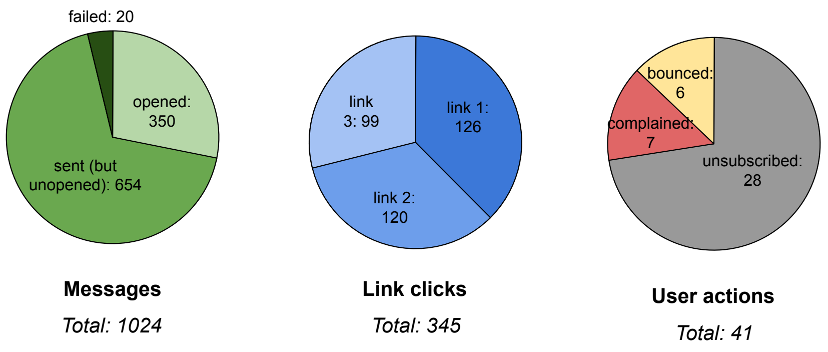


Figure 3.4: A draft of the campaign overview visualization

The campaign overview visualization uses similar data as campaign messages, but formats them differently.

The campaign overview visualization aims to display a statistical summary of a single campaign. Like campaign messages, the visualization is split into three categories, this time three pie charts, each displaying the values of each of its arcs, but also their sum.

The first pie chart should display the amount of sent messages, failed messages, and opened messages. However, the idea is for the pie chart's sum to equal the number of recipients, and its arcs then be visualized as a fraction of all recipients. This would work if the pie chart only contained sent and failed messages, but it also contains opened ones. We can solve this by subtracting the opened messages from the failed messages. The result can be viewed as follows: the first arc contains the messages which were not successfully sent, the second arc contains the successfully sent messages that were not opened, and the third arc contains messages that were both sent and opened. An opened message must have been successfully sent, so the difference between sent and opened messages always yields a non-negative number.

Similarly, the third pie chart has arcs corresponding to the number of e-mails that bounced, the number of e-mails rejected as spam, and the amount of unsubscribes.

The middle pie chart's arc corresponds to the campaign's links. As stated in the Campaign Messages section, this means that the number of arcs is not known ahead of time.

Unlike campaign messages, the campaign overview visualization does not need statistics over time, it only needs a single result from all the so far logged data in the campaign's tracker. This means that processing data directly from campaign tracker events would not be as computationally expensive as it would be in the case of campaign messages. Still, the data is the same as the data of lines in the campaign messages visualization at its most recent point. This means that the visualization may simply use the most recent record from the campaign messages transformed signal set, described in 3.2.2, which would save even more computation time.

3.3.5 Channel Campaigns

A channel is used for grouping campaigns, and it is safe to assume that campaigns in the same channel have a similar purpose. Therefore it could be useful for channel maintainers to be able to compare statistics of the channel's campaigns, which could help them assess how the campaigns are performing over time. This visualization is meant to serve this purpose by providing an overview of the performance of recent campaigns in the selected channel.

The channel campaigns visualization is essentially a modified bar chart. The bars are grouped by three, with each bar group representing one campaign. The bars themselves are divided into segments, with each segment having a height proportional to the value it represents. Each group of three bars has a similar meaning to a campaign overview visualization as described in section 3.3.4, except with bars instead of pies.

The first column's segments represent the amount of failed messages, sent

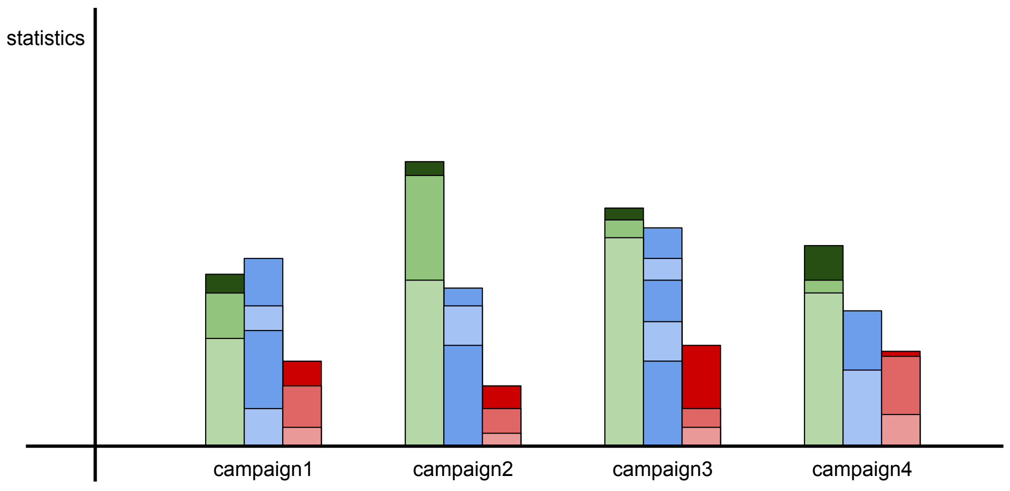


Figure 3.5: A draft of the channel campaigns visualization

but unopened messages, and opened messages. The third column's segments represent the amount of bounced e-mails, spam complaints, and unsubscribes. Unlike campaign overview, each campaign in the channel campaigns visualization serves only as a preview and does not need to display detailed data about itself. For that reason, the campaign's middle column is not segmented. Instead, the column represents the number of people who clicked on at least one link in the campaign. Other values of columns correspond to the non-link-related pies in the campaign overview visualization. All this data about the channel's campaigns is obtainable from the channel campaigns transformed signal set, described in section 3.2.3.

With that in mind, the columns of a single campaign already provide a generous amount of information, and having too many campaigns in this visualization may be overwhelming. A good idea may be to set an upper limit for the number of displayed campaigns to show only the most recent ones.

3.3.6 Channel Campaign Contributions

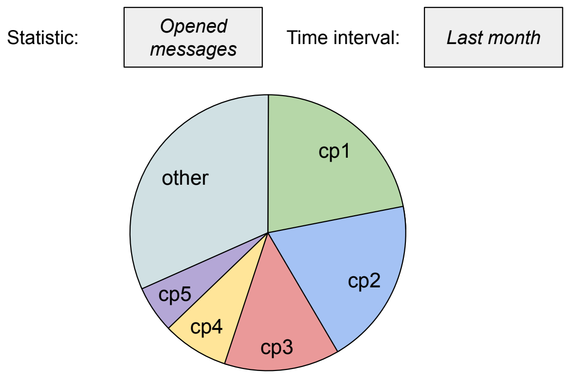


Figure 3.6: A draft of the channel campaign contributions visualization

A more direct way of comparing campaigns in a channel is by comparing only certain statistics of the campaigns. This allows users to quickly identify which campaigns contributed most or least to a channel, which can be used to adapt the content of future campaigns accordingly. For example, if users want to increase link clicks, and find that certain campaigns receive more link clicks than others, they can make the content of future campaigns more similar to those highly clicked campaigns.

The way to visualize this information is by using a simple pie chart, where each slice represents the selected statistic of a single campaign. We need to be able to select the statistic we want to compare, so a radio-type selector with possible statistics should be present. Finally, most of the time we do not need to compare all campaigns of the channel, but only ones from a certain time, so a time range selector should also be added.

Meaningful campaign statistics which can be compared are: sent messages, failed messages, opened messages, bounced e-mails, spam complaints, unsubscribes, and link clicks. As for link clicks, only general link clicks (the nubmer of users who clicked on at least one link) are comparable. Comparing specific link clicks would also require a selector of the link URL, which would drastically complicate the design of the visualization for little benefit. With that in mind, all of this data can be queried from the channel campaigns transformed signal set, described in 3.2.3. The records taken into account with regard to a time interval would be those records with the campaign's creation time within that interval.

4. Solution Architecture

In the previous chapter, we identified all logged events and visualizations to be added to Mailtrain. Before implementing them directly, we need to describe how Mailtrain and IVIS are to be integrated, which then lays a foundation for full implementation. This chapter describes the integration, both from a static module view and from a runtime view.

The first section describes the integration's architecture from the perspective of how the two projects are connected. This mostly covers the integration from a runtime point of view. The description includes how the integration allows various Mailtrain and IVIS processes to communicate.

Afterward, the architecture of the integration is described in two chapters as descriptions of extensions implemented for Mailtrain's and IVIS' side of the integration respectively. In contrast to the previous chapter, these sections focus more on the static decomposition of the architecture, i.e. describing all newly added modules, and existing modules that were modified.

Lastly, the extension made in this thesis must provide authorization with respect to Mailtrain's authorization system, so that it does not leak data that should only be visible to some users. Therefore, the final section of this chapter is dedicated to explaining how security is handled so that IVIS can accept authorized requests without leaking any data to unauthorized persons.

4.1 Architecture of the Integration

Both Mailtrain and IVIS are already existing applications with their own architectures. Therefore the integration architecture is designed as extensions on the side of Mailtrain and IVIS. Explanations of Mailtrain's and IVIS' extensions are reserved for future sections. Before that, this section instead focuses on how the two projects are connected. This includes both the code and runtime perspective. However, from a code standpoint, Mailtrain and IVIS are simply two separate modules and further decomposition is left for sections describing the extensions themselves, so the runtime perspective is prioritized.

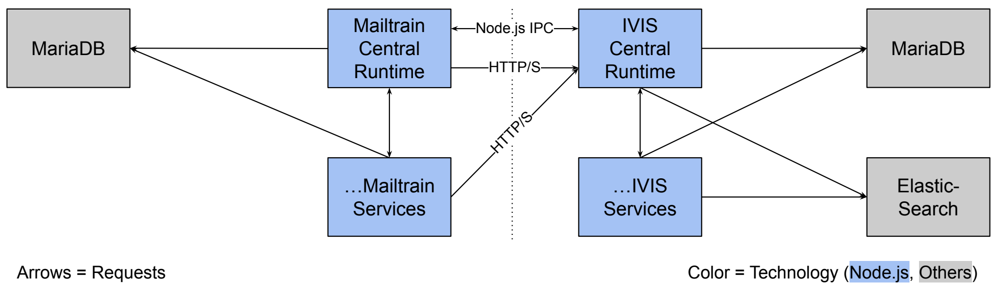


Figure 4.1: Runtime view of the integration's architecture

4.1.1 Decomposition into Runtime Modules

Mailtrain and IVIS share most of their key technologies, and have almost identical architectures. Both use Node.js for server-side code and React for client-side code, both can run their backend on Ubuntu or CentOS operating systems, and both use MariaDB as an SQL database to store their data. Additionally, as stated in section 2.1 on Mailtrain's background, if we want to extend IVIS with our code, we need to use IVIS' extension manager, which necessitates directly including IVIS from another Node.js process.

Due to the commonalities between Mailtrain and IVIS, it might be possible to run IVIS directly from Mailtrain's code, making them share one runtime process. This would bring some advantages, for example, communication between them would be very easy since one project could directly call the methods of the other. Unfortunately, there are many more disadvantages. First, this would impede the integration's scalability. Due to the nature of JavaScript, it is impossible to create a multi-threaded process in Node.js. Mailtrain already needs to split its computational load to other processes, so adding IVIS to it would needlessly slow it down. Second, even if the projects share many properties, they were not designed with this integration in mind. Running them from a single process could cause conflicts of functionality, and even if all of them are accounted for now, more errors may appear when IVIS receives an update.

Therefore the projects are to be split into two runtime components (in fact possibly more, since both Mailtrain and IVIS can run as multiple runtime components themselves). Since a custom Node.js program is needed to use IVIS' extension manager, the solution is to create a wrapper program defining all the extensions, and make Mailtrain launch it as a new process on its startup, alongside other Mailtrain service processes.

A similar problem is how to treat the databases. While both Mailtrain and IVIS use MariaDB, they use different databases themselves to avoid conflicts of table names, etc. From a runtime perspective, both databases are accessed from a single MariaDB instance. However, they can also be separate MariaDB instances, which may allow future extensions to run IVIS on a completely separate machine from Mailtrain, allowing for even more scalability in the future (even if it is not implemented in this thesis).

4.1.2 Communication

At runtime, Mailtrain and IVIS need to coordinate and share some of their data, which requires some amount of communication between them. Both Mailtrain and IVIS have methods of communication with their child processes, but the existing methods are not sufficient for the needs of this integration. This section describes how the communication between Mailtrain and IVIS is handled from a general technical perspective.

The section focuses on server-to-server communication. While client-to-server communication is affected by this integration as well, its needs are supported by existing communication methods in Mailtrain and IVIS, so it does not require any special treatment. Therefore it will be omitted here.

Communication via a Node.js IPC

As stated in the previous section, IVIS is launched from Mailtrain on each startup. Mailtrain already manages other Node.js processes (which it calls services) and launches them on startup in a consistent way. the process is started using the `fork()` function of the child process [9] module in the Node.js standard library. This function is designed to spawn a child Node.js process, and after spawning it, it returns a handle for communicating with the child process.

