The importance of data validation and parsing when working with external data sources

Alexander Gallen

Master's thesis in Computer engineering
Supervisor: Annamari Soini
Åbo Akademi
Faculty of Science and Engineering
February 2024

Abstract

Working with data from external sources often revolves around combining data from multiple sources to analyse or process it in new ways to generate value. The developer is often faced with uncertainties in the retrieved data due to lacking or non-existent documentation. In this thesis the praxis of parsing, validating and transforming the data will be explored in depth to showcase how these challenges can be tackled to obtain a robust data fetching pipeline, and keep the backend free from unknowns and extra validation that is not part of the business logic that is being developed.

The focus will be on TypeScript and more specifically the Zod library. The language and technology were chosen based on their popularity, relevance in the modern programming field, and previous experience in production environments by the author. The data that is examined to showcase the benefits and a minimal setup to achieve a robust parsing and validation flow are ledger receipts in Procountor, a widely used Finnish bookkeeping system. By implementing these best practices and data fetching techniques it is possible to eliminate unknowns from the backend of the application under development. All the parsing, validation and transformation of the external data are handled in a single place of the data processing pipeline and are in their entirety extracted from the business logic of the application. These functions can also be re-used to manifest uniform practicalities throughout the application to easily scale the system to facilitate data from more integrations. Handling of common problems such as miss-matching data types and different ways to portray certain information such as “nullish” values or dates becomes trivial and uniform and the type of data that enters the backend, business logic of the application is fully known and robust.

Taking advantage of these learnings will allow developers to increase the productivity, robustness and maintainability of their applications, specifically when dealing with large or complex data from third party applications. Typical issues such as various string representations of types e.g. dates or currencies

become an issue of the past and the types of all the fields can be managed uniformly in the application to allow the developers responsible for the business logic, data analysis, or data processing to focus on the problem at hand, instead of the wrangling of the unknown external data to deal with unknown edge cases.

Table of Contents

1 Introduction

In modern programming, it is very common to sooner or later have to deal with data from external sources. Regardless of whether this data arrives from files, external APIs or loosely integrated legacy systems, one will at some point have to deal with the fact that it is not possible to be certain about the state or structure of the data when it arrives to the system.

Most modern programming languages are either statically typed or at least provide ways to give type hints during development to reduce iterations and development time [1]. This solves most of the problems when working with complex data structures, but it falls short when the actual type or structure of the data that enters the system is not fully known.

The naive way of solving the issue is to assume the type to the best of one's knowledge and then validate the data when required at different stages of the backend processes of the system. Over time, this will lead to a complex structure of data validation at different places in the code base which, in turn, increases the complexity and decreases the maintainability and debuggability of the code [2]. Furthermore, the structure of the data that enters the system is at no point in time fully understood, typed, or known.

This can be solved by parsing the data that enters the system, so that the structure of the data one is dealing with is always known. By applying a parsing function at the point of entry for the external data, one is able to pinpoint the exact structure of the data that is being dealt with, and for statically typed languages, the type of the parsing function can be reused throughout one’s system to achieve full end-to-end type safety for the external data.

The goal of this thesis is to investigate all the different types of issues one usually has to deal with when working with complex external data sets and how parsing functions can be implemented to ensure full end-to-end type safety for all the data that is being processed in the backend of the system. Any individual or team working with complex data sets that arrive from external sources, where the type of the data cannot be fully relied on or known, will hopefully find something new and enlightening to take away from this thesis that can increase the productivity and robustness of the development teams and the code base they deal with.

For the purpose of this thesis, I will focus on showing how to solve these problems in TypeScript. As a statically typed language, TypeScript can already deal with full type safety with internal data, but since the types of external data will only be validated at runtime, it is required that the data is parsed when it enters a system. A schema validation library can be used to handle this. The most popular ones for TypeScript are Zod and Yup, with Joi being a third option native to JavaScript that also has a TypeScript version. In this thesis, the popular Zod library [3] will be presented and used. Zod provides an easy-to-read syntax to define validator functions with infinite possibilities of refinement and transformation to deal with the worst possible source data, so that the data can be conformed to one's needs. It allows for easy type extraction from the validator function, so it can be reused throughout the system [2]. In the interest of keeping the explanations and examples concise, the thesis will focus on dealing with data from external APIs, but this is of course applicable to data from any external systems, and all takeaways and paradigms offered and explained here can be applied to any such data.

2 Programming languages

The choice of programming language and how they deal with typing has a significant impact on how well the type of structured data can be enforced, and they assist the developer during the development process of the system. An overview of the key aspects of the different types of typed languages is presented below.

2.1 Statically typed languages

Statically typed languages are languages where variable types are determined at compile time and cannot change during runtime. Some popular examples include Java, C++, C#, Rust, and TypeScript.

2.1.1 Key Characteristics

Most statically typed languages share the following key characteristics. Types are checked at compile time, leading to type mismatches or errors being caught before the program is executed, minimising the chance of errors being caught late in the development process or in production. Explicit type declarations, which require the developer to specify the type of the variable or the parameters and return values of functions, assist the development of the system by providing useful support to the developers and ensuring that trivial errors are caught as early as possible. Some modern languages, such as TypeScript, allow for partially implicit variable declaration which can speed up development but is also more error prone when the implicit declaration is not what one expects. Type safety improves code readability and greatly enhances the developer experience when working in larger projects with multiple developers and older code bases. Statically typed languages also often compile to more efficient code, since the compiler has complete knowledge of the types. This can also greatly vary between languages. TypeScript, for example, compiles the vanilla JavaScript and has no knowledge of types.

2.1.2 Benefits

The nature of statically typed languages offers some general advantages. Errors in data are caught at compile time, reducing the chance of runtime crashes and errors, which are often more difficult to track down and correct. Code readability is greatly improved, since explicit type declarations function as a kind of self-documentation, helping other developers to understand how functions and variables are and can be used. These two major advantages combined also lead to greatly increased possibility to refactor the code base down the line, since many of the errors are caught already during the refactoring phase, with only more specific business logic-related issues having to be unit tested or manually tested more in depth.

2.1.3 Issues

While there are several benefits with statically typed languages, they also present some common challenges. When the data sets can include multiple data types or dynamic elements, the code to handle these types can be very verbose and unintuitive. The up-front development cost can be increased since types must be explicitly written, which for larger data sets can be extremely time consuming. Dynamic behaviour might not be so well supported, e.g., dealing with unstructured data or highly flexible data structures. Last, but not least, the risk for overhead might grow considerably when defining explicit types for complex or large data structures.

2.2 Dynamically typed languages

Dynamically typed languages are languages where variable types are determined at runtime, and the types can change during the execution of the program. The most popular dynamic languages today are Python, JavaScript, Ruby, and PHP.

2.2.1 Key Characteristics

Types are only checked during runtime, and the types can change throughout the execution of the program. Variables do not have explicit type declaration, but rather the type is determined by the value the variable holds. Dynamically typed languages are very flexible and have low overhead from a developer's perspective due to not requiring explicit definition of variable or function types.

2.2.2 Benefits

The above characteristics allow for rapid development and iteration. The code can be very lean and concise with full focus given to logic. Dynamically typed languages are generally well suited for programs where the data structure is not known in advance or may change during runtime, such as data processing and scripting.

2.2.3 Issues

The lack of type safety leads to type-related errors only surfacing during the execution of the program which, in turn, leads to more challenging debugging and increased development cost and time. The absence of explicit types also causes the code to be less self-documenting, which requires developers to rely more heavily on comments and external documentation. Code maintenance may become increasingly difficult once the program grows, making it difficult to keep track of variable types and unexpected behaviour when the data structures are large or complex.

2.3 Gradually typed languages

While languages are commonly grouped into the two major categories statically typed and dynamically typed, for the purposes of this thesis we shall also briefly describe the characteristics of a third type of paradigm or typing, which can be achieved, at least partially, in most popular modern programming languages. This approach is commonly referred to as “Gradually typed” or “Optionally typed” languages, sometimes also referred to as “Hybrid typing”. Some examples of this would be TypeScript, with a looser typing approach allowing the use of the *Any* type when strict typing is not necessary, and Python, by partially using the *Mypy* static type checker.