The inter-process communication (IPC) handle provides a simple, reliable, and secure way to send messages both ways between Mailtrain and IVIS server processes. Unfortunately, both Mailtrain and IVIS servers are comprised of multiple Node.js processes and the child process handle can only be used to communicate between the parent and the spawned child, which in this case is the central Mailtrain server process and the central IVIS server process. Due to the advantages of communication using the child process handle, it is the preferred server-to-server communication method, where it is possible to use it.

Communication via HTTP

However, for any communication between other processes than Mailtrain's and IVIS' central server processes, the required child process handle is inaccessible, so a different method of communication needs to be used. Fortunately, the only communication not between Mailtrain and IVIS central server processes is communication of Mailtrain's Activity Log module (explained in section 4.2.2), in which case the recipient is always the central IVIS process, i.e. only the sender may be a random process. This is very convenient, as both Mailtrain and IVIS central server processes can be communicated with by anyone using HTTP calls to their REST API. Therefore, HTTP is the communication method used where the target process cannot be reached using a child process handle.

There is still a problem with authentication because HTTP requests can be impersonated, so the recipient needs to recognize that a request comes from a trusted sender. This can be achieved using a secret token that all trusted processes know. Each of these cross-process requests then includes the token in its data, and the recipient can confirm its validity by checking that it is the same as their own secret token. Once a request like this is validated, since it comes from another server process, no more permissions need to be checked, i.e. the request has global access to the recipient's data (limited to what the request itself can do, of course). Therefore, this token shall be called a global access token.

Because IVIS and all other communicating processes are spawned by Mailtrain or by some other process that was in turn spawned by Mailtrain, we do not even need to store this access token permanently. Instead, Mailtrain shall generate a global access token at each startup, and pass it to all of its child processes when it spawns them.

4.2 Mailtrain Extensions

Regarding the integration, Mailtrain has several responsibilities it needs to manage. First, it needs to spawn IVIS on every startup, so that it can work with

IVIS later. It is also up to Mailtrain to detect events established in Chapter 3. It then needs to send these events to IVIS where they are logged, so that IVIS has up-to-date information about the state of Mailtrain. Finally, Mailtrain should display IVIS' visualizations from its own client site, so Mailtrain should modify its client to allow for these visualizations to be displayed.

4.2.1 Spawning and Managing IVIS

One module added to Mailtrain in this integration is dedicated to managing IVIS as a child process. As stated in section 4.1.1, Mailtrain spawns the IVIS process using the fork function from the child process module of the Node.js standard library. The described module is responsible for this functionality. Other processes spawned by Mailtrain also send a message notifying the main process that they have started. To retain consistency, this module should also listen for a message from the IVIS process that it has started, and log it in the standard log (i.e. not the event log which transfers data into IVIS).

This module should also handle communication with the IVIS process to some degree. The description of the integration's architecture proposed two types of communication with IVIS: via the child process IPC, and HTTP. In both cases, the responsibility of this module is to encapsulate this functionality into simple and easy-to-understand functions, which can be called directly from the process code.

In the case of communication via the child process handle, all possible requests are those designed in this extension, so each request type can be mapped to one function. Similarly, incoming requests are also all known, so a response can be coded directly in this module.

In the second case, we only need to consider outgoing HTTP requests, since IVIS does not need to communicate with Mailtrain using its processes. IVIS includes an API endpoint that can be extended to provide any functionality, and Mailtrain is most likely to communicate only with IVIS using the API, to keep the convention of trusted routes being used only by IVIS clients. That said, due to the flexibility of HTTP, only one general-use function needs to be provided, which sends a request to IVIS given a URL path, HTTP method, and request data.

As mentioned in the previous section, HTTP requests are authenticated using a global access token generated on each startup of Mailtrain, and passed to any child processes. As a module for managing communication with IVIS, this module also has a responsibility of generating this token, or recognizing that it was passed from a parent process and accepting it.

4.2.2 Event Logging

The detection and logging of events in Mailtrain is handled by a new code module named Activity Log. This module is meant to track events reported to it by Mailtrain, process them as needed, and send them to IVIS to be logged. Activity Log provides an interface for logging an event, which can be used by other modules of Mailtrain in which the event happens. This means that many other modules in Mailtrain have to be changed to use this interface when they encounter an event to be logged.

Processing the data by Activity Log mostly means formatting it to be accepted by IVIS. This usually requires little to no work, as IVIS' extensions can be programmed to accept this data. The most important part of processing is stripping sensitive data off the event data if a setting to log them has not been turned on in the configuration.

Regarding the sending of logs, it would be inefficient to send the logs one by one, since they can be very numerous. Therefore, Activity Log instead builds a queue of events to be sent, and once it reaches a certain threshold of messages, or some time passes, it sends them in bulk.

From a runtime perspective, the Activity Log is not differentiated. Instead, Mailtrain splits itself into multiple processes, all of which use Activity Log independently in their own process. This leads to having multiple queues, so not all events will be logged in the order they happened, but since pretty much all of them include a timestamp, logging them out of order does not break anything.

Synchronization of Data

Activity Log does not store its event queues in the database. This means that if an unexpected crash occurs, some events may be lost. To keep IVIS up to date with Mailtrain's state despite this, we would need to add some functionality that synchronizes the data with IVIS. However, even if we designed Activity Log to be completely reliable, we would still need the synchronization, because a user may replace Mailtrain's database for another one, either as a backup or a migration from another machine, and Mailtrain wouldn't be turned on to log this. The user could also simply update Mailtrain from an older version.

The solution is for Activity Log to include functionality to send information about Mailtrain's data to IVIS to synchronize them. The function only needs to be invoked once on each startup, which is as soon as possible after a crash, an update, or a database change. While the synchronization cannot fully restore all of the lost data, it can at least restore some portion of it (e.g. subscriber count of lists), and inform IVIS whether any entities have been removed, so IVIS does not have to dedicate resources to tracking them anymore.

4.2.3 Displaying Visualizations from the Client

Finally, the client side of Mailtrain has a responsibility to display the logged data to the users. This entails making new pages for visualizations to be displayed or extending existing pages to display the visualizations. Mailtrain has pre-defined styles and a flexible structure of client routes, so most of the work in adding a new page is creating an appropriate React component for the page, and adding it into the client routes structure. Once it is added, Mailtrain automatically takes care of displaying it as an item in the appropriate menus to be navigated to and rendered.

Another thing to solve is how to display the visualizations themselves. Luckily, IVIS has an intended way to implement displaying its visualizations from other sites, which is by embedding them. The IVIS project contains a JavaScript module called embedding, which contains methods that embed various entities from IVIS into other web pages. The module is not used by IVIS itself. Instead, it is intended to be imported and used by other applications, to simplify embedding IVIS' content in other pages. The pages which display the visualizations in Mailtrain's client shall therefore use the embedding module to embed the visualizations inside them.

4.3 IVIS Extensions

The responsibilities of IVIS added by this thesis are accepting data sent to it by Mailtrain's Activity Log and storing them appropriately, transforming the data where needed, and providing client-side visualizations. Unlike Mailtrain, most new code forms new modules and little is added to existing modules. The volume of added code is also larger than in Mailtrain's extensions, so IVIS' extensions are split into more modules than Mailtrain.

4.3.1 Mailtrain IVIS

Since IVIS is a standalone application designed for various uses, the code of IVIS should not be modified to the needs of Mailtrain directly. Instead, it should either be extended generally, so that multiple applications can benefit from the extension, or it should be extended from Mailtrain's code using IVIS' extension manager. This can be done by creating a new Node.js program, which we will call MVIS (short for Mailtrain IVIS). Its main module, which is called on its startup, defines all extensions to IVIS that are needed to implement the integration with Mailtrain. After defining them, it launches IVIS' code directly inside itself, for which all the defined extensions then become accessible. Some extensions are simple and require only a few lines of code to write, such as changing the application title, but some extensions need to be split into their own code modules.

4.3.2 Modules by Logged Event Type

A large portion of the responsibilities of MVIS is managing IVIS' entities, e.g. signal sets or jobs, and due to the large volume of code, it may be beneficial to split these responsibilities into multiple modules. This chapter describes how that is achieved and the reasoning behind it.

Let us now explain how logged data is stored in IVIS. As mentioned in section 2.2 on IVIS' background, for IVIS to use its data effectively, it needs to store them in signal sets, which are similar to database tables. The proposed events from section 3.1 support this format of data. Therefore the extensions must define several signal sets, to which the logged events are then stored.

A data transformation in this context means code that queries one or several input signal sets, and based on their data, adds or transforms data in some other output signal set. In MVIS, data transformations are implemented in two ways: using IVIS tasks or jobs, and directly writing the transformations in MVIS code. IVIS jobs run in another process, which makes them more suitable for computation-heavy transformations. However, since they have to be run in another process, it takes some time for them to start, and there is no good way for

MVIS code to wait until a job is finished. For this reason, smaller and more versatile actions can be done manually in MVIS code. Regarding IVIS tasks, since we know what tasks are needed, they are implemented as built-in tasks using the extension manager. This way, unlike regular tasks, IVIS' database does not need to be changed every time the task's code is updated.

An obvious option to decompose this code is into modules of signal sets, and data transformations. However, signal sets or data transformations may have very little in common among themselves. The better option is to decompose them into modules based on the types of events they manage, or more generally, the types of entities they manage. This way, for example, transformations of campaigns mostly interact only with campaign-related signal sets. The following subsubsections describe these modules and their responsibilities.

Entity Activity Module

One such module is responsible for maintaining entity activity data. It defines one signal set for each event type of entity activity event types, as they were defined in section 3.1.1. This means it defines separate signal sets for lists, subscription forms, campaigns, templates, Mosaico templates, send configurations, channels, reports, report templates, namespaces, users, shares, global settings, and blacklist, for a total of 14 signal sets. Each signal set contains signals corresponding to all the logged data listed when describing each event type.

Lists Module

Another module is responsible for maintaining list data. It defines a signal set for logging list tracker events, as described in section 3.1.2. Unlike entity activity signal sets, which contain all entity activity events of a given entity type, a list tracker signal set is to be instantiated for each list separately, since the amount of events in them is likely to be very large, so splitting the events by their lists makes data in each of these signal sets easier to query. Consequently, the signal sets themselves can omit the list ID from the signals (since a list ID of a record is now determined by the record's containing signal set) and store only the rest of the logged data as signals.

For each list, the module also manages a job that transforms the data from the list tracker signal set into a subscription counts transformed signal set, which was described in section 3.2.1. Similarly, since this signal set is instantiated for every list, it can omit the list ID from its signals. The job aggregates and accumulates the subscription counts, which is a computation-heavy task, so it is implemented as an IVIS job.

Finally, for each list, the module also instantiates a signal set that stores list activity events (described in section 3.1.1) of that list only. This is for security reasons. As described in section 2.2.1, signal sets cannot control permissions for each of their records individually, meaning if a user is allowed to view one record from a signal set, then they are allowed to view all records from the signal set. This means that when a user views a list subscriptions visualization (as described in section 2.2.1), they must query entity activity data from a signal set dedicated only to the list's data, otherwise, they would have permission to see entity activity data of all other lists.

Campaigns Module

A similar module to the one above is for managing campaigns. For each campaign, it defines a campaign tracker signal set, which stores the campaign's tracker events, described in section 3.1.2. For each campaign, it also defines an IVIS job, which transforms the campaign tracker data into another transformed signal set for each campaign, which contains logged data as described in section 3.2.2. And lastly, for each campaign, it also creates a signal set dedicated to storing the campaign's activity data, so that permissions to view one campaign's data can be separated from permissions to view another. Similarly to signal sets in the module managing lists, Since signal sets only contain data of a single campaign, they can also omit the campaign ID signal.

The jobs transforming the campaigns' data have an extra responsibility compared to ones transforming list data, as the transformed signal sets' signals include click counts of each link. This means that the jobs need to detect newly registered links for the campaign, extend the schema of the transformed signal set to include a new field for the number of the link's clicks, and then also include the link's clicks as events the jobs look for.

Channels Module

Finally, there is a module for channels, which for each channel defines a signal set of this channel's campaigns, as described in section 3.2.3. Once again, since there is a signal set for each channel, the signal set may omit the channel ID from its signals.

The module contains a data transformation code, which adds or removes campaigns from the channel's signal set based on some of the channel's entity activity events (the ones logged when a campaign is added or removed from the channel), and update the record if the campaign's statistics change. This task is implemented directly in MVIS' code, since its functionality is rather broad, and at the same time not too computationally expensive.

4.3.3 MVIS Activity Log

All of the modules in the previous subsection are meant to store certain events sent to MVIS by Mailtrain's Activity Log. However, we also need a system that accepts these events and stores them inside the appropriate signal sets. This is why MVIS also has an Activity Log module.

Similarly to the Activity Log on Mailtrain's side, MVIS' Activity Log exposes an interface meant to be used by other modules. This time the interface provides functions to register a listener for events of a particular type. Then, when MVIS receives a batch of events from Mailtrain, the Activity Log sorts the events by their type, and callbacks each of the registered listeners with the events of the given type. The content of the listener function can then react to these new events. Usually, there should be a function that stores the events in the appropriate signal set. The described listener design allows having multiple functions reacting to a single event but the listeners can also be used to react to events in other ways. For example, many signal sets correspond to a specific entity. These signal sets need to be created on the entity's creation, and deleted on its deletion, which can be done by registering a listener in the Activity Log, that listens for creations and deletions of the given entity type, and creates or deletes corresponding signal sets in response.