2.3.1 Key characteristics

Gradual typing allows the developer to seamlessly blend static and dynamic typing in a single language.

2.3.2 Benefits

The developer has the flexibility to gradually add explicit types to parts of the codebase where strict types are required. This aims to strike a natural balance between the reliability of static type checking and the flexibility of dynamic typing. Most programs have different requirements on the reliability of data structures, depending on the actual data that is being processed and how it is being processed. Not all datasets might be processed so heavily, and might just flow semi-loosely through the program, thus requiring minimal to no type safety, while still functioning properly in all situations. A common example of this might be reading data from an input or a function and storing it in a file.

2.3.3 Issues

The main issues with gradually typed languages are the difficulty to enforce best practices and not letting the developers grow “lazy”. The rules and best practices must be set up in such a way that it is easy to understand when and where to explicitly type the data structures, and not just omit the types for the sake of indolence, eliminating the benefits of gradually typed languages.

2.4 Differences

Statically typed languages require more up-front development but lead to earlier error detection, improved code readability, optimised performance [1] and greater refactoring support.

Dynamically typed languages offer rapid development and flexibility but can cause issues due to type-related bugs, lack of explicit documentation and code maintenance, and difficult debuggability when dealing with large or complex data structures.

Gradually typed languages attempt to bridge the gap between statically and dynamically typed languages by allowing developers to add explicit static types when necessary while retaining the advantages of dynamic typing in parts of the program where explicit declaration of types is not necessary.

2.5 Criteria to consider when weighing the different approaches

There are several different criteria to consider when deciding what kind of programming language to use for one's project. Apart from the non-technical aspects, such as development team expertise and personal preferences, and business specific reasons, such as legacy systems and language specific libraries that simplify the problem at hand, there are also several technical criteria that depend on the nature of the language used that should be considered. Some of the general aspects are described more in detail below to give a broader understanding of what is worth considering before starting the development of a software project.

2.5.1 Error handling

If the project is likely to grow large and include a large amount of complicated business logic and large complex data structures, it will become increasingly important to catch errors and bugs in the system at early stages. Many of the errors related to complex data structures can be caught already during development, during compile time, if a statically or gradually typed language is to be used.

If the bounds of the project are well known in advance and the functionality and scope are relatively limited, the advantages of choosing a statically typed language for a project might become outweighed by the rapid iterative development allowed by dynamically typed languages.

2.5.1.1 Cost of catching errors early vs. late in the development process

Detecting and resolving errors early in the software development lifecycle have several advantages which can significantly improve project efficiency and productivity. If errors are identified early on during development, the cost of correcting them is relatively low [4].

Another key benefit of catching errors and edge cases early is not to let uncertain states flow through the codebase. This is one of the aspects where explicitly typing complex data structures can greatly reduce development cost in the long run.

Finding the different scenarios that need to be dealt with early during development also pushes the structure of the code to conform and deal with all these special cases early on, reducing the possibility of requiring a major rewrite of the codebase later in the development process, when the cost of having to do so would be drastically increased.

2.5.2 Development time

The development time and cost of a project can be divided into three major categories: First is the up-front development cost that is required to develop a minimum viable product for the customer. The second category is iterating an initial deployed production version. The third is fixing bugs, maintaining the system, and adding new features as required. These categories unquestionably vary immensely depending on the size and the specifications of a project.

It is, however, important to try to recognise the attributes of the project that is about to be developed to try to determine in which of these categories much of the development time will be spent. If the bulk of the work will be spent in the initial development phase and the complexity of the data that is dealt with is relatively low, there is a probability that a dynamically or gradually typed language is favourable, since the extra benefits of complete type safety will be lost due to the extra overhead and initial development time this requires.

However, if the better part of the work will go into continuously improving the software over a longer period, or if the project is expected to grow large or complex in size, a statically typed approach might be preferred.

2.5.3 Reliability

The significance of a reliable and robust application does not so much depend on the size of the codebase or the complexity of the data structures that are dealt with, but more so on the industry, or the actual tasks that the application is touching. Regardless of the complexity of the data being worked with or the size of the application in production, these kinds of applications usually have an immense cost tied to errors and discrepancies that occur when in production. It might even affect people's lives, such as the cases with medical or legal applications. In these instances, the reliability of the software might outweigh all other key factors combined, and the only reasonable approach might be to choose a statically typed approach where much effort and time are invested to ensure that every step of the application is thoroughly tested before entering the hands of the end users.

2.5.4 Maintenance

The time spent on maintenance in a software development project often depends on the initial scope of the project and a few other factors of varying importance. Factors such as the existence of third parties that are involved in the data flow, needs for feature development, and the rigidity of the scope of the project all play a major role.

Third-party applications, be they sending or receiving parties, might lead to continuous demands for feature development or bug fixing. These can include changes in the data structure from a third-party API, new functionality that is available to be incorporated into the application due to improvements in a third-party application, or a sudden change in some API or data source that completely breaks the current functionality in the system at hand.

Sometimes the scope of a project is quite ambiguous in the beginning, and only after some time in production with continuous feedback from the end customer, the final form of the application starts to take shape.

The optimal solution to these kinds of challenges depends greatly on what kind of data and system is being dealt with. See sections 4.2 and 4.3 below for a more in-depth explanation of issues to consider. Broadly speaking, however, if the correctness of the system is not so crucial and the data structures are relatively simple a more dynamic approach might be beneficial, while otherwise a statically, explicitly typed approach might lead to great time savings down the line.

2.6 Conclusion

The choice between whether to use a statically typed language, dynamically typed language, or gradually typed language depends on several factors, including the nature of the project, personal preferences, and specific requirements.

Statically typed languages tend to be more beneficial when strong type safety is crucial, performance matters, the codebase might grow large and complex, or there might be a frequent need for modifications and refactoring. Some examples of such projects might be financial software or critical system components where high type safety is crucial to catch any unintended behaviour or discrepancies already during compile time. Others are game engines or real-time systems that are highly performance-critical, or any large or vaguely scoped project where the code base might grow indefinitely, and the requirements might change over time when the program is already in production.

Dynamically typed languages can be used when rapid prototyping is essential and flexibility is needed, while scripting or automating manual tasks, in small or medium-sized projects, and during web development and prototyping machine learning models. [5]

To get the best of both worlds a gradually typed or hybrid approach can be used. This does, however, add its own challenges, relying more heavily on the developers to make the correct decision about when to add or not to add explicit types to build a robust and flexible system.

Ultimately, there is no definite correct answer, and the choice of language should be considered based on the specific needs of the project at hand, the expertise of the development team, and the trade-offs between development speed, performance, and maintenance and robustness. The choice is often not so clear-cut, and the requirements differ from project to project.

3 Complex data structures

Most applications rely on one or multiple more complex data sets that are fundamental to the functionality of the application. These datasets are commonly frequently accessed or created by the users of the application. Since they play such a crucial role in the application it is important to manage and structure the data accordingly to ensure that the functionality surrounding the data is both easily maintainable and robust.

3.1 Determine importance of structural integrity in data

Oftentimes a system will consist of multiple types of data which might have different requirements on the structural integrity. There are datasets that are critical for the overall functionality of the system that require very precise definition, and then there are less crucial components which might just flow through the system without much input or modification from the application. To optimise the robustness while keeping development time and complexity to a minimum it is important to distinguish between these different types of data. Managing to define the importance of different data well will, in addition to keeping down development costs, also accumulate benefits over time by keeping the codebase clean and endorsing flexibility in the system where robustness is not required.

3.1.1 Different types of data

Not all data are equal when it comes to deciding on the importance of strict typing, validation, or parsing. There might be very simple small data structures that are crucial to the functionality of the application or that require a very strict structure within major functions in the software. These data structures might seem simple at a first glance but, depending on the language one is dealing with, edge cases might still arise that are difficult to foresee up front. On the other hand, there might be large complex data structures, which only flow through the system from external APIs and are pumped out on the other end to the client, without being processed at all within the system. While these datasets might seem important at first, the validity, or more precisely the structure, of this data might be quite insignificant for the system.