The module also accepts Mailtrain's messages for the synchronization of data at each startup. When it receives synchronization data from Mailtrain, MVIS modules that maintain signal sets should also react to it and update their data. This includes creating or deleting some of their signal sets tied to entities in Mailtrain. Some signal sets may also update their data, which includes synchronization of subscriber counts for each existing list, and synchronization of campaigns belonging to a channel for each existing channel. It does not include synchronization of event counts from campaign messages, as since the data from campaigns is usually only important over shorter time periods, synchronizing the event counts would not bring much benefit.

4.3.4 Visualizations

Finally, let us address the module for visualizations. As stated at the beginning of this section, IVIS already provides options for embedding panels or templates in other client applications. We can deduce which signal sets are used for what visualization directly from code, so panels need to be instantiated. Furthermore, we also know the templates used at any point, so templates do not need to be instantiated either, and built-in templates can be used instead, which, like built-in tasks, do not require manually changing the database when they are updated.

The actual built-in templates need to be created, which are programmed as client-side React code defining what data are queried from the server and how they are displayed. Alongside the definition of their content on the client side, each built-in template also needs to be registered with its parameters on the server side, so that it can be embedded. This is mostly for security reasons, as parameters of the template need to be known for IVIS to understand what signal sets can an embedded template query.

The module implements several server-side listeners, which, on receiving a request to embed a visualization, provide the appropriate built-in template, its parameters, and an access token. The built-in template and its parameters are passed since they may be dependent on the data which may be only known to IVIS. The access token is returned to allow data from the template to be securely queried. Another REST route should be implemented for requests of data that are known only by Mailtrain and not IVIS. This call simply relays the client's request to Mailtrain to find out certain entity properties. Details about both why the token needs to be passed and why an extra REST route for Mailtrain data is needed are explained in section 4.4.2.

Regarding built-in templates, IVIS does not provide support for embedding built-in templates, so it needs to be extended to be able to embed them. This is an extension that can be made to IVIS itself since it can be useful to more applications than just Mailtrain.

4.4 User Management and Authorization

As stated in section 2.1 on Mailtrain's background, Mailtrain provides a rather advanced system of permission management. It allows multiple users, each with permissions controllable to the level of different actions to entities, thanks to user-configurable roles. It also supports a hierarchical system of namespaces, where if a user has a role in a given namespace, they also receive certain permissions on all entities in the given namespace. This is quite a complex system, and when users interact with IVIS, it should understand the permissions the users have with respect to Mailtrain. Not providing the same permissions could cause a data leak, where unauthorized users see information about an entity they should not have access to.

The most thorough way of mapping user permissions from Mailtrain to IVIS is to clone the entire structure: copy the hierarchy of namespaces, copy every entity but place the appropriate signal sets and jobs instead of it, and copy the placement of users and their permissions. However, this would be very complicated to make, and in many ways, users usually will not be interacting with IVIS in a way for this to be needed, so permission management is done differently.

There are several ways in which IVIS interacts with the outside environment, and as such requires to check certain permissions. The first are requests from Mailtrain's server, each of which contain a batch of events passed to Mailtrain's Activity Log, or synchronization data. The second are requests from Mailtrain's clients when they require data for an embedded template displaying a visualization. The potential third way is a direct connection to the IVIS server from its web client.

4.4.1 Handling Requests From Mailtrain's Server

In the first case, checking for permissions is very simple. If a request is a batch of events or synchronization data from Mailtrain's server, then no permissions need to be checked, because the Mailtrain server is a safe environment. The only problem is verifying whether a request is really from Mailtrain. Luckily, this is a problem already solved in section 4.1.2 describing the communication between Mailtrain and IVIS, where the communication is achieved either using a safe communication handle from Node.js, or using HTTP authenticated with a shared secret global access token.

4.4.2 Handling Visualization Embedding

The second way IVIS is communicated is when a Mailtrain client requests data for an embedded template. In this case, a client is not a trusted environment, so permissions need to be checked in some way, and without the knowledge of permissions that Mailtrain has, this cannot be done directly. Luckily, for the most part, this is not necessary.

An embedded entity in IVIS runs in its sandbox endpoint. The sandbox endpoint in IVIS does not check permissions based on user credentials, instead, it checks them based on an access token. Unlike the previously mentioned global access token, this token is scheduled to expire after a minute, but as long as the client has it, they can renew it to prolong its validity, regardless of who they are. The only problem with this is that in order to acquire the access token, authentication is necessary. Fortunately, this only needs to be done at the initial request to view an embedded template, even if the visualization continues to query data after that.

The way to solve this is to use an indirect call through Mailtrain's server. While IVIS does not know the permissions of the client's user, Mailtrain does, and so the first call to request data for an embed is always directed to Mailtrain's server. As described in section 4.3.4, an initial request needs to be made to obtain the built-in template and its parameters anyway, so it is easy to also make it serve as a request for an access token.

Once the Mailtrain server receives the client's request, it checks if the user has the appropriate permissions to view this embed, and may immediately reject the request if the permissions aren't satisfied. If the user has the required permissions, the Mailtrain server can now relay the client's request to IVIS. As has been explained in the previous section, this server-to-server communication is safe, and IVIS accepts Mailtrain's request. Once the Mailtrain server receives request data, including the token, it returns them to the client, which can then use it to query the required data directly from IVIS.

Finally, sometimes a visualization may need data that is known only to Mailtrain. However, since the template is embedded in an iframe element, any tokens used to authenticate the user in Mailtrain's server are not accessible to the template inside. Therefore, the template must do a relayed request again, this time through IVIS. IVIS then relays the request to Mailtrain, which checks the user's permissions and either returns the data or denies the request. For this request to work, IVIS needs to remember the Mailtrain user which is tied to each access token, so that it can tell Mailtrain which user is doing the request. That in turn requires Mailtrain to include the user's ID in the original request where the access token is initialized.

4.4.3 Handling Direct Connection To IVIS

A final problem is that IVIS also has a direct web interface, and if a user wants to use it, Mailtrain would need to offload permission checks to Mailtrain for every request, which would be cumbersome. In the end, the main feature of IVIS from the perspective of user interaction is providing embedded visualizations. This is why we have decided to not implement anything to solve this last problem. The administrator of Mailtrain should gain access to IVIS from the default admin login. If a non-administrator user wants to have access to some of IVIS' entities, then the administrator has to grant them the permissions manually.

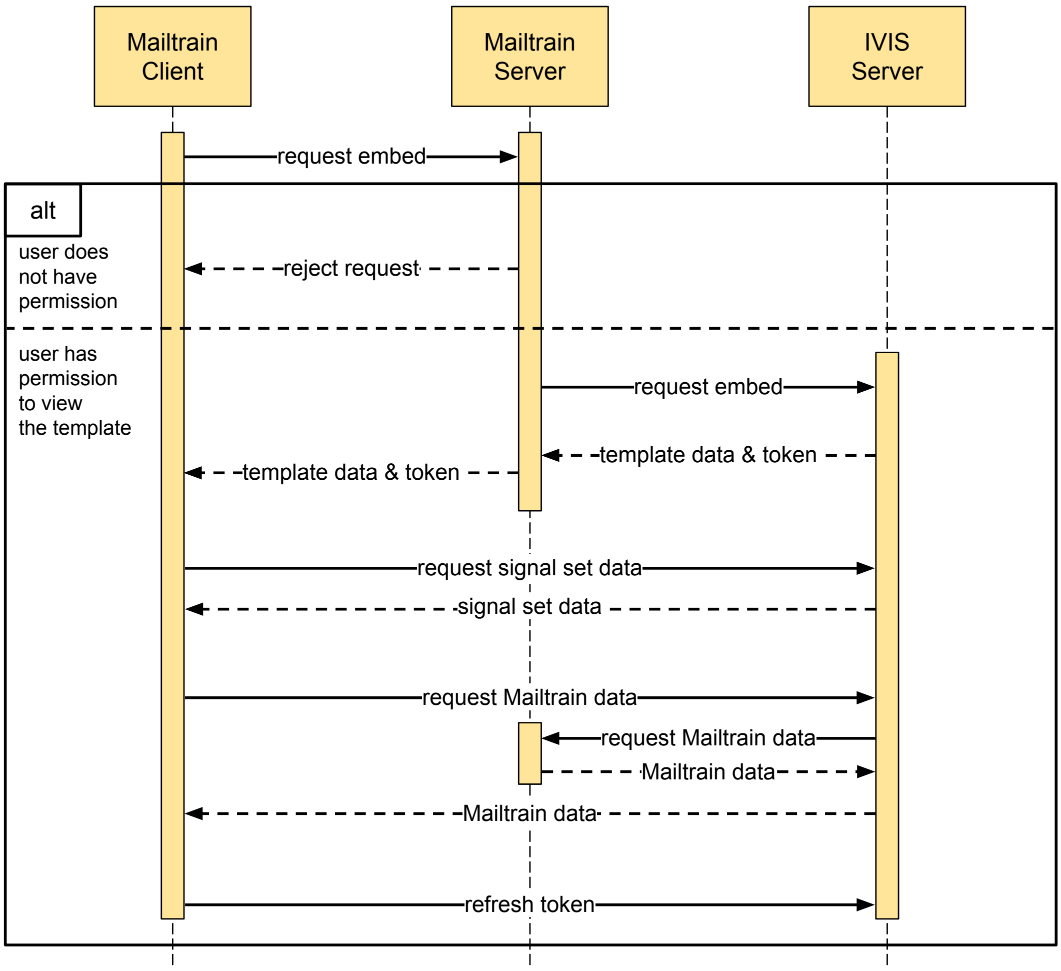


Figure 4.2: A sequence diagram of a request to embed a template

5. Implementation

The previous chapter described the general architecture of the integration of Mailtrain and IVIS, as well as the decomposition of the extension into modules, including their responsibilities. These modules are further described in this chapter from a detailed, technical perspective of their implementation. Sometimes, files containing the implemented code will be referenced. The files' paths represent a path from the root directory of Mailtrain's GitHub repository, e.g. `/README.md` represents a file in the repository's root directory.

The descriptions are split by module, and by the project they are extending. Extensions of standalone IVIS are described first, as they can be mostly explained on their own. After that, the modules of the integration are described in the general direction of data flow. This means the sections describe logging event data and managing IVIS at Mailtrain's server, the data being stored and transformed by IVIS' server, the data being visualized by IVIS' client, and visualizations being displayed from Mailtrain's client respectively.

The final section is dedicated to describing changes done outside of the main application code, which mostly means modifications of Mailtrain's setup code, configuration, and documentation.

5.1 IVIS Extensions

IVIS is a framework for general-purpose data processing and visualization, designed to be easily extensible and integrable into other applications. Therefore, its code should not be manipulated to serve Mailtrain's needs at the expense of IVIS' flexibility. With that in mind, IVIS was extended in a few ways throughout the development of the work done in this thesis, and while the extensions were generally done to serve Mailtrain's needs, they were implemented as general features that can be used by various applications seeking to integrate IVIS into them, or simply use its services. This way the main draws and advantages of the framework are not harmed. Apart from the extensions, several existing bugs were also found in IVIS and fixed during the development of this extension, which, of course, also does not harm IVIS in any way.

5.1.1 New Extension Manager Events

The extension manager is an IVIS module that allows extending IVIS using JavaScript code. It works by exposing a publish-subscribe interface to be used by the extending code, which can then register listener functions to insert its own functionality or reactions to certain events happening in IVIS. The framework already provides many extension points which can be 'subscribed to', but there were a few events that were not extendable, but needed to be added to the extension manager's published events for the integration with Mailtrain to work properly.

Global Access Authentication

The first of these is global access authentication, which in Mailtrain's case is done by a global access token, as described in section 4.1.2. This is done by inserting an extension manager method to IVIS' code which authenticates an incoming HTTP request. Before authenticating using the other ways, the extension manager method invokes any listener functions subscribed to this event, which is called `app.validateGlobalAccess`, with the request object as a parameter. If no functions are subscribed, by default the global access authentication fails and authentication proceeds as usual. If a function is subscribed to this event, it may inspect the request, and if it decides that it satisfies the conditions for authentication, it can let the authentication function know, which then results in the request getting authenticated as administrator. Extension manager methods cannot return values, so the extension manager instead passes a reference to an object to the listener function, which then mutates it to mark the authentication as successful.

Application Is Ready

Another new extension manager event is called `app.ready`. As might be hinted at by the name, this event invokes its subscribers when IVIS is ready to provide its services. This means that it has finished its initialization and is prepared to respond to requests from external agents. This event is mainly useful for synchronization when another application needs to wait for IVIS to start before it can use it.

Install Sandbox Routes

The last added event is called `client.installSandboxRoutes`. IVIS' pages in a web browser are each represented by a URL. However, since IVIS is a single-page application, these URLs do not correspond to the content of web pages served by the server. Instead, the page URLs are only relevant to the client to know what content to display. In the code, these URLs are organized into a tree-like structure and it is this structure that we want to be able to extend. IVIS has two endpoints, those being the trusted endpoint used for native IVIS code, and the sandbox endpoint used for foreign, potentially unsafe code.

The structure of the two endpoints is distinct, i.e. each endpoint has a client URL structure handled by its own code. Extensibility is already provided for the trusted endpoint as `client.installRoutes`, which is done by passing the structure to the listeners, which can then modify it at will. Extending the sandbox structure is done in the same way. Since all visualizations are run in the sandbox endpoint, making this event extensible is key to using built-in templates, the details of which will be described in section 5.4.

5.1.2 Visualization-related Changes

Visualizations developed in this thesis all use IVIS' visualization library, located in the `/client/src/ivis` directory of IVIS' GitHub repository (this time not

Mailtrain's). This mostly includes either charts, which are single React components representing an element of a visualization, and data providers, which are React components designed to make querying IVIS' server for data easier.

In the case of charts, the notable changes include `LineChartBase`, which received new optional properties `onSelect` and `onDeselect`, that allow other charts to use it to react to mouse movement in the chart. Another changed chart is `StaticPieChart`, which received minor changes to its visual customization options, those being an option to display text at the pie's center, and an option to display both percentages and real values of the pie's slices.

In the case of data providers, changes were mostly made to allow data queried using the providers to be processed using asynchronous functions. From MVIS' perspective, this is because sometimes visualizations require additional data from Mailtrain in relation to the data queried from MVIS' server. Therefore, after the query, MVIS needs to send another HTTP request before it can display the data, which data providers did not allow for in their previous state. The changes are simply to add an optional property to the provider components, which is the data transformation function that can be used asynchronously after the data is queried. The data providers affected by this are `TimeSeriesDataProvider` and related data providers, such as `TimeSeriesProvider`.

5.1.3 Embedding

IVIS already includes an embedding module which is meant to be run from other applications to embed IVIS' visualizations in their web pages. However, this module has a few drawbacks which don't allow it to be practically used in certain ways. This subsection describes the changes done to this module to expand the situations in which it is suitable to be used.

So far the module has had functions for embedding panels and templates, which receive, among other parameters, an IVIS access token to be able to authenticate in IVIS, and information about the embedded entity. Both panels and templates are entities the program doesn't know beforehand, so the embed functions have to first make a REST call to IVIS to get information about them, and only then proceed with embedding.

Once the function knows the embedded entity, it creates a new HTML `iframe` element in the selected page, linking to a generic panel render page in IVIS. The initialization parameters of the entity are passed to the iframe from outside by the function using the `Window.postMessage()` JavaScript method. The `Window.-postMessage()` is also used for all other communication between the iframe and its parent window.

Inside the iframe, an IVIS React component called `UntrustedContentRoot` manages communication between the parent window and the visualization template component, which `UntrustedContentRoot` contains inside itself. If an initialization request is received, the component loads the appropriate template and passes the initialization parameters to in via React's `props`. Figure 5.1 shows a schema of how these components are layered. In the case of Mailtrain, the "browser web page the user views" would be content from Mailtrain's trusted endpoint.

The function for embedding also takes care of periodically refreshing the given

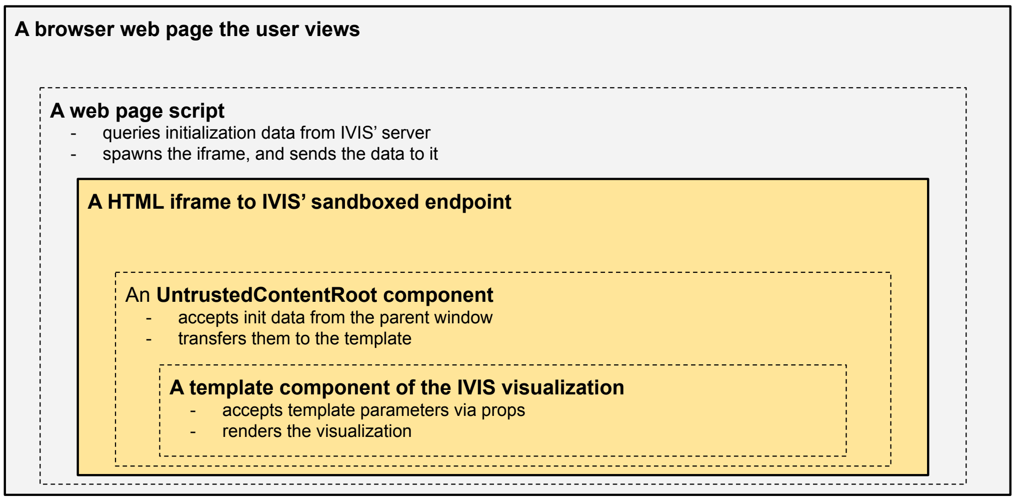

A browser web page the user views
A web page script
- queries initialization data from IVIS' server
- spawns the iframe, and sends the data to it
A HTML iframe to IVIS' sandboxed endpoint
An UntrustedContentRoot component
- accepts init data from the parent window
- transfers them to the template
A template component of the IVIS visualization
- accepts template parameters via props
- renders the visualization

Figure 5.1: A schema of an embedded IVIS visualization

access token so that the visualization can be viewed for a longer time.

Multiple Embeds In A Single Window

Listening for messages sent by `Window.postMessage()` is done by adding a listener function for an event of type *'message'* to some HTML element in the current browser window, which then calls the listener function anytime the element gets notified of a message event. The initial message is sent from the parent window to the iframe, but messages can be sent the other way too, e.g. when the embedded content wants to inform the parent window of its height. So far, when embedded IVIS content, controlled by the `UntrustedContentRoot` component, sends a message to the parent window, it sends it to the entire window, without identifying the origin of the message. This means that if the parent window embeds more than one visualization at a time, it cannot distinguish messages from different iframes.

This can be fixed by giving each iframe a unique ID. One parameter of the embedding functions is the ID of the DOM element inside which the iframe with the embedded content will be inserted. This DOM element ID should be unique among the window's other DOM elements, which is sufficient to distinguish different iframes.

Therefore, the embedding module has received two important changes. The first is that, when initializing an iframe, it passes the ID inside the iframe, which is done with a *sendId* parameter added to the query of the embedded content's URL. The second is that any received message events are only accepted if they are identified by the appropriate DOM element ID, otherwise, they come from a different iframe and are left to be handled by their intended listener.

The `UntrustedContentRoot` IVIS component was also updated to recognize the *sendId* parameter in its URL's query, and to add this ID to every message it sends to its parent window, so that it can be properly distinguished from messages from other iframes.

Embedding For Single-Page Applications

When a function for embedding a visualization is called, among other things in its initialization, it sets up a call to IVIS, which refreshes the given access token. The call then automatically repeats approximately every 30 seconds. Before the extensions, however, the only way to turn this refreshing off was to refresh the whole web page, which would terminate all currently running JavaScript code tied to it.

The problem with this is that single-page applications rarely switch or refresh pages, meaning that even if a user changes the content of the page, then any code for automatic refreshing of access tokens keeps running, even if it is not needed anymore.

This is solved by making the access token refreshing function keep a local variable indicating whether to keep refreshing and return a stopping function, which sets this local variable to indicate that refreshing should stop. The entire embedding function then returns this stopping function to the caller, which can then call it once the token is no longer needed.

Built-in Template Embedding

So far, IVIS' embedding module supports embedding panels and templates, but not built-in templates. Since it is beneficial for Mailtrain to use built-in templates, some extensions were made for IVIS to also support embedding them.

First, on the server side, a new method for generating a restricted access token was added. Methods for generating a restricted access token are different for each entity type. For example, when making a token for a regular template, the token creation entails finding the template entity in the database, assigning its basic permissions to the token, and then also assigning permissions to any signal sets and signals the template specified in its parameters. Built-in templates are not stored in the database, so their method only grants permissions to the required signals and signal sets.

On the client side, it must be possible for built-in templates to be displayable when embedded, as existing client-side embedding functionality only allowed embedding regular templates. Luckily, built-in templates are accessible directly from client code, so their content can be hard-wired into a client route. In the case of embedded content, which is always served from IVIS' sandbox endpoint, one only needs to add a custom sandbox path to the template, which can be done using the extension manager method `client.installSandboxRoutes`, as described in section 5.1.1. One technical detail is that the template should still be displayed as a child of an `UntrustedContentRoot` component, which, in the case of an embedded template, takes care of communication with the parent window.

Finally, the embedding module had to be slightly updated to account for these changes. This means adding a new method `embedBuiltinTemplate`, which is similar to a method for embedding a regular template, but instead of a template ID receives a path to the built-in template. The function also doesn't need to make a REST call to IVIS to find out details about the built-in template, since, as built-in templates are directly accessible from code, the function already knows all it needs to know.

5.2 Mailtrain Server

The first step of the implementation is at Mailtrain's server. The responsibilities of Mailtrain's server are to log proposed events to IVIS and to manage IVIS as a separate process called MVIS, which also includes managing communication with it. The following sections describe the implementation of modules responsible for this, as well as extensions of any related existing modules.

5.2.1 MVIS Manager

The first new module added to Mailtrain is responsible for spawning MVIS on Mailtrain's startup and managing communication with MVIS at runtime.

Spawning MVIS

The functionality of spawning MVIS is put into a new file called `/server/-lib/mvis.js`. Spawning MVIS is a straightforward task. As stated in section 4.1.2, Mailtrain already spawns other processes using a Node.js function `fork()`. There is a wrapper implementation of this function in Mailtrain, which handles terminating the child process after Mailtrain terminates, so all that needs to be done is use this wrapper function to spawn MVIS' root code file, located in `/mvis/server/index.js`.

IPC Management Using Child Process

Once MVIS is spawned, and its child process handle is obtained, the code also sets up listeners to various calls from IVIS using the Node.js child process IPC, namely calls to synchronize, query information about entities, and announcing that MVIS is ready (the events will be further explained in future sections). Along with spawning MVIS itself, this functionality is contained in a function called `spawn()`, which is called on Mailtrain's startup.

HTTP Communication Management

Management of communication with MVIS' API using HTTP is also rather simple. If MVIS' global access token is to be generated by Mailtrain on each startup, it needs to be shared with all child processes of Mailtrain, including MVIS itself. A good way to do this is as an environmental variable, which can be specified in the `fork()` function.

Therefore, the file `/server/lib/mvis-api.js` shall export a `token` variable. If the process was spawned as a child of Mailtrain, the token is accessible in an environmental variable called `MVIS_API_TOKEN`. If the process is Mailtrain's root process, then this environmental variable does not exist, in which case the token is generated as a long random string using the Node.js crypto [10] library. This way, no matter where the file is imported from, the `token` variable either generates the token or gets its value passed from a parent process. One remaining thing is that the environmental variable needs to be passed to MVIS and other child processes of Mailtrain which use Activity Log, namely sender-master for sending e-mails (which needs to pass it on to its subprocesses), and importer for handling list imports.

To handle the actual communication with MVIS, the module exports two functions for HTTP GET and POST requests to MVIS API. The module already knows the global access token, and it also needs to know MVIS' API URL base, which must be noted in a configuration file during MVIS' installation. Then, since the HTTP method is built into the functions' names, all the functions need to know are arguments of the target URL path and possibly the request body or extra configuration of the request. With these data, the function can then realize the request using the Axios [2] JavaScript library, where the global access token is inserted into the request's HTTP headers.

5.2.2 Activity Log

The core functionality of the Activity Log is contained in a new file `/server/-lib/activity-log.js`. The module provides several functions for logging events described in section 3.1. Different kinds of events contain different schemas of data, so this is reflected in the functions. The functions `logListTracker-Activity()` and `logCampaignTrackerActivity()` are meant to be called to log the tracker events, while `logEntityActivity()` logs events of entity activity. Even though entity activity contains multiple event types, most entity activity events share the basic data schema, i.e. timestamp, actor, activity type, and entity ID, so if `logEntityActivity()` also includes an identifier for entity type, and an argument for any extra data which the event wants to log in certain fields, which is implemented as a JavaScript object with key-value pairs, allowing to specify the fields and their data. That said, there are some logged events not tied to entities, and for them, special functions had to be made with appropriate parameters, those being `logShareActivity`, `logSettingsActivity`, and `logBlacklistActivity`.

Every event log includes a timestamp in its data, and a timestamp can be computed inside the logging functions, so for that reason, a timestamp is not in any of the functions' parameters and is instead computed when the functions are called. Similarly, many events of entity activity include an actor, which is the ID of a user causing the event. In Mailtrain's code, user IDs are usually not worked with directly, and instead, the user making a change is identified with a *context* variable, which contains the user information, along with other details of their authentication. To make calling the logging functions more convenient, they do not include the actor ID in their parameters directly. Instead, the entire context variable is passed to them through parameters, from which the functions then extract the actor's ID.