Technical aspects of the data and how different types of data can present their own challenges in particular programming languages due to language-specific quirks and limitations will not be described here, but rather the data structure itself and what role the data plays in the system. The language-specific limitations are out of the scope of this thesis, and these are more of programmatic questions, while the focus here will be on the general concepts.

While contemplating whether the data one is dealing with requires explicit typing, there are a few factors that should be considered: the importance of the data structure, the role of the data in the system, and how the data is being processed in it. Below we will review a few common scenarios and types of data to give a better understanding of how one should approach the problem, helping one make an informed decision about how to deal with said data sets.

3.1.1.1 Critical data

Critical data is usually data that in some manner, or form, is essential for the correct functioning of the system. This data can be characterised by it playing a key role in the major or critical functionality of the system (e.g., credentials, forms), and main data structures that the system is dealing with and processing (e.g., invoices in an accounting software).

3.1.1.2 Non-critical data

Non-critical data can be any data that requires very limited to no processing within the application. This data, even though it might be large and complex, often does not require a strict structure. An example might be a reporting application which shows aggregates of complex data that are fetched via an external API. While the data might seem complex it can be displayed on a UI with minimal processing, leading to a much more flexible system without the need for excessive typing and data processing, which would be error prone depending on the degree of reliability of an external system.

3.2 Dealing with external data sources

Up to this point, we have mainly touched on the benefits of dynamic vs. statically typed languages and the hybrid approach of gradually, or optionally, typed languages. While these paradigms all have their place depending on the application that is being developed, there is a scenario that typing is unable to deal with, and that is common in most modern applications.

This scenario concerns data arriving from external sources. This encompasses data that is fetched from external APIs and various types of user input (e.g., command line interface, loosely defined UIs, files, and web-scraped data). While the structure of this type of data usually can be partially known, as is the case with external APIs, where there is documentation that at least to some extent describes the structure and type of the data that is fetched or uploaded, one can be almost certain that the exact structure of the data is unknown. When this external data is critical for a properly functioning system, it suddenly presents several substantial challenges. The type of this data can only be assumed, thus rendering explicit types only vaguely helpful, so when typing alone is unable to solve the issues that this kind of data presents, a different solution must be explored.

The approach is the same whether one is working with a statically or dynamically typed language, and that is parsing the data when it enters the system. This ensures that the type of the data is entirely known and that no surprises arise down the line in a different part of the system in production. This approach also gives the added benefit of being able to provide the types (for statically typed languages) directly from the parsing function used, either by defining them by custom or by using third-party libraries which provide easier-to-use syntax and added functionality around these kinds of parsing functions.

3.2.1 Data types

In addition to the inherent problem with uncertain data structures, there is also the problem of the actual types of the data that are received from external sources. External data from files, the command line, or a legacy user interface will most likely always initially be interpreted as text, i.e., strings, when entering the backend of an application. Data from APIs, in turn, can take many different shapes, with edge cases being treated in multiple different ways depending on the underlying application. How this data should be treated to some extent depends on the language of the application and its limitations, but in general the problem cases listed below are generic and should be considered when building a robust application that relies on external data.

For the purpose of keeping this chapter concise, the specifics of different data sources will not be covered. Instead, the focus will be on the specific data types and in what format they are usually fetched, as well as key aspects to consider when validating and parsing this kind of data.

The most common data types that are dealt with on an individual field/property level when dealing with any more complex data structure from external sources are the following: text, numbers, dates, Boolean, lists, and enumerations. In addition to these generic types for individual fields, the fields might, or might not, also be optional.

In the following subchapters, the special cases of each data type will be more thoroughly examined. The data will be considered from the perspective of what the type of the data, or field, is considered to be, and not what it is returned as from the external data source. This is an important distinction that will greatly increase the robustness of the system under development.

3.2.1.1 Text

The most common data type that is encountered when dealing with data is text, commonly referred to as strings in most programming languages. Data that is considered a string will almost always arrive from any external source as a string, regardless of it originating from user input, files, or external APIs.

The only circumstance where this would not be the case is some identifier, or identification number, i.e., fields which might commonly be returned from APIs as numbers. The decision whether these kinds of fields should be considered text or numbers is individual and both can be correct depending on the format of the data. Most notably, they could still be treated as numerical data if the size of the numerical value is of interest.

The most common edge case for text data is the empty string, or "", which is quite commonly returned from APIs and should be treated as a nullish value [6] in the system under development. Leaving empty strings in the system will greatly decrease the robustness and might lead to several unexpected problems which might be difficult and expensive to track down.

3.2.1.1.1 Strictly formatted strings

While it is usually enough to just validate and parse text as a generic string, there are a few cases where this might not be enough. One of these cases is the generic enumeration-like type, which will be examined more in depth in a separate section below, but additionally there are also a vast number of types of data that could, and possibly should, be strictly formatted.

These types of fields are generally recognised by them having a specific pattern that they must conform to in order to be valid. Common examples of this would be phone numbers, emails, personal identification numbers, business identification numbers, and so on. If this type of data and its validity is crucial for the functionality of the system at hand, it might be beneficial to run it through a parsing function that validates that the data is, in fact, correct.

Since the topic of this thesis is not strictly about parsing data, these different examples will not be examined more in depth, but it is important to understand the existence of this type of data and when it might be beneficial to strictly parse the data, to ensure the validity of it when it enters the system.

3.2.1.2 Numerical

Numerical data can take several different forms and should be treated slightly differently depending on the programming language that the system is built with. In general, numerical data can be divided into three main categories: generic numerical data, percentual values, and currency-like data.

3.2.1.2.1 Generic numerical data

In general, numerical data usually arrive as either some type of numerical value (e.g., integer or floating-point number) or a string. The most common issue that one is faced with is when numerical values are fetched as strings from third-party systems and the formatting of the number can differ based on the language [7]. These differences usually arise when the numbers grow large (i.e., to the thousands), or when the numerical value is a floating-point number.

With floating-point numbers, the most commonly found difference is how decimal points are separated from the whole numbers. For many English-speaking countries, the decimal points are separated by a dot ".", while for most other European languages they are separated by a comma ",". Many of the programming languages expect the decimals of floating-point numbers to be separated by a dot, which in practice means that the comma has to be replaced by a dot before the string is parsed to a numerical value.

Large numbers (i.e., values over 1000) might include additional country specific formatting. The most common ones encountered are either a dot ".", a comma "," or a space between every three whole numbers, separating each thousand to make the number more readable. These also must be removed in order to properly parse the string into a numerical value in most programming languages.

It is very important to be sure about the specific formatting of the numbers that are fetched to the system, since improper formatting prior to parsing might lead to incorrect values existing in the system. This is specifically important when working with large floating-point numbers, where commas and dots might coexist, requiring specific formatting to ensure that the string will be parsed correctly.

For example, with German formatting (e.g., 4 294 967.295,000) the spaces and dots would have to be removed and the comma replaced by a dot, while in American formatting (e.g., 4,294,967,295.00) the commas should be removed.

3.2.1.2.2 Percentages

Percentages follow the same rules as generic numerical data with the possible further addition of a percentage symbol "%". Except for the possible need of removing the percentage symbol for the percentage to be converted to a numerical value, the same rules apply as for the generic numerical data type.

3.2.1.2.3 Currency-like data

The same principles apply for currency-like data, but they also include a few other important nuances that should be considered. The actual currency symbol might be included, and while the currency information usually exists separately in the data it is important to make sure that this is the case, since this can be important information that should be stored if it is not the case. The other important difference is the number of digits that exist after the decimal point, i.e., 1.21 and 1.211 might be identical when the number is rounded to an accuracy of two decimals, but an external resource might have an issue with the rounding that occurs.