Sending Logged Events To MVIS

The Activity Log needs to communicate with MVIS from potentially different processes, so it imports the module for communication with MVIS using HTTP, defined in the previous section. Now sending of logged event data is done using HTTP requests, but doing so using a request for each event log would be inefficient. For that reason, the functions for logging do not send the events to MVIS right away. Instead, the module maintains a queue of events to be logged, and sends them to MVIS in bulk with a single request once the queue exceeds a given length (e.g. 100 events), or once some time passes with the queue not being empty (e.g. 1 second).

The queue is a global variable in the scope of the Activity Log module, and adding events to the queue is done asynchronously. This may cause synchronization problems, where two functions may decide to send the queue to MVIS at a similar time, which would duplicate the sent data. To avoid this problem, there is also a module-wise global variable informing whether the queue is being sent at the time. Then, if, after appending data to the queue, a function wants to send the data to MVIS, it first needs to check that another function already is not doing the same.

Another synchronization problem is that if events are added to a queue while it is being sent to MVIS, some data may be lost. That is why, when the Activity Log is sending events, it isolates the queue being sent and makes the functions append their events to a different queue, which is then used for the next request.

Extending Modules To Use Activity Log

With the logging functions defined, the other modules now only need to use it. Code-wise, this task is simple, as for each logged event, one only needs to insert a call of an appropriate logging function with appropriate variables into the code processing this event. Sometimes this also includes slightly restructuring the code, such as modifying a Knex database query to return IDs of the modified entities instead of only modifying them without any additional information, but the changes are mostly minor. The main challenge is finding the code which processes each logged event in the first place, which is made difficult since for the most part, Mailtrain lacks in-depth documentation of its code.

Talking about those changes in detail would probably not bring much new information. Nevertheless, the modified files in which Activity Log functions were inserted include most of the files in `/server/models/` and in `/server/-services/`. They also include two additional files of `/server/routes/rest/-subscriptions.js` and `/server/lib/importer.js`.

5.3 MVIS Server

After events from Mailtrain's server are logged, they are sent to MVIS' server to be stored and processed. Management of this logged data is the main responsibility of the MVIS server. This entails creating signal sets for the data to be stored in and managing jobs for transforming the data. Responsibilities also include accepting the event data from Mailtrain to be logged, and of course, integrating the extensions of MVIS with IVIS in the first place.

5.3.1 MVIS Program Entry Point

The entry point of MVIS' code in Mailtrain's repository is located in `/mvis/-server/index.js`. This is the code that is spawned as Mailtrain's child process, and its responsibility is to apply any extensions by MVIS into IVIS' functionality, and then launch IVIS inside of its process with the applied extensions.

Launching a custom Node.js program with the defined extensions which then includes the original has to be done not only for the main process, but for all of IVIS' services as well, i.e. IVIS' app builder, task handler, and ElasticSearch handler. Even if the services' actual functionality is irrelevant for now, they need certain extensions as well to function properly. For example, MVIS adds an extra configuration file for IVIS (the reasons for which will be explained in later sections), and all of IVIS' processes should know about it, which is achieved using an extension manager event dedicated to setting extra configuration. Ensuring that the extended services are spawned instead of the default ones is once again done using the extension manager, which allows setting the paths to the services' entry points. That said, we don't want to define many extensions repeatedly for every executable. Therefore, the extensions used by multiple processes are defined in a shared `extensions-common.js` file, which is then imported by all of the extended services' entry files.

Many defined extensions are simple, such as changing the application title to Mailtrain IVIS, or setting the extra configuration file. These can be defined in a few lines directly in the common extensions file or the target executable's entry code. However, many extensions are more complicated. In fact, all other modules in MVIS are eventually used in some extension set by this module, otherwise, it would be impossible to make them be used by IVIS.

5.3.2 MVIS Activity Log

The Activity log module in MVIS is responsible for accepting the event data from Mailtrain and providing an interface for other modules to react to these events being received, either to store them or to create, update, or remove signal sets and jobs dedicated to single entities from Mailtrain. Its code is located in `/mvis/-server/lib/activity-log.js`. The interface is realized as a publish-subscribe interface which other modules subscribe to if they want to react to certain event logs. The reactions to the messages can be listened to based on their event type, i.e. campaign tracker, list tracker, or different types of entity activity (list, campaign, template, etc.).

A special server route is set up using the extension manager to listen for events sent by Mailtrain's activity log. Every time MVIS receives a request containing a batch of events from Mailtrain, the Activity log splits the events by their event type, which yields several batches of events of a single type. For each batch of events, the Activity log then calls all the listeners for the batch's event type.

Letting listeners react to batches of events instead of single events leads to more efficiency since storing events into signal sets can be done using a single call for the whole batch of events. That said, there is a problem with this approach, which is that it does not call the listeners in any particular order. To give an example, the module managing campaign signal sets registers a listener which listens for campaign events and reacts to campaign creation or removal events by creating or removing a signal set dedicated to the campaign. The module also registers another listener which then listens for campaign tracker events and inserts them into those dedicated signal sets. Creating the signal set should happen before inserting the records, but removing the signal set should happen after, to avoid records being inserted into a non-existent signal set. To help with coordinating these listeners, the Activity log provides three functions for registering listeners: `before()`, `on()`, and `after()`. The `on()` function registers listeners that want to store events into signal sets. Listeners subscribed using `before()` are called before all listeners subscribed using `on()`, which makes the function ideal for creating signal sets before any events are inserted into them. Similarly, listeners subscribed to `after()` are called after `on()`, which is ideal for removing signal sets.

5.3.3 Communication With Mailtrain Through Node.js IPC

MVIS sends messages to its parent Mailtrain process using the Node.js child process IPC on several occasions. In some cases, a response is required from Mailtrain, but the system for IPC in Node.js does not support this, so a special system must be made for it. This is done using a system of request IDs. Put simply, when a function desires to receive a reply from Mailtrain, it sends a message and in its data includes a unique request ID number (in practice, the number is incremented modulo a very large number with each request, which should achieve uniqueness). Mailtrain is expected to reply to this message with a message containing the same request ID, so the function can set up a promise that is resolved when this reply message is received. There are 3 kinds of messages sent to Mailtrain using Node.js IPC.

Synchronization Request

The first is a request for synchronization of data with Mailtrain (as described in section 4.3.3). From an implementation perspective, it is more beneficial for MVIS to initiate the request for synchronization once it is prepared to synchronize the data, since Mailtrain does not know when that happens. Mailtrain's MVIS manager module was extended to respond to the request with information about its entities, i.e. an object with lists containing Mailtrain's campaigns, lists, and channels as JavaScript objects. Each entity contains its ID. Each list also contains the count of the list's subscribers, and each channel contains the IDs of its campaigns. Once MVIS receives this information, it uses this data to update the instances of signal sets dedicated to the given entities. The synchronization of lists' subscriber counts is done by inserting a 'synchronization' event into each of the lists' tracker signal sets, containing the subscriber count with its data, which tasks transforming the trackers' data then use to update counts of list subscribers. The synchronization of channels' signal sets is done by updating the records to include the data from signal sets of the campaigns belonging to the signal according to the synchronization data.

Entity Information Request

The second kind of message is a request for entity information. It is similar to the synchronization requests but may be sent at any point in MVIS' runtime when a visualization needs information about Mailtrain's entities. The request includes IDs of entities about which information is needed, as well as an ID of the Mailtrain user making the request. The ID is needed since the request is indirectly made by a client, which may not have permission to access some entities. The need for the user's ID also means that on the creation of an access token for a visualization, Mailtrain needs to send IVIS the requesting user's ID, and IVIS needs to remember it alongside the access token.

Once Mailtrain receives the request, it checks permission for each of the requested entities in regards to the user, and sends back a message with information from the database about each of the entities which the user was authorized to view.

MVIS Ready Message

The third kind of message is a message that MVIS is ready. As stated in section 5.1.1, this is a newly implemented extensible IVIS event, so all that needs to be done is to set the extension manager to send a message to Mailtrain once MVIS is ready. Mailtrain then waits for this message before it starts to ensure that MVIS is not communicated with while it is still not prepared.

5.3.4 Data Managing Modules

The responsibility of modules managing data is to maintain signal sets, either global ones or ones ties to single Mailtrain entities, and jobs related to them. The modules should also register proper methods to Activity log to store events or update signal set instances based on the events.

Functions Used For Implementation

Both signal sets and jobs already have IVIS functions for creating, updating, and deleting them in code. Properties required to create a signal set include (among other things) its ID, name, and a schema of signals in a JSON structure, which for each signal specifies its name, data type, etc. Signal sets also include a function for inserting a batch of records into them.

Properties required to create a job include its name, task specifying its code, parameters given to that task, and signal sets that trigger the job to activate when they get updated. The last property is useful to set jobs to activate automatically when their input signal sets update. This is used by the list managing module to set the subscription counting job of each list to activate when the list's tracker signal set is updated, and by the campaign managing module to set the campaign message event counting job of each campaign to activate when the campaign's tracker signal set is updated.

The above-described functions, along with functions to subscribe to MVIS Activity log events form the majority of the Node.js implementation of MVIS' data managing modules. There is one exception with channels, which is described below.

Querying Data

When a campaign is updated, and the channel module is notified of it through Activity log, the channel module wants to know if the campaign belongs to a channel, and if it does, the module must update the record of that campaign in the campaign's channel's signal set. The way to find the campaign's channel is to do a query of the channel activity signal set for the most recent event where the campaign in question was added or removed to some channel. Then, if the result is an event where the campaign was added, it must belong to the channel in the event's data. If the result is an event where the campaign was removed, or no such event exists, then the campaign does not belong in any channel at the moment.

Once the campaign's channel is known, another query needs to be made to find out the statistics of the campaign. This is done as a request for the most recent record in the campaign's signal set which contains counts of the campaign's various message events over time (as described in section 3.2.2). The campaign's record in the channel's signal set can then be found by the ID of the campaign and updated with the returned data.

5.3.5 Built-in Tasks

Part of the functionality of data managing modules in MVIS is done in Python, which is the language IVIS tasks are written in. There are 2 tasks in MVIS, which are used for transforming list data and campaign data respectively. Both are realized as built-in tasks since they are known at the time of development. To differentiate the task code written in Python from other code written in JavaScript, the code for built-in tasks is placed in files `/server/builtin-files/tasks/`, including the tasks' parameters, which mostly include the IDs of input and output signal sets. The built-in tasks' code and parameters are then registered on each startup of MVIS using IVIS' extension manager so that they can be used.

Both are based on IVIS' existing built-in aggregation task, which transforms a signal set containing records with values into another, which contains statistics (e.g. minimum, maximum, average) of those records aggregated by buckets of time intervals. The statistic was modified into a plain count aggregation, where a record of a given type counts as a single value. In both cases, the gathered values are not only aggregated but also accumulated over time. When the jobs are first initiated, they also insert a record with a timestamp of the campaign's or list's creation (which is also passed as a parameter to the task) and otherwise all zero values to their output signal set.

List Processing Task

The list processing task calculates the difference of subscriber counts for each bucket, based on subscription status and previous subscription status signals of list tracker records, as described in section 3.2.1. The task also checks if any bucket contains a synchronization event of list subscribers in the list tracker, described in section 5.3.3. If a synchronization event is detected in one of the buckets, then the count of subscribers in it is added to the difference of subscribers in the interval between the synchronization event in the bucket and the bucket's end, and the resulting value becomes the new synchronized subscriber count. If multiple synchronization events are detected in a single bucket, only the most recent one is needed to be known.

Campaign Processing Task

The campaign processing task calculates cumulative values of various events of its campaign tracker. Unlike the list processing task, it does not need to subtract any values or look for synchronization events. However, it does need to look for link registration events, and if it finds any such event for a link whose click count does not have a signal in the output signal set, it has to extend the signal set's schema to include the new signal. The task then computes the cumulative values for all events and clicks of all known links simultaneously.

5.3.6 Template Viewing Permission Management

As stated in section 4.4.2, users from Mailtrain authenticate themselves to view templates using IVIS' access tokens. These tokens grant permissions permission based on parameters of the chosen template, but also a selected IVIS user to which the token 'belongs'. This user is called an *impersonated* user. The total permissions are an intersection of these two sets of permissions so that no data leaks.

Since Mailtrain's users viewing an embedded template are not meant to modify anything, it is a good idea to make the impersonated user have read-only permissions for everything in Mailtrain, to avoid granting the users permission to edit anything by accident. This MVIS user entity is called *Mailtrain User*, and there only needs to be one used by all of Mailtrain's users. If every generated token for an embed impersonates *Mailtrain User*, then anyone using the token does not have access to writing anything in MVIS, and combined with the intersection of permissions of the selected template and its parameters, it ensures that the user can only query data for the requested visualization.

Mailtrain User can be created by defining a new IVIS role in MVIS' configuration, which gives the assignee of this role read-only access to every entity in MVIS, and then creating a user in MVIS with this role. Then, to impersonate the user in an access token, one only needs to include the Mailtrain User's ID as a parameter when using IVIS' function to generate the token.

5.4 MVIS Client

Once data are stored and potentially transformed in MVIS' server, they can be queried by MVIS' client to be visualized. Since we already know all visualizations in development time, the visualizations are realized using built-in templates. IVIS already provides functionality for displaying templates, and extensions were made in this thesis to allow IVIS to also display embedded built-in templates. Therefore, most of the work done on MVIS' visualizations is coding the built-in templates themselves.