While the number of digits after the decimal point might not always be crucial, there are some specific use cases that can cause difficult-to-track-down issues that might arise from how some programming languages deal with floating-point numbers (i.e., most use IEEE 754 [8] which can lead to inconsistencies when portraying decimal numbers due to the nature of how these values are stored) where elementary arithmetic operations might not lead to the expected outcome. A simple example would be in JavaScript where the arithmetic operation 0.1+0.2 ==! 0.3 due to the nature of floating-point values and how they are stored in memory, leading to some values not being able to be precisely portrayed. Inconsistencies such as this one can be very difficult to track down in production and might lead to several unexpected states in the system. There are numerous ways to circumvent this problem. One is to simply round the numbers to 2 decimal places before and after doing operations on them and then compare those values. The other is to store decimal numbers in separate variables (on the backend) and columns (in the database), completely circumventing the issue of floating-point numbers altogether. What this means in practice is that the integer part and the fractional part of a decimal value are stored as integers as separate values, completely eliminating the issues that can be encountered while dealing with floating-point values. This is a common practice in many financial systems where exact portrayal of the data is of significant importance.

3.2.1.3 Dates

While some structures, such as XML, support “date” as a property type it is also common to encounter dates as strings and, depending on the language, they might have to be formatted as strings before being parsed back to dates for storage. One example of this is JSON data structures. Dates can exist in a multitude of different formats and should always be investigated on a case-by-case basis. The most common format is the ISO-8601 [9] standard, i.e., dates portrayed as “2023-10-24” and dates with time portrayed as “2023-10-24T19:04:31Z” for UTC times and “2023-10-24T12:04:31−07:00” if there is a time offset.

In addition to just parsing the date correctly depending on its format, one should also keep in mind that time zones might differ between different third parties. If data is retrieved from multiple different sources and comparisons are made between the different dates in these datasets, it can be beneficial to convert all of them to a standardised format in the system. This might be either local time zone, if the system only operates within one time zone, or UTC to be more robust and future proof.

3.2.1.4 Boolean

Boolean-like values can exist in various forms depending on the third-party source. The ones commonly encountered are explicit Boolean values, such as “true” and “false”, the numeral “1” and “0”, and language-specific variations on the explicit Boolean values.

3.2.1.5 Lists

While lists, or arrays, in themselves are very self-explanatory and require zero parsing, there are a few specific cases that should be considered when parsing data that is expected to come as a list. More specifically, they usually have two common edge cases that might require separate parsing logic; these are the empty value and a list with one value.

Especially when working with REST or SOAP APIs it is common to encounter different structures in the retrieved data depending on whether the list includes multiple values, one value, or no values. The parsing function needs to cover all these cases not to incorrectly return a parsing error. Furthermore, it should be decided on a case-by-case basis whether there should be a separation between the empty list and a “nullish” value.

3.2.1.6 Enumeration-like data

In most data structures, some kind of status or type fields exist that take on a value from a finite number of predefined values, which might or might not be known beforehand. Whether these should be considered enumerations in the system or not greatly depends on how these values are dealt with. If they are accessed frequently either in-code or by the end user, it might be beneficial to parse them as enumerations to obtain a more explicit and strict type that can be reused throughout the system.

3.2.1.7 “Nullish” data

Regardless of what type the data is, there are cases where some fields or properties of a data structure take on “non-existent” values, usually denoted as “nullish” or undefined values in the popular programming languages. How these values are portrayed in external systems can vary greatly depending on the underlying implementation. It is important to parse them correctly in one’s system to properly know what kind of data is being worked with and to be able to individually solve the nullish states on a functional level for each property. For example, some legacy API implementations and user interfaces might not have an explicit nullish state but, instead, it has to be understood from the context and parsed and transformed in the system not to have faulty true values in the system.

When working with strings the common nullish state is the empty string, or “”. Leaving such values in the codebase might lead to unforeseen errors due to the assumption that the value exists when, in fact, it might be preferred to have a different functionality in such cases.

For numerical values, a particularly difficult case to parse properly is the one where “0” has been used as a nullish value. In such cases, it is important to properly understand the property of the data and be certain that this is the case and that it is not just a record-like data type where the index 0 is referred to.

For Boolean values and for lists, it should be decided on a case-by-case basis whether the nullish value is of importance to the system. The distinction between a null value, an undefined value, and false might be relevant for the system in the case of Boolean values. For lists, cases might exist where the difference between a nullish value and an empty list might be relevant for the system; in such cases the differences should be conserved.

4 Schema validation libraries

A schema validation library is beneficial when working with third-party data, since it enhances data quality, security, and structure. It enables a smooth integration of external data sources while mitigating the challenges and risks commonly encountered in unpredictable data from third parties.

4.1 General characteristics

Schema validation libraries can validate any incoming data, checking for correct syntax, missing or malformed elements, and other anomalies. This assists in ensuring that the data is correct and reliable before being incorporated into one's system. This practice is crucial when dealing with systems that require precise data and communication with third parties, e.g., financial systems, personal information, and forms.

Especially when working with larger datasets or a larger system, it is common to run into efficiency and scalability issues at some stage during the development cycle. Schema validation libraries assist in streamlining this process from the data standpoint, minimising the risk of running into unforeseen issues when parsing large datasets, both from inefficient functions and the ability to reuse blocks of validation and parsing code to be used throughout the system. The possibility to manage and maintain reusable validation and parsing functions that can be used throughout the system both when dealing with data from multiple third parties and when having a large developer team will greatly assist in normalising the properties of the data structures that exist in the system.

There are three distinct types of functionalities that are expected from a schema validation library: data validation, data transformation, and data normalisation.

4.1.1 Data validation

4.1.1.1 Syntax checking and schema compliance

The schema validation library assists in providing easy-to-use functionality for the developer to define functions that validate the external data when it enters the system. The functions analyse the input data and ensure that it conforms to the expected structure. They can identify syntax errors, missing elements, or improper formatting, helping the developer to catch issues before they lead to runtime errors later in the development process. For example, when parsing JSON or XML data, the schema validation library can detect missing or mismatched brackets, tags, or quotation marks, ensuring that the data is well formed.

Schema compliance goes a step further than syntax checking by validating data against a user-defined schema or structure. A schema defines the rules, constraints, and data types that the data must adhere to, ensuring that it meets the specific business and technical requirements of the project. Schema validation libraries assist in schema validation by parsing the data into a structured representation, which can then be compared against the schema definition. In the context of JSON validation, for example, schemas created through the schema validation library can verify that the JSON data conforms to these specifications, preventing invalid or incompatible data from being processed. This provides full control to the developer of the type of the data once it has entered the system and been parsed by the schema, eliminating the risk of having unknown fields, unexpected field types, or missing elements existing in the system.

4.1.1.2 Error detection and handling

If the data contains any errors or unexpected formats, the schema validation library assists in providing robust error-handling mechanisms to make debugging efficient. In addition to simple error handling, they also provide functionality to skip problematic records in the data sets, or trigger alerts.

When the data fails to meet the syntactic or schematic requirements, these libraries can generate informative error messages, allowing developers to easily identify the issue and take the appropriate corrective action. This usually involves improving the schema to fit the requirements of the third party and transforming the data to fit the needs of one's own internal system. This kind of proactive error handling saves time and effort during development and troubleshooting phases and, additionally, allows the developer to have greater control of the error messages, which is especially beneficial when these error messages are displayed to the end user via the user interface.

Additionally, the problematic data can be sanitised and validated to protect against potential security vulnerabilities such as SQL injection attacks [10] or code execution exploits [11]. While many modern frameworks and libraries provide such protecting out-of-the-box it is always good to consider these threats when building a custom application that allows unstructured user input. This is especially important if the user input would communicate directly with the database of the system, or allows for defining custom functions that are run on the back-end of the application.

4.1.2 Data transformation

4.1.2.1 Data transformation and mapping

Schema validation libraries can take unstructured or semi-structured data and convert it into a structured representation that fits the needs of the language being used by the system. This structure is then easily accessible for manipulation and transformation, for example, converting XML data to JSON data to make it more developer friendly when dealing with the data in a language that natively supports structured JSON data, such as JavaScript or TypeScript.