5.4.1 Charts And Data Providers

As stated in section 5.1.2, charts in IVIS represent single elements of a visualization, implemented as a single React component, while data providers are React components designed to query data from IVIS' server. IVIS already provides several charts and data providers in its code that are usable by extensions, and while some of them were used by MVIS, some needed to be made without using any of them. In these cases, at the very least, the code of the existing charts and data providers in IVIS was able to provide a general guide of how charts work, which lead to slightly smoother implementation. The code for MVIS' charts and data providers is located in the `/mvis/client/src/` directory of Mailtrain's git repository.

DocsDataProvider

The only newly implemented provider is called `DocsDataProvider`, which is used to query single signal set records, called *documents* in ElasticSearch terms, given a signal set and its signals, and optionally instructions to filter or sort the data in a particular way. It is needed because IVIS' data providers for querying documents limit their queries to a given time interval of their timestamp values. The newly implemented data provider does not have this constraint and can query documents with any timestamp in a single query.

Once the documents are received from MVIS, the visualization may need to query additional data from Mailtrain, so an optional `processDataFun` property can be passed to the provider, which may modify the documents after they are queried.

Finally, a `renderFun` is passed to the provider, which, given the resulting documents, renders the visualization. When `DocsDataProvider` obtains the queried and processed documents, it calls `renderFun` with the documents as an argument, letting it render the visualization using the data.

The following code shows how `DocsDataProvider` may be used:

```
const visualization = <DocsDataProvider
    sigSetCid={"signal_set_1"}
    sigCids={["signal_5", "signal_6"]}
    sort={[{
        sigCid: "signal_timestamp",
        order: "desc"
    }]}
    limit={5}
    renderFun={docs => <GroupedSegmentedBarChart
        config={{barGroups: docsToBarGroups(docs)}}
        height={400}
    />}
    processDataFun={fetchMailtrainDataIntoDocs}
    loadingRenderFun={null}
/>;
```

EventLineChart

The `EventLineChart` is a component for displaying a line chart together with vertical lines signifying events. The component uses a chart from IVIS called `LineChart`, which takes care of displaying the lines and dynamically loading necessary data. `EventLineChart` then adds the vertical event lines obtained by querying a single signal set containing the activity events, to the created line chart so the lines are displayed together. The chart also modifies the tooltip displayed when the chart is hovered over to include information about the events. This was made possible by adding `onSelect` and `onDeselect` properties to the charts in IVIS, as explained in section 5.1.2.

EventChart

This chart still uses an existing chart in IVIS called `TimeBasedChartBase` to make dynamically querying data based on a selected time interval easier. `Line-Chart` uses `TimeBasedChartBase` as well, so their interfaces are quite similar. This allows `EventChart` to contain similar code to `EventLineChart`. One difference with `EventLineChart` is that `EventChart` can query multiple signal sets for activity events, and events from different queries can have different colors.

GroupedSegmentedBarChart

The final chart developed in this thesis is a bar chart in which the bars are organized into groups (hence it is *grouped*), and each bar is also vertically split into segments (hence it is *segmented*). This chart is the only one coded completely from scratch, although its code is loosely based on IVIS' `BarChart`. The chart does not query data dynamically, as that is generally quite difficult. Instead, it is meant to receive the data directly in its parameters, which define the structure of the chart, i.e. the groups of bars, the bars of each group, the segments of each bar, and properties (such as labels or values) of all of them.

The chart also includes a tooltip when hovered over each segment, which gives some extra information about the segment, its bar, and its group.

5.4.2 Built-in Templates

As stated before, charts represent single elements of a visualization, usually only focused on displaying data, and sometimes also querying it. Templates are composite elements representing a whole visualization. They usually consist of charts, along with data providers to query data, and also extra control elements helping to make the visualization more flexible. Often, a template only uses a single chart with some helper components. This is the case with most of the used templates, and all of the used charts.

The `EventLineChartTemplate` adds a selector of time range and a selector of line visibility to the original `EventLineChart`. This template is then used for list subscriptions and campaign messages visualizations, described in sections 3.3.2 and 3.3.3 respectively. Its final render can be seen in sections 6.2.2 and 6.2.3.

The `EventChartTemplate` is used to display the entity activity visualization (described in section 3.3.1) and adds a selector for filtering events by their type, a selector for filtering events of a single entity, and a selector for filtering events done by a single user. Its render is shown in section 6.2.1.

The `GroupedSegmentedBarChartTemplate` is used to display the channel campaigns visualization as described in section 3.3.5, and uses the newly implemented `DocsDataProvider` to query the most recent entries from a given channel's campaigns signal set to display their statistics. Its render is displayed in section 6.2.5.

There are also a few charts from IVIS which are used as well. The `RangeValuePieChart` is a new template used to display the channel campaign contributions, described in section 3.3.6, which uses a `StaticPieChart` with added selectors for a time interval to search the channel's campaigns in and for a statistic to compare the campaigns with, along with IVIS' `TimeSeriesProvider` to query campaign records in the selected interval. Its render is shown in section 6.2.6.

The `NPieCharts` template is used for the campaign overview visualization from section 3.3.4 and uses multiple static pie charts and a `DocsDataProvider` to query the latest entry from the selected campaign's messages signal set. Its render can be seen in section 6.2.4.

Obtaining Extra Data From Mailtrain

Sometimes, a visualization needs more data than what is easily accessible from MVIS' signal sets. This mostly entails details about entities, for example, names of campaigns. In that case, the visualization needs to request Mailtrain to get this data.

Since the initial request for embedding a visualization is done through Mailtrain, it is sometimes possible for Mailtrain to inject the required data in the template parameters while the request is passing through it. This is the case with `EventChart`, `EventLineChart`, and `NPieCharts`.

In other cases, the visualizations must do these requests during their runtime. Luckily, this was already handled in sections 5.1.2 and 5.3.3, ensuring that the visualization has space to query the data and that the query can be satisfied with regard to the permissions of the user viewing the visualization.

5.4.3 Displaying The Visualizations

As explained in section 4.3.4, all templates are realized as built-in templates, since they are known at the time of development. Built-in templates used to not be embeddable in IVIS, but an extension was made for IVIS to allow for this, as explained in section 5.1.3.

With that in mind, each built-in template needs to be registered on the server side with its parameters, as IVIS needs to know the parameters to be able to grant permissions to the template to query certain signal sets. Additionally, for built-in templates to be displayable from an embed, a client route needs to be installed at the sandbox endpoint of MVIS for each of the built-in templates using the extension manager. The routes are done for each visualization instead of each template, where 'visualization' refers to one of the visualizations described in section 3.3. This means that two routes representing two different visualizations may display the same template, only with different parameters. This is the case with `EventLineChartTemplate`, which is used both by *list subscriptions* and *campaign messages* visualizations. Generally, though, the visualizations are paired one-to-one with templates.

Each visualization also requires a server route in MVIS which, given a REST call to it, returns information about the visualization's template and parameters, as well as a newly initialized access token for that template. As was said in the previous paragraph, two routes representing two different visualization requests may use the same template. The visualizations are usually also parametrized in some way, e.g. the campaign messages visualization takes as a parameter the campaign whose message event counts should be visualized, which then affects the returned template parameters.

5.5 Mailtrain Client

Once visualizations of the logged data are accessible from MVIS, it is up to Mailtrain's client to create space for the visualizations in its web pages, and embed the visualizations to be displayed.

5.5.1 Visualization Requests And Embedding

Embedding the visualizations is the first part of the extended functionality in Mailtrain's client. The first step to embedding a visualization in IVIS is obtaining the template used for the visualization along with its parameters, and an appropriate access token that grants permission to query the template's data. This needs to be done using an initial request to IVIS.

As explained in section 4.4.2, requests from Mailtrain's client to access an MVIS visualization need to be relayed through Mailtrain's server, to be able to check the user's permissions. The previous section 5.4.3 described the routes that MVIS sets up, which provide the necessary data to display each visualization. Mailtrain's server then sets up a route for each of these visualizations as well. Each one first checks the appropriate permissions needed to access the visualization, which may lead to it rejecting the request. For most visualizations linked to an entity, e.g. subscriptions of a *list*, the permission check amounts to checking whether the user has permission to view the visualized entity. For the *entity activity* visualization, which visualizes the activities of every single entity, global permissions are needed, which only the administrator has, so the visualization can be accessed only by them.

If the permissions are sufficient, the route relays the request to the respective route in IVIS to obtain the visualization. Once it is returned to Mailtrain's server, it may insert some additional data about its entities if they are needed, and then return the result to the client.

Once Mailtrain's client receives the answer to the initial request, it has all it needs to embed the template. Therefore it imports IVIS' embedding module and uses the newly added `embedBuiltinTemplate` function to insert the embedded visualization into a target DOM element on the currently displayed page. This functionality is contained in the file `/client/src/lib/embed.js`.

5.5.2 Displaying The Visualizations From Mailtrain

With the embedding functionality implemented, Displaying the visualizations from Mailtrain's web pages is a straightforward task.

If a new page needs to be added for a visualization, then a new React component representing the page's content should be made, which then needs to be inserted into Mailtrain's client routes structure, so that Mailtrain can link to the page and display it. This is the case for the *entity activity* visualization, which is accessible from Mailtrain's administration dropdown menu. Since the menu is always present, but not all users have permission to access this visualization, the menu should contain the link only if the user has appropriate permissions.

The visualizations may also be added to an already existing page, in which case it is only needed to add elements with unique DOM element IDs for the visualizations to be embedded into. This is the case for all other visualizations. The *list subscriptions* visualization was inserted in a page listing the list's subscribers, the *campaign overview* and *campaign messages* visualizations were inserted in a page containing the campaign's statistics, and the *channel campaigns channel campaign contributions* visualizations were inserted in a page listing the channel's campaigns.

As stated in section 5.1.3, after the embedded templates are not needed anymore, it is required to call a stopping function to prevent any owned access tokens from refreshing indefinitely and causing a memory leak. This is why each React component containing an embedded visualization calls the stopping function in its `componentWillUnmount()` method.

5.6 Auxiliary Changes

While the JavaScript code forms most of the implemented code, Mailtrain also relies on several pieces of auxiliary code to be practically usable in production. Changes done to Mailtrain in this thesis require some changes to these areas as well.

5.6.1 Integrating Mailtrain's and IVIS' Git Repositories

One of these changes is about the organization of Mailtrain's and IVIS' repositories. Mailtrain has a distinct Git repository from IVIS but expects to use IVIS as a part of it. The inclusion of IVIS code is achieved using a Git submodule, which allows a specific commit of a Git repository to be included in a folder of another Git repository. In this case, IVIS is a submodule of Mailtrain's repository, located in folder `/mvis/ivis-core/`.

5.6.2 Configuration Files

Another change concerns Mailtrain's configuration files. There are several things that Mailtrain's or IVIS' code cannot deduce from its code alone and needs configuration files for. The biggest change is the definition of the *Mailtrain User* role in MVIS, as described in section 5.3.6. MVIS' configuration files are located in `/mvis/server/config/`, so that is where *Mailtrain User* has to be defined.

There are also some settings on Mailtrain's side which are needed be put in its configuration files located in `/server/config/`. These settings include things such as whether Mailtrain's Activity Log should log sensitive user data, or whether link clicks should be logged repeatedly for a single link, which is an option described in section 3.1.2. Other settings are mostly technical in nature, such as the information about which URL should be used to reach MVIS over HTTP.

5.6.3 Installation Scripts

To make Mailtrain easy to set up in production, there are several installation scripts included in Mailtrain's repository. The scripts are differentiated by the target operating system, and by whether the application should set up public HTTPS endpoints, or whether it should only make endpoints locally accessible. IVIS also has installation scripts, and they are structured in the same way.

In the case of both Mailtrain and IVIS, the installation has three stages. First, any prerequisite applications, such as MariaDB, are installed. The second stage installs the application itself, which includes updating configuration files to include the user's settings and installing node modules. The final stage sets up the application as a service to be executed on each startup of the host server machine.

Because of the integration, the installation scripts for Mailtrain now also need to include the installation of IVIS. Since Mailtrain spawns IVIS on its startup, only the first two phases of IVIS' installation have to be done. IVIS' installation is also configurable, so it is called with some modified variables so that it can run under Mailtrain. The main variable modification done in this way is specifying the operating system user who owns the directories to be `mailtrain`, which is a user generated by Mailtrain's installation script to run the application.

5.6.4 Documentation

Lastly, there are some minor changes to be done concerning Mailtrain's documentation. The main piece of up-to-date documentation in Mailtrain is a `README.md` file in its root directory, which contains basic information about Mailtrain's features, and an installation guide.

Adding MVIS to Mailtrain caused an increase in used HTTP endpoints, which users wanting to install Mailtrain now need to account for, so this information has been added to the `README.md` file. A similar piece of information is contained in the installation script help text, so it was changed too to inform users that they have to set up additional DNS entries if they want to use Mailtrain with HTTPS.

6. Evaluation

This chapter describes how the implemented integration was evaluated. This entails describing any tests made to verify the integration's correctness, as well as showing screenshots of the completed visualizations.

6.1 Testing

Despite Mailtrain being a fairly large project, the tests it contains are limited mostly to testing user logins and subscriptions. The relatively small scope of these tests is likely due to the technologies involved; Apart from setting up Mailtrain itself, thorough tests of Mailtrain would also require setting up a custom e-mail server to be able to test sending messages from campaigns. The tests would also possibly require a system which simulates subscribers' interactions with Mailtrain's messages. This would be very difficult to implement, which is likely why tests of these features are not included in the project.

The tests developed in this thesis have similar limitations, so any larger-scale newly implemented tests avoid having to interact using e-mail, so that setting up any extra servers is not necessary.

6.1.1 Manual Testing

A basic but important testing method is manual testing of the integration. As stated before, tests using e-mail are difficult to implement automatically, but since Mailtrain is a newsletter application, of which sending messages through e-mail is a key feature, it was at least tested manually on a small scale. Other testing methods also omit parts of Mailtrain from their runtime, so this method is important to test the functionality of the application as a whole.

Tested Modules

One group of modules tested exclusively using manual testing are existing Mailtrain modules extended to log events to IVIS using the Activity Log. The changes done to these modules are usually limited to single lines of code in a select few functions of each module. Due to the small scope of these changes, manual testing is likely sufficient to detect any errors in the new functionality.

Another module tested is Mailtrain's client, which received extensions to embed and display the visualizations from MVIS on certain existing pages, and a couple of new ones. The goal is to test that the new pages exist and that all visualizations are successfully embedded and displayed in the correct position. Due to the mostly visual nature of Mailtrain's client, this would be almost impossible to test automatically anyway, so, once again, manual testing should be sufficient.

Actions Done During Testing

To test the modules extended with Activity Log calls, every event that is logged should happen in Mailtrain at least once, so that the coverage of new code is as thorough as possible. This entails causing all events described in section 3.1, i.e. creating a list, creating a campaign and launching it, clicking a link in the campaign's message's content, etc.

It should be checked that all of the events were successfully logged in MVIS. This can be checked without the need for visualizations, using IVIS' user interface accessible from a web browser. There, signal sets in MVIS can be inspected, including records contained within them.

Eventually, all visualizations described in section 3.3 should become accessible. The pages they are located in are described in section 5.5.2. Once each visualization is accessible, its page should be checked to confirm it successfully embeds the proper visualization in the correct position on the page. The content of the visualizations is tested elsewhere, so it does not need to be checked at this point.

This test was done with Mailtrain being locally installed, and using a custom send configuration set to send campaign e-mails via SMTP using a test e-mail address located at an existing e-mail service. Several other test e-mails were created for the purpose of this test, but their count was minimal, and in theory, only a single address is needed, which can then serve both as a sender and a recipient.

6.1.2 MVIS Testing

The second testing method tests MVIS by simulating Mailtrain's input to insert test data into MVIS. Afterward, it can be tested that the data were logged properly and that visualizations resulting from them are displayed correctly. This method tests the application on data that is less diverse, but of larger size than the first method. There is also some automation involved.

Tested Modules

The modules tested are all modules of MVIS which are part of Mailtrain's project. This includes MVIS' Activity Log, modules in MVIS responsible for managing signal sets and jobs, MVIS' builtin tasks, and builtin templates in MVIS' client.

The added module responsible for testing also uses Mailtrain's Activity Log module to log data to MVIS, so the Activity Log's functionality is also tested.

Actions Done During Testing

The test is done using an executable, located in `/mvis/test-embed/`. Once it is executed, it spawns MVIS as Mailtrain would, and serves as a mockup of Mailtrain which then communicates with MVIS, including logging test data into it. The data is slightly randomized to make it feel more natural, but the state of data in key moments (e.g. when a campaign is launched, or after all data is logged) is deterministic. Due to this determinism, it can be checked if the data was logged correctly.

Once test data is logged, the test executable starts a local HTTP server, where all of MVIS' embedded visualizations are accessible. It should then be checked that the visualizations' content is displayed properly. Since this mostly concerns the visual representation of the content, it needs to be done manually. Sometimes, the visualizations may require additional data from Mailtrain, which the test also only provides a mockup for, but it provides it regardless so that the visualizations are displayed without errors.

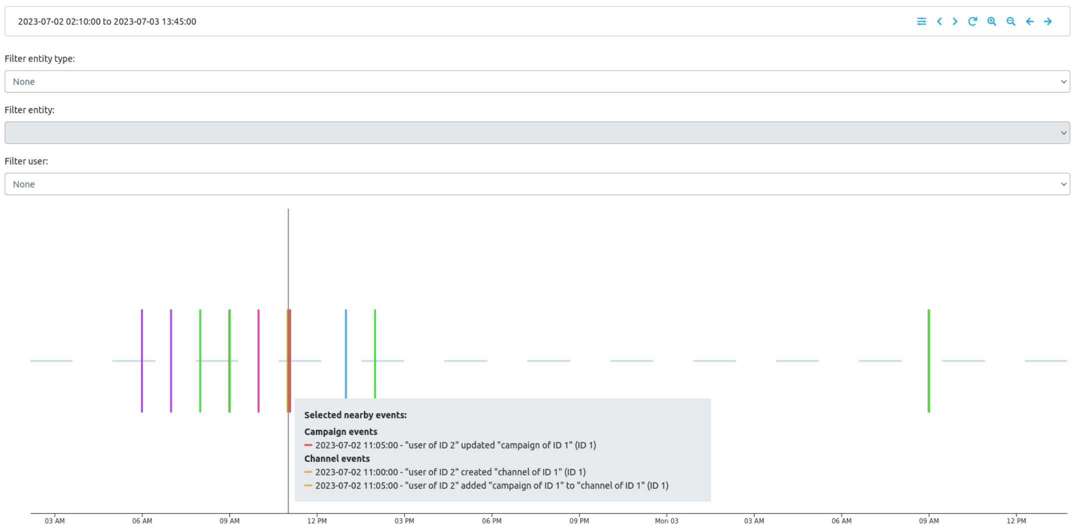
The manual testing method explained in section 6.1.2 works with only a very small amount of data, so any visualizations resulting from it look overly simple. In contrast, due to the larger amount of logged events, the testing method in this chapter produces visualizations that look much more like ones that may be seen in production. For that reason, their output is used in the next section to show how the visualizations look like.

6.2 Output Visualizations

Finally, this section shows the outputs of all proposed visualizations. The visualizations use test data generated using a test executable in `/mvis/test-embed/`.

Since the data is only a mockup, the entities are named in a way that includes their ID, e.g. ‘campaign of ID 3’. This leads to redundancy in some visualizations, which display entity names including their ID, e.g. ‘campaign of ID 3 (ID 3)’. In production, entity names are unlikely to include their ID, so this redundancy will not exist.

6.2.1 Entity Activity



2023-07-02 02:10:00 to 2023-07-03 13:45:00

Filter entity type: None

Filter entity:

Filter user: None

Selected nearby events:

Campaign events

— 2023-07-02 11:05:00 - "user of ID 2" updated "campaign of ID 1" (ID 1)

Channel events

— 2023-07-02 11:00:00 - "user of ID 2" created "channel of ID 1" (ID 1)

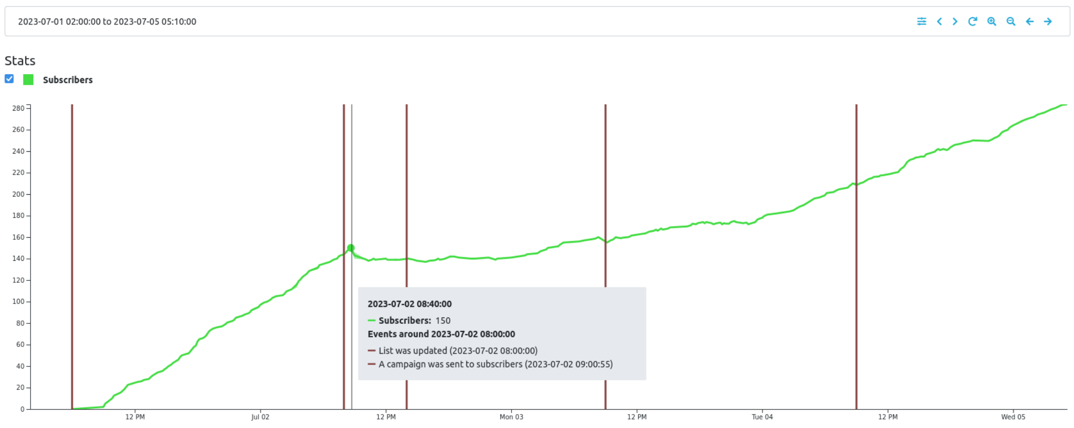
— 2023-07-02 11:05:00 - "user of ID 2" added "campaign of ID 1" to "channel of ID 1" (ID 1)

Figure 6.1: The entity activity visualization using test data

The entity activity visualization, as described in section 3.3.1 shows events organized by time, and optionally filtered by type of entity, a single given entity, or the user causing the events to happen.

Figure 6.1 shows this visualization as displayed from MVIS, including a tooltip providing extra information on events near the cursor. The dropdown of filtered entity is only enabled when events are filtered by an entity type, at which point the dropdown only shows entities of the filtered type. The events in the figure are not filtered, therefore the filtered entity dropdown is disabled.

6.2.2 List Subscriptions


2023-07-01 02:00:00 to 2023-07-05 05:10:00

Stats

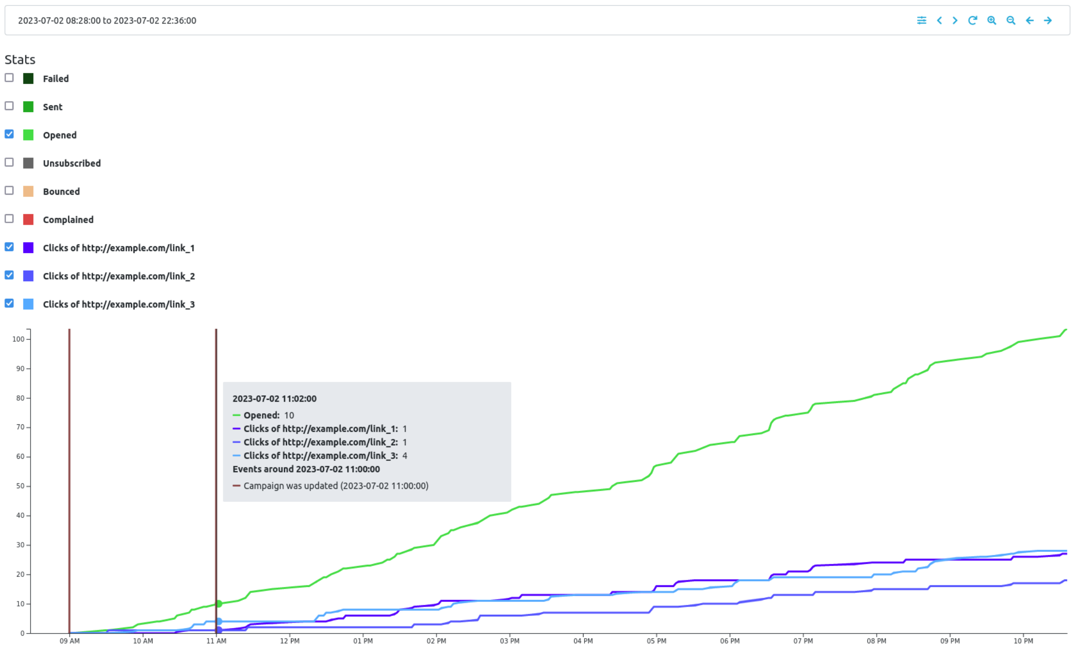
✓ 🟩 Subscribers

2023-07-02 08:40:00
— **Subscribers:** 150
Events around 2023-07-02 08:00:00
— List was updated (2023-07-02 08:00:00)
— A campaign was sent to subscribers (2023-07-02 09:00:55)

Figure 6.2: The list subscriptions visualization using test data

The list subscriptions visualization was described in section 3.3.2. Figure 6.2 shows the visualization from MVIS displaying subscription counts of a list from the test data. The screenshot includes a tooltip with extra information on the precise subscriber count at the selected point, and additional information about nearby activity events of the list.

6.2.3 Campaign Messages


2023-07-02 08:28:00 to 2023-07-02 22:36:00

Stats

☐ 🟩 Failed
☐ 🟩 Sent
✓ 🟩 Opened
☐ ⬛ Unsubscribed
☐ 🟧 Bounced
☐ 🟥 Complained
✓ 🟦 Clicks of http://example.com/link_1
✓ 🟦 Clicks of http://example.com/link_2
✓ 🟦 Clicks of http://example.com/link_3

2023-07-02 11:02:00
— **Opened:** 10
— **Clicks of http://example.com/link_1:** 1
— **Clicks of http://example.com/link_2:** 1
— **Clicks of http://example.com/link_3:** 4
Events around 2023-07-02 11:00:00
— Campaign was updated (2023-07-02 11:00:00)

Figure 6.3: The campaign messages visualization using test data

The campaign messages visualization was described in section 3.3.3. Figure 6.3 shows its MVIS visualization. The visualization shows various metrics of the campaign, but it is generally not meant for all to be displayed at the same time, so only opened message counts and link clicks are enabled in the screenshot.