When dealing with external data the developer often needs to extract specific elements from it. Schema validation libraries can simplify this process by providing simple-to-use methods to navigate through the external data structure. They also assist in mapping the data from one format to another, allowing developers to define how different elements should be transformed or mapped.

A regularly occurring practical example of this is dealing with SOAP APIs that return data in XML format, which needs to be converted to a structure that is easily managed in one's language of choice, be it a custom class or a JSON structure. The different data structures seldom have completely identical data types and naming conventions, naturally leading to greater benefits of such transformation functionality early in the development process, allowing the data that is being actively dealt with to be consistent with the rest of the data in the backend of the system under development.

4.1.2.2 Content validation and sanitization

When transforming the data, it is also important to validate and sanitise the content to ensure that it is easy to work with in the backend of the system and stays coherent with the rest of the data that is being dealt with. In addition to simply transforming and mapping the data, the schema validation libraries are also able to verify that the data adheres to the specific schema and its content rules. They allow developers to apply validation logic to the data on a field-by-field basis, ensuring that the data meets the requirements of each and every field to be able to be used in the system. They can also notify the developer as soon as possible in the runtime that the data is not as expected, disallowing such data to enter the system and giving a clear error message to developers, so these are able to fix the schema and improve the functionality around the errored fields, prior to the next deployment.

For example, when validating an email field from a dataset originating from an API request that will be used later in the process to actually send an email, it might be beneficial to actually validate that the string received conforms to a general email address, so that such a state never exists in the system that it repeatedly attempts to send an email to an incorrect email address. Errors like these can be extremely difficult to track down and might incorrectly give a sense that everything is working as expected.

4.1.3 Data normalisation

Another key aspect of schema validation libraries are robust and reusable data validation and transformation functions, which allow the developers to define reusable blocks of validation and transformation functions to normalise the process. This approach assists in aligning an external data set with the internal data structures and requirements of the data, focusing on dealing with discrepancies in the data at the earliest possible stage of the process from the system perspective. When working with multiple third parties this modularity also provides greater security throughout the system.

The reusable parsing and validation blocks also ensure consistency throughout the whole system, for example, by standardising different datasets from multiple third-party sources. In addition to solely standardising the format between datasets from multiple sources, they also help in aligning and standardising the format between the developers of the system. Many data fields such as dates, currencies, and structured strings, such as phone numbers or identification data, greatly benefit from having these predefined functions, normalising the data throughout the system.

4.2 Common schema validation libraries

There are multiple alternatives for choosing a schema validation library. In addition to depending on the programming language that the system is built on, the different aspects and characteristics should also be weighed according to their importance for the system that is being developed.

A short introduction to the most popular schema validation libraries for the three most popular programming languages will be given below. The programming languages have been determined based on the most recent yearly Stack Overflow developer survey [12]. The languages have been limited to only backend programming languages (eliminating SQL and HTML/CSS), and JavaScript and TypeScript have been combined to one language, since most packages for these languages can be used interchangeably, leaving us with JavaScript/TypeScript, Python, and Java as the languages for which we will explore the most popular schema validation libraries. The two most popular schema validation libraries for each of these languages have been chosen based on the number of stars on Github [13].

4.2.1 JavaScript/TypeScript

JavaScript and TypeScript are two of the most used programming languages [12] and will be the example language used in the case study of this thesis. They provide a lot of flexibility for the developer and due to having large communities also provide a wide variety of libraries and packages that can assist the development teams when managing non-standard use cases.

4.2.1.1 Zod

Zod [3] is a TypeScript first schema validation library, meaning that it has originally been developed for TypeScript and not been ported over from JavaScript to be compatible with TypeScript, leading to improved integration with the native type system of Typescript. The main goal of Zod is to eliminate duplicative type declarations. Zod allows the developer to declare a validation function once and it automatically infers the static TypeScript type from it.

Zod also supports a wide range of predefined validations such as date, IP, and email validation on strings, covering the vast majority of the commonly encountered specific cases that must be resolved when dealing with external data. Custom refinements are also supported for any edge cases that are not covered by the predefined set.

Data transformation is also a built-in part of Zod, reducing the need of separate transformation functions post-validation, which also significantly increases the validity and usability of the automatically inferred type that Zod provides.

4.2.1.2 Yup

Yup [14] is a schema builder for runtime parsing and validation originally built for JavaScript that also offers support for TypeScript.

Yup schemas allow the developer to create easy to read schemas for validation and value transformation of complex data structures.

4.2.2 Python

Python is a versatile, high-level programming language known for its readability and simplicity. Its syntax allows developers to express concepts in fewer lines of code than languages such as C++ or Java. Python's extensive libraries and frameworks support a wide range of applications, including web development, data analysis, artificial intelligence, and automation. Its popularity in these fields has made it a relevant language across various industries.

4.2.2.1 Pydantic

Pydantic [15] is a fast and extensible data validation library for Python. The schemas are defined by Python type hints, allowing for seamless integration with widely used static typing tools such as Mypy and Pyright, as well as commonly used IDEs such as Pycharm and Visual Studio Code.

Pydantic provides functionality to serialise the data into either native Python dictionaries or JSON. Together with the possibility to generate JSON schemas for any Pydantic defined schemas this allows for easy integration with a wide variety of other tools which support JSON schemas.

4.2.2.2 Cerberus

Cerberus [16] is a simple and lightweight data validation tool. Cerberus schemas are built on top of the vanilla Python types and much of the most common validation and normalisation rules are predefined as part of the package.

Error handling is highly customisable and by default it provides a clearly structured output that makes debugging of large, complex data sets with multiple errors easily manageable.

4.2.3 Java

Java is a robust, object-oriented programming language designed for portability and platform independence. Java is widely used for enterprise-level applications, mobile development (Android), and web development.

Its strong emphasis on reliability, security, and scalability has made it a cornerstone in the development of large-scale, mission-critical systems.

4.2.3.1 Jackson

Jackson [17] is a suite of data-processing tools for Java. Jackson consists of a vast number of individual modules for processing data in a wide range of different structures including JSON, XML, and YAML.

4.2.3.2 Gson

Gson [18] is a library for converting Java objects into their JSON representation and vice versa. Gson offers full support of Java Generics [19], allowing for support of more flexible data conversions. In addition to this Gson also does not require the source code of the Java objects, allowing for JSON conversion of Java objects from external applications.

5 Case study

This case study will focus on creating a robust implementation of a data pipe between a system and a third-party API where the main data that is being processed in the system is retrieved from.

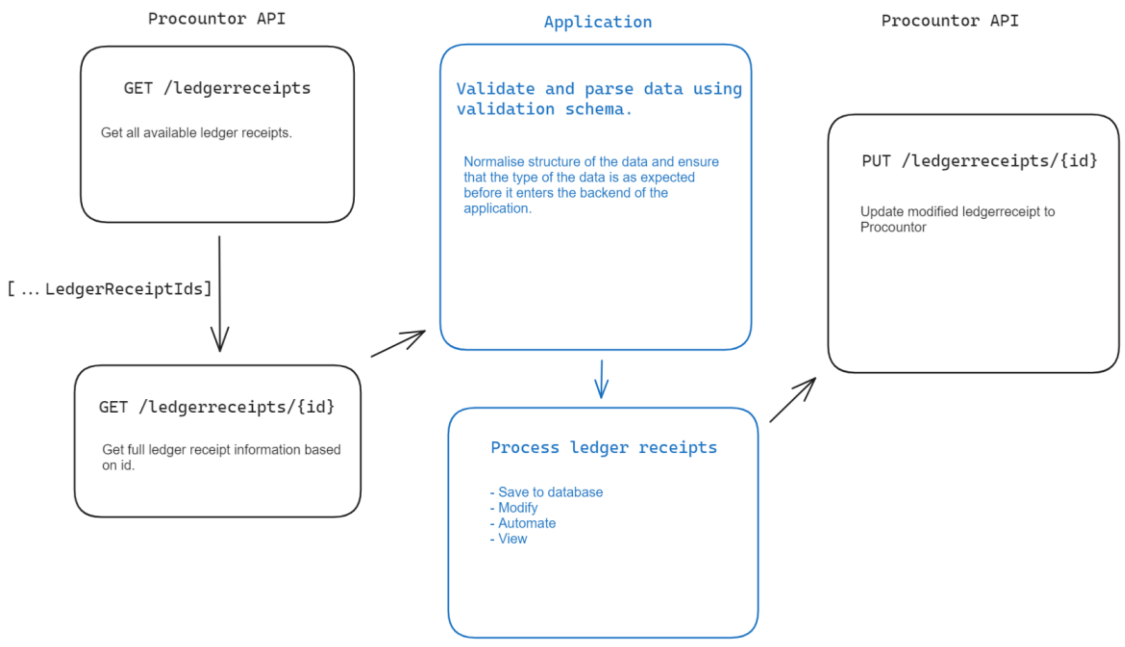
The implementation will include data validation, transformation, and normalisation, so that it will fit in a system regardless of its scale and the state of the third party where the data is retrieved from.

5.1 Scope of the case study

The focus of the study will lie completely on the actual processing of the data and how it is validated, transformed, and normalised to build a robust and error-free implementation that is easily scalable and maintainable regardless of the size of the system. A full implementation towards the external system including authentication and all relevant endpoints will not be explored, as it is outside of the scope of this case study.

How the data is dealt with after successfully fetching, validating, and normalising it from the external party is also beyond the scope of this study. In a real-world scenario the data would commonly be stored in a database, possibly through the assistance of an ORM [20] (An Object-relational mapping framework works as a conversion layer between a database and the objects within a programming language, assisting the developer in dealing with entries from the database from the backend of the system) , or simply instantly processed and sent back to the external party with the help of their PUT/POST endpoints.

To create a concrete example that is easily applicable to real-world scenarios this implementation will be based on a hypothetical system dealing with automating ledger receipts from an already existing third-party bookkeeping software, in this case Procountor [21]. The emphasis will be on the GET /ledgerreceipts/{receiptId} endpoint in the Procountor API [22], which returns the full ledger receipt data from Procountor.



Procountor API
GET /ledgerreceipts
Get all available ledger receipts.
[... LedgerReceiptIds]
GET /ledgerreceipts/{id}
Get full ledger receipt information based on id.
Application
Validate and parse data using validation schema.
Normalise structure of the data and ensure that the type of the data is as expected before it enters the backend of the application.
Process ledger receipts
- Save to database
- Modify
- Automate
- View
Procountor API
PUT /ledgerreceipts/{id}
Update modified ledgerreceipt to Procountor

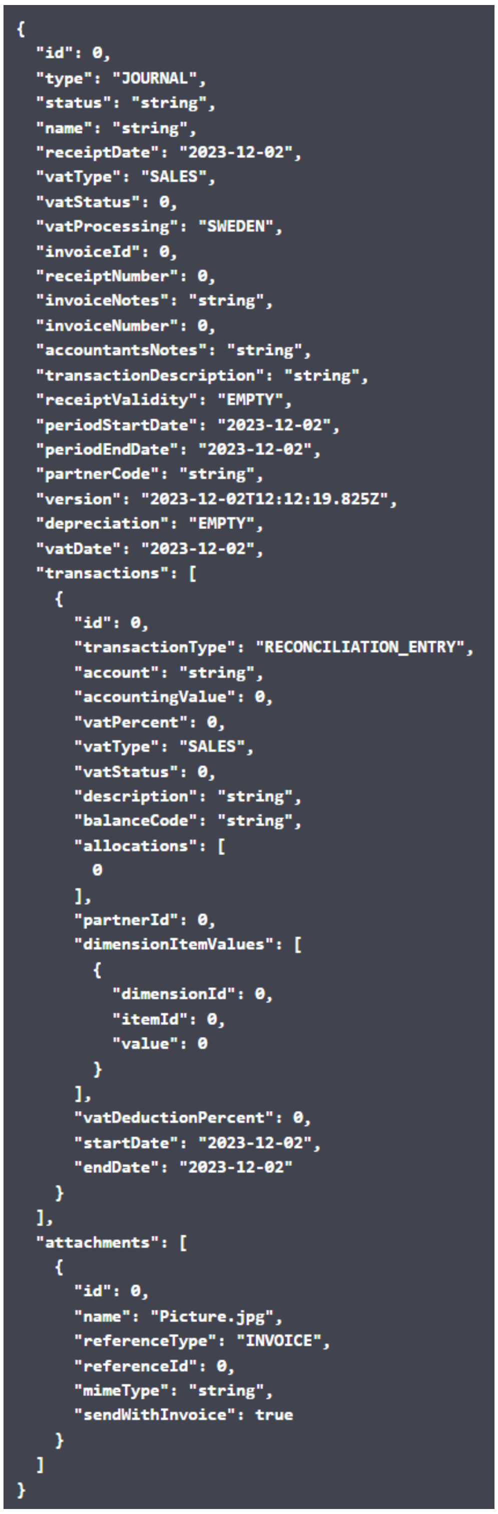
Figure 1: Diagram showcasing the Ledger receipt data pipe between Procountor and the case study application.

5.2 Tools and technologies

For the purposes of this case study TypeScript will be used. The language of choice is not relevant for such an implementation, and it should be decided on a case-by-case basis depending on the needs of the system and the previous experience of the development team. See chapter 3 above for a more in-depth explanation of what to look for when deciding on a programming language for a system.

Zod will be used as the schema validation library, since it allows for excellent customisation and flexibility. Zod includes several helper functions for validating and parsing the data. These offer considerable help when dealing with large and complex data structures from external APIs. Zod also natively supports data transformation, which will greatly improve the developer experience when reusing the automatically inferred static type provided by it throughout the system.

5.3 Relevance of the data in the system



```json
{
  "id": 0,
  "type": "JOURNAL",
  "status": "string",
  "name": "string",
  "receiptDate": "2023-12-02",
  "vatType": "SALES",
  "vatStatus": 0,
  "vatProcessing": "SWEDEN",
  "invoiceId": 0,
  "receiptNumber": 0,
  "invoiceNotes": "string",
  "invoiceNumber": 0,
  "accountantsNotes": "string",
  "transactionDescription": "string",
  "receiptValidity": "EMPTY",
  "periodStartDate": "2023-12-02",
  "periodEndDate": "2023-12-02",
  "partnerCode": "string",
  "version": "2023-12-02T12:12:19.825Z",
  "depreciation": "EMPTY",
  "vatDate": "2023-12-02",
  "transactions": [
    {
      "id": 0,
      "transactionType": "RECONCILIATION_ENTRY",
      "account": "string",
      "accountingValue": 0,
      "vatPercent": 0,
      "vatType": "SALES",
      "vatStatus": 0,
      "description": "string",
      "balanceCode": "string",
      "allocations": [
        0
      ],
      "partnerId": 0,
      "dimensionItemValues": [
        {
          "dimensionId": 0,
          "itemId": 0,
          "value": 0
        }
      ],
      "vatDeductionPercent": 0,
      "startDate": "2023-12-02",
      "endDate": "2023-12-02"
    }
  ],
  "attachments": [
    {
      "id": 0,
      "name": "Picture.jpg",
      "referenceType": "INVOICE",
      "referenceId": 0,
      "mimeType": "string",
      "sendWithInvoice": true
    }
  ]
}
```

Figure 2: Example ledger receipt taken from the Procountor API documentation. [22]

The data being retrieved are the ledger receipts. The structure of the ledger receipt can be examined in Figure 2. The functionality of the system completely relies on them and on the processing of them, requiring full structural knowledge of them as a data structure, and any future updates and changes to the API must be caught as early as possible in the system so that no data is unknowingly lost.