Similarly to the previous screenshots, this one also includes a tooltip with precise counts of line values at a point hovered over with a cursor, as well as information about nearby campaign events.

6.2.4 Campaign Overview

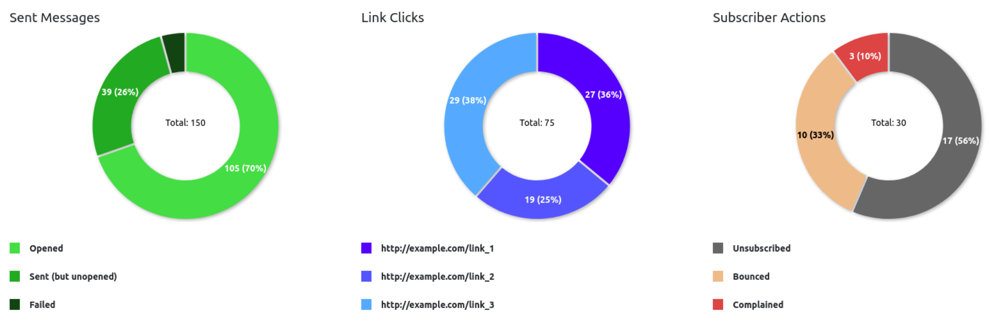


Figure 6.4: The campaign overview visualization using test data

The campaign overview visualization, described in section 3.3.4, shows a summary of the current state of a selected campaign in the form of three pie charts. Figure 6.4 shows a screenshot of this visualization.

6.2.5 Channel Campaigns

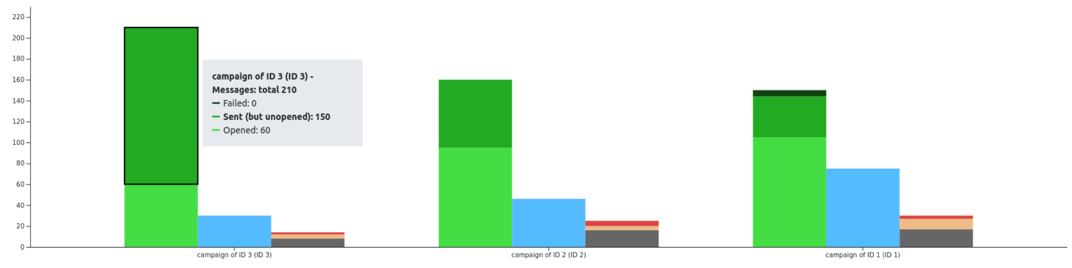


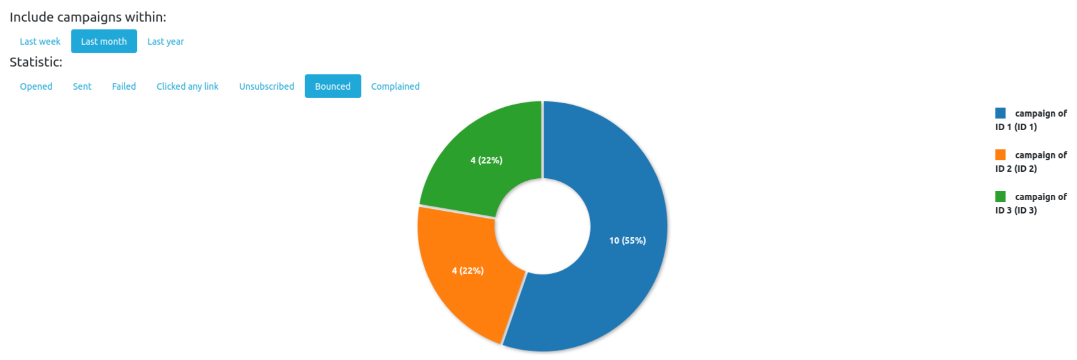
Figure 6.5: The channel campaigns visualization using test data

The channel campaigns visualization, described in section 3.3.5, displays the performance of its recent campaigns with each campaign being represented as a group of three columns. Figure 6.5 shows a screenshot of this visualization. Although the limit for the number of displayed campaigns is larger, The test data used only generates three campaigns, so the visualization stretches them to the size of the whole bar chart.

The visualization also shows a tooltip when a segment is hovered over, which gives the user information about the column and its group.

6.2.6 Channel Campaign Contributions

Lastly, the channel campaign contributions visualization, described in section 3.3.6, displays a pie chart with selected metrics of campaigns within a given interval. Figure 6.6 shows a screenshot of this visualization, which compares


Include campaigns within:

Last week | [✓ Last month] | Last year

Statistic:

Opened | Sent | Failed | Clicked any link | Unsubscribed | [✓ Bounced] | Complained

4 (22%)
4 (22%)
10 (55%)
🟦 campaign of ID 1 (ID 1)
🟧 campaign of ID 2 (ID 2)
🟩 campaign of ID 3 (ID 3)

Figure 6.6: The channel campaign contributions visualization using test data

the number of messages that bounced back of campaigns created within the last month of viewing the visualization.

7. Conclusion

The goal of this thesis was to extend Mailtrain with logging, processing and visualizing data of its entities, which includes basic data about all entities being modified by Mailtrain's users, and more detailed data about the performance of Mailtrain's lists and campaigns. This was meant to be achieved using the IVIS framework, which would be integrated into Mailtrain and provide most of the data processing and visualization functionality, limiting most of the work to designing a data logging system, data storage schemas, and visualization templates for IVIS. The goal was eventually fulfilled, but it proved to be more complicated than expected.

The implementation included both client-side and server-side code. The work entailed scanning the majority of Mailtrain's server code for potentially loggable events. Additionally, to make the integration work properly, IVIS' code also had to be extended in multiple ways. Getting to understand the projects' codebases was a rather difficult task due to their very sparse documentation, combined with their code being written in a dynamically typed programming language. Lastly, both Mailtrain and IVIS have not had their software updated for a long time now, with the versions of technologies that they use being outdated to the point where up-to-date interfaces and documentation of several libraries used by the projects have noticeably changed. Among other things, this also makes the projects capable of running only on relatively old versions of operating systems, which may soon no longer be supported.

That said, ultimately, the integration was successfully implemented, and the final result looks serviceable. The implemented logging system logs a majority of Mailtrain's events which modify Mailtrain's state in some way. There are many more logged events that the current visualizations use and it is likely that most event data, which visualizations implemented in the future will need, are already sufficiently logged. The implemented visualizations make a good amount of previously hard-to-access or inaccessible information easy to see and analyze, which should assist Mailtrain's users who wish to improve the performance of their newsletters. The integration of IVIS into Mailtrain itself has also made a lot of new data processing and visualization-related functionality accessible to Mailtrain, which Mailtrain's future features will hopefully be able to make good use of.

IVIS has also received its share of extensions in this thesis. While they are much less far-reaching than in the case of Mailtrain, they should still help IVIS broaden the number of situations in which it can be used.

7.1 Future work

The last thing to explain is how the work done in this thesis impacts future work on Mailtrain. Implementing the integration of IVIS into Mailtrain has brought new possibilities for features that may be implemented into Mailtrain in the future. There is also space for further refining and improving Mailtrain's already existing features.

7.1.1 Improving Existing And Adding New Visualizations

The existing visualizations provide a decent amount of information about the activity of Mailtrain's entities, and performance of the its lists, campaigns, and channels. So far the visualizations were not used in production. It may be useful to gather some user feedback and update the visualizations accordingly, so that using the visualizations is a better experience.

Furthermore, since the system for embedding IVIS templates in Mailtrain already exists, the majority of work done when adding a new visualization only amounts to programming the visualization template. Therefore, if Mailtrain's users have reasonable requests for new visualizations, it should be relatively easy to implement them to be displayed from Mailtrain.

7.1.2 A System For Custom Tasks And Visualizations

As stated before, Mailtrain has a built-in system for creating custom user tasks for processing data, called reports. With the functionality brought to Mailtrain by IVIS, it may now be possible to realize reports completely using IVIS' features.

The new IVIS-based report system would need to allow Mailtrain users to create their own IVIS tasks and jobs, which they would then be able to run. This would lead to having various user-made signal sets, which would require further refining permission mapping from Mailtrain to IVIS, so that it is clear which Mailtrain users have access to which MVIS signal sets. The signal sets themselves, mostly ones related to entity activity, would also need to be split by single entities to be able to restrict permissions to non-administrator users, if the signal sets are to be accessible to the users' tasks.

If this new report system is implemented, it could also be rather easily extended with user-made visualizations, by also allowing Mailtrain users to create their own IVIS templates and panels.

7.1.3 Updating Mailtrain And IVIS Libraries

The versions of libraries that Mailtrain and IVIS use are often outdated, sometimes by years from the latest version, which, at this time, leads to some minor development problems, like having to use an interface or a library function that is no longer used in the latest version, but for which an alternative does not exist yet in the old version, which means that if the libraries are updated, the code will have to be changed.

Due to this state of the projects, it is only possible to run them in old operating system versions that are almost no longer supported. Despite their newer features, this low compatibility means fewer people are likely to use them. Furthermore, soon it may be impossible to run Mailtrain or IVIS anywhere.

This is why, despite being mentioned last, updating Mailtrain and IVIS is a very high-priority task and it is very likely that completing it will become the next goal in Mailtrain's and IVIS' development.

Bibliography

[1] Mike Bostock. D3: Data-driven documents. URL `https://d3js.org/`, Accessed July 8th 2023.

[2] Axios contributors. Axios. URL `https://axios-http.com/`, Accessed July 16th 2023.

[3] Knex contributors. Knex.js. URL `https://knexjs.org/`, Accessed July 8th 2023.

[4] Webpack contributors. Webpack. URL `https://webpack.js.org/`, Accessed July 8th 2023.

[5] ZoneMTA contributors. ZoneMTA GitHub repository. URL `https://github.com/zone-eu/zone-mta`, Accessed July 8th 2023.

[6] Free Software Foundation. GNU general public licence. URL `https://www.gnu.org/licenses/gpl-3.0.en.html`, Accessed July 8th 2023.

[7] MariaDB Foundation. MariaDB, accessed july 8th 2023. URL `https://mariadb.org/`, Accessed July 8th 2023.

[8] OpenJS Foundation. Node.js. URL `https://nodejs.org/`, Accessed July 8th 2023.

[9] OpenJS Foundation. Node.js Child process. URL `https://nodejs.org/api/child_process.html`, Accessed July 8th 2023.

[10] OpenJS Foundation. Node.js Crypto. URL `https://nodejs.org/api/crypto.html`, Accessed July 16th 2023.

[11] Amazon Inc. Amazon SES, accessed july 8th 2023. URL `https://aws.amazon.com/ses/`, Accessed July 8th 2023.

[12] Docker Inc. Docker. URL `https://www.docker.com/`, Accessed July 8th 2023.

[13] Facebook Inc. React. URL `https://react.dev/`, Accessed July 8th 2023.

[14] Canonical Ltd. Ubuntu. URL `https://ubuntu.com/`, Accessed July 8th 2023.

[15] MailtrainOrg. Mailtrain GitHub repository. URL `https://github.com/mailtrain-org/mailtrain`, Accessed July 8th 2023.

[16] Elastic NV. Elasticsearch. URL `https://www.elastic.co/elasticsearch/`, Accessed July 8th 2023.

[17] The CentOS Project. CentOS, accessed july 8th 2023. URL `https://www.centos.org/`, Accessed July 8th 2023.

[18] SmartArch. IVIS CONCEPTS.md file. URL `https://github.com/smartarch/ivis-core/blob/master/CONCEPTS.md`, Accessed July 8th 2023.

[19] SmartArch. IVIS-CORE GitHub repository. URL `https://github.com/smartarch/ivis-core`, Accessed July 8th 2023.

[20] European Union. General data protection regulation. URL `https://gdpr.eu/`, Accessed July 16th 2023.

List of Figures

A. Attachments

A.1 Mailtrain Source Code

The source code for Mailtrain is attached to the electronic version of this thesis in the form of a git repository. The branch `mvis-dev` contains the integration implemented in this thesis. All changes to files in all commits to this branch are a part of the thesis' implementation.

At the time of writing, the source code is also accessible online at `https://github.com/PhiStCZ/mailtrain/tree/mvis-dev`, and will likely remain accessible until the changes are merged into Mailtrain's GitHub repository [15].

A.2 IVIS-CORE Source Code

The source code for IVIS is attached to the electronic version of this thesis in the form of a git repository. The branch `mailtrain-dev-latest` contains the extensions implemented for IVIS in this thesis. All changes to files in all commits to this branch are a part of the thesis' implementation.

Similarly to Mailtrain's source code, this source code is also accessible online at `https://github.com/PhiStCZ/ivis-core/tree/mailtrain-dev-latest`, and will remain accessible at least until the changes are merged into the IVIS-CORE GitHub repository [19].