Depending on the functionality required by the system some of the fields might be of greater importance than others. Fields that will continuously be modified by the system as a part of the functionality provided by the system should be treated in such a way that they are easy to process both by the developers on the backend, as well as the end users through the user interface of the system if applicable. Sometimes this information is hard to know before hand, but some qualified guesses can be made early on to improve the usability down the line and limit the migration needs later in the development process. Two immediate characteristics that come to mind from looking at this example data are that dates are stored as strings (due to the nature of JSON) and that there are enumeration-like fields that should be treated as such instead of a generic string. The solution to both these observations will be described in section 5.4 below.

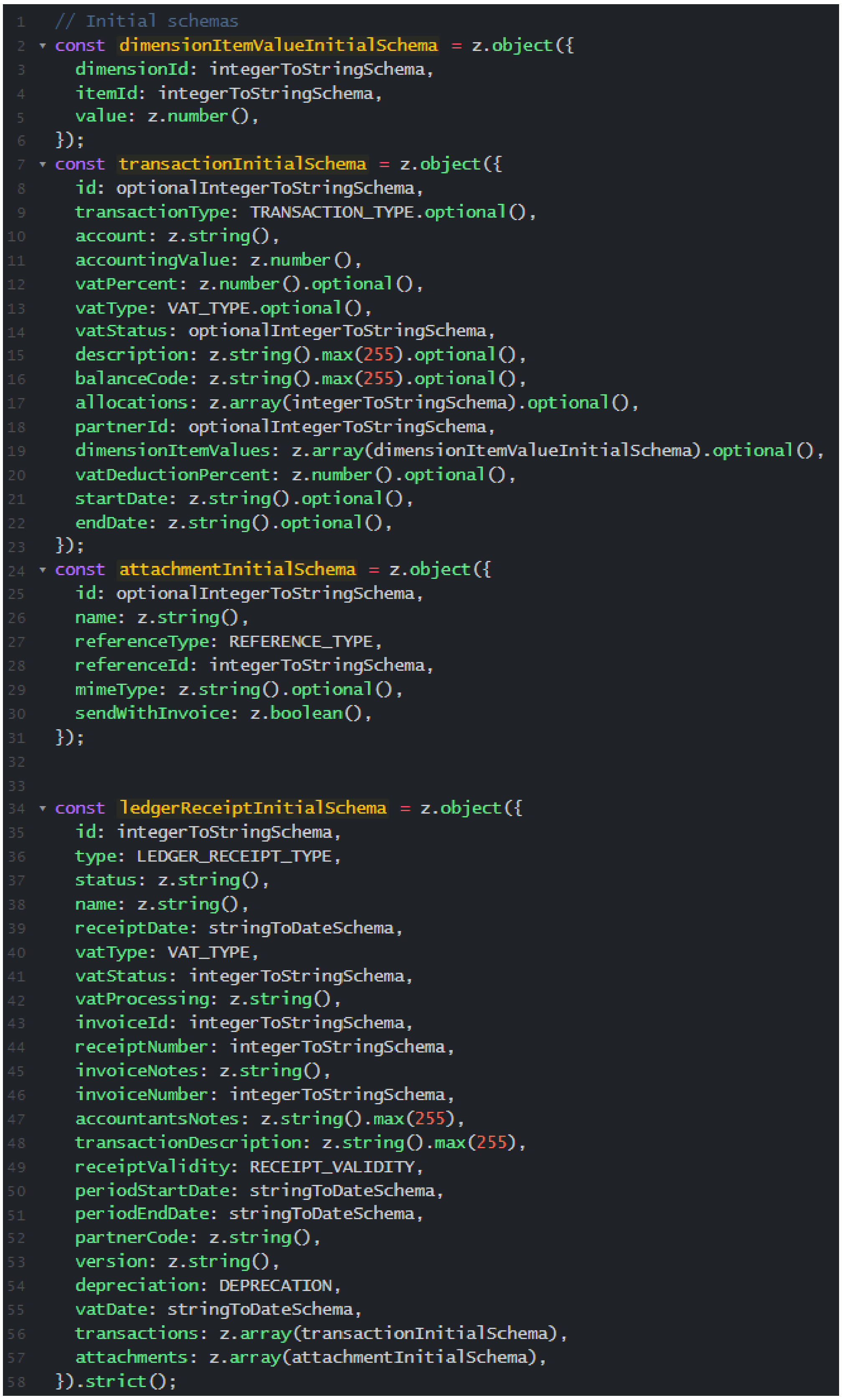
The schema validation should be built in such a way that nothing unknown can exist in the data from the external API, and the types of each field should be as narrow as possible so that the possible state of the data is as close to reality as possible. In addition to improving the robustness of the system, this also assists the developers by narrowing down the possibilities of the types of each field, reducing the need for unnecessary checks and steps in the functions in the system and giving a clearer overview of what can and cannot be modified on the data and which fields might be more relevant than others.

5.4 Validation schema

5.4.1 Initial implementation

The initial implementation is done based on the API documentation that is available from the third party, since this is the only information that is known to the developer. The documentation is seldom complete and usually either describes the structure too loosely, or is in some cases even completely lacking information about the existence of certain properties.

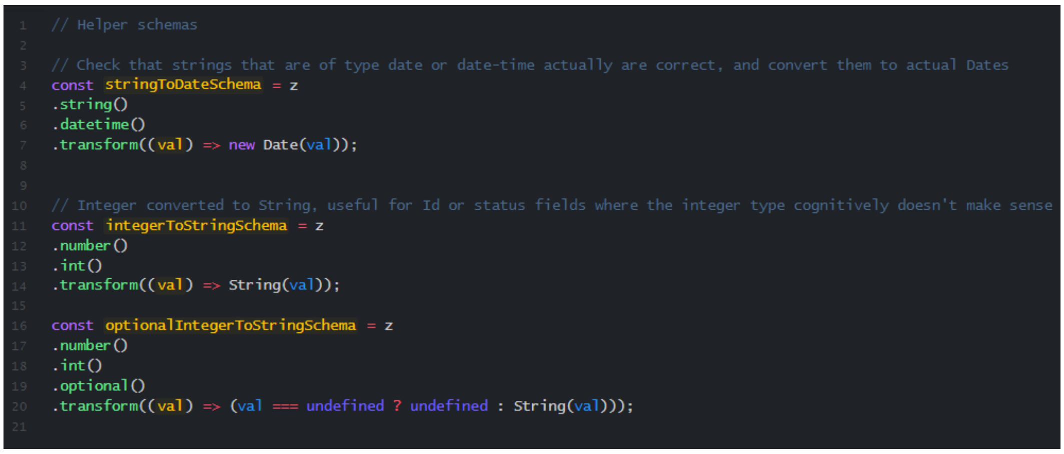
Since the full structure and all properties are of great importance to the system, it is essential that the validation function catches such discrepancies as early as possible. This is resolved by creating a schema validation function for the data that is as strict as possible, leading to it catching all unknowns as soon as they are encountered.



```
// Initial schemas
const dimensionItemValueInitialSchema = z.object({
  dimensionId: integerToStringSchema,
  itemId: integerToStringSchema,
  value: z.number(),
});
const transactionInitialSchema = z.object({
  id: optionalIntegerToStringSchema,
  transactionType: TRANSACTION_TYPE.optional(),
  account: z.string(),
  accountingValue: z.number(),
  vatPercent: z.number().optional(),
  vatType: VAT_TYPE.optional(),
  vatStatus: optionalIntegerToStringSchema,
  description: z.string().max(255).optional(),
  balanceCode: z.string().max(255).optional(),
  allocations: z.array(integerToStringSchema).optional(),
  partnerId: optionalIntegerToStringSchema,
  dimensionItemValues: z.array(dimensionItemValueInitialSchema).optional(),
  vatDeductionPercent: z.number().optional(),
  startDate: z.string().optional(),
  endDate: z.string().optional(),
});
const attachmentInitialSchema = z.object({
  id: optionalIntegerToStringSchema,
  name: z.string(),
  referenceType: REFERENCE_TYPE,
  referenceId: integerToStringSchema,
  mimeType: z.string().optional(),
  sendWithInvoice: z.boolean(),
});


const ledgerReceiptInitialSchema = z.object({
  id: integerToStringSchema,
  type: LEDGER_RECEIPT_TYPE,
  status: z.string(),
  name: z.string(),
  receiptDate: stringToDateSchema,
  vatType: VAT_TYPE,
  vatStatus: integerToStringSchema,
  vatProcessing: z.string(),
  invoiceId: integerToStringSchema,
  receiptNumber: integerToStringSchema,
  invoiceNotes: z.string(),
  invoiceNumber: integerToStringSchema,
  accountantsNotes: z.string().max(255),
  transactionDescription: z.string().max(255),
  receiptValidity: RECEIPT_VALIDITY,
  periodStartDate: stringToDateSchema,
  periodEndDate: stringToDateSchema,
  partnerCode: z.string(),
  version: z.string(),
  depreciation: DEPRECATION,
  vatDate: stringToDateSchema,
  transactions: z.array(transactionInitialSchema),
  attachments: z.array(attachmentInitialSchema),
}).strict();
```

Figure 3: The initial schema for the Procountor Ledger receipt and its children, based solely on the Procountor API documentation. [22]

The initial schema is presented in Figure 3. All the fields are defined exactly as described in the API documentation [22]. Some important decisions have been made to improve the robustness of the system from a developer standpoint.

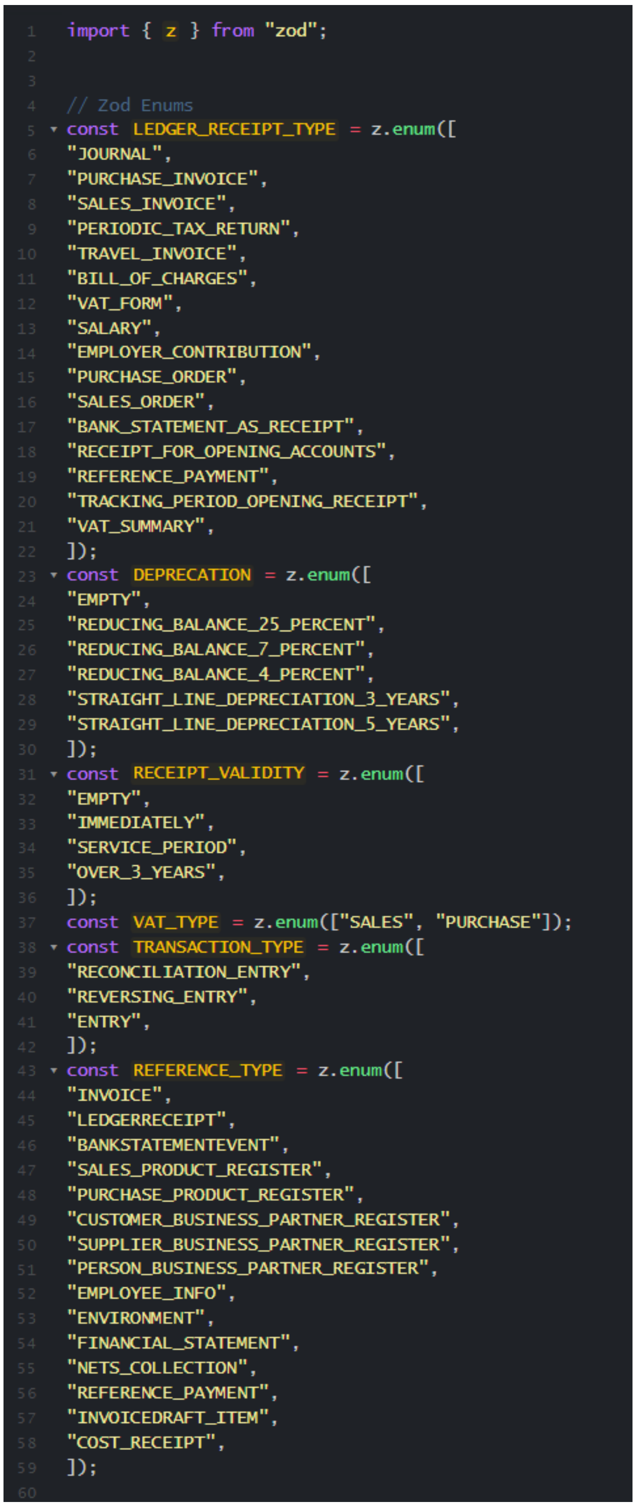


```
// Helper schemas

// Check that strings that are of type date or date-time actually are correct, and convert them to actual Dates
const stringToDateSchema = z
.string()
.datetime()
.transform((val) => new Date(val));


// Integer converted to String, useful for Id or status fields where the integer type cognitively doesn't make sense
const integerToStringSchema = z
.number()
.int()
.transform((val) => String(val));

const optionalIntegerToStringSchema = z
.number()
.int()
.optional()
.transform((val) => (val === undefined ? undefined : String(val)));
```

Figure 4: Helper functions that are reused throughout the schema validation functions.

Several of the status and identification fields in the API are returned as integers, and it has been determined that the numerical properties of such fields are not of importance, and they should be treated as strings in our system, to minimise the risk of running into unforeseen issues. A common example of such an issue is if the fields are optional and could take on the numerical value *0*. A true/false check on such a value would lead to unintended results. Helper functions for dealing with these fields have been created (Figure 4) and are reused through the schema validation functions to normalise the types. The helper function simply converts the integer to a string, and also takes into account the special case of the value of the field being *undefined*, explicitly checking against the undefined value so that the value *0* is also treated correctly.

The date and date time fields are returned from the API as strings. A helper function (Figure 4) has also been created for these fields that validates that the string is in fact a date, and then parses the field to an actual Date object.



```
import { z } from "zod";


// Zod Enums
const LEDGER_RECEIPT_TYPE = z.enum([
"JOURNAL",
"PURCHASE_INVOICE",
"SALES_INVOICE",
"PERIODIC_TAX_RETURN",
"TRAVEL_INVOICE",
"BILL_OF_CHARGES",
"VAT_FORM",
"SALARY",
"EMPLOYER_CONTRIBUTION",
"PURCHASE_ORDER",
"SALES_ORDER",
"BANK_STATEMENT_AS_RECEIPT",
"RECEIPT_FOR_OPENING_ACCOUNTS",
"REFERENCE_PAYMENT",
"TRACKING_PERIOD_OPENING_RECEIPT",
"VAT_SUMMARY",
]);
const DEPRECATION = z.enum([
"EMPTY",
"REDUCING_BALANCE_25_PERCENT",
"REDUCING_BALANCE_7_PERCENT",
"REDUCING_BALANCE_4_PERCENT",
"STRAIGHT_LINE_DEPRECIATION_3_YEARS",
"STRAIGHT_LINE_DEPRECIATION_5_YEARS",
]);
const RECEIPT_VALIDITY = z.enum([
"EMPTY",
"IMMEDIATELY",
"SERVICE_PERIOD",
"OVER_3_YEARS",
]);
const VAT_TYPE = z.enum(["SALES", "PURCHASE"]);
const TRANSACTION_TYPE = z.enum([
"RECONCILIATION_ENTRY",
"REVERSING_ENTRY",
"ENTRY",
]);
const REFERENCE_TYPE = z.enum([
"INVOICE",
"LEDGERRECEIPT",
"BANKSTATEMENTEVENT",
"SALES_PRODUCT_REGISTER",
"PURCHASE_PRODUCT_REGISTER",
"CUSTOMER_BUSINESS_PARTNER_REGISTER",
"SUPPLIER_BUSINESS_PARTNER_REGISTER",
"PERSON_BUSINESS_PARTNER_REGISTER",
"EMPLOYEE_INFO",
"ENVIRONMENT",
"FINANCIAL_STATEMENT",
"NETS_COLLECTION",
"REFERENCE_PAYMENT",
"INVOICEDRAFT_ITEM",
"COST_RECEIPT",
]);

```

Figure 5: Enumerations for the Procountor Ledger receipt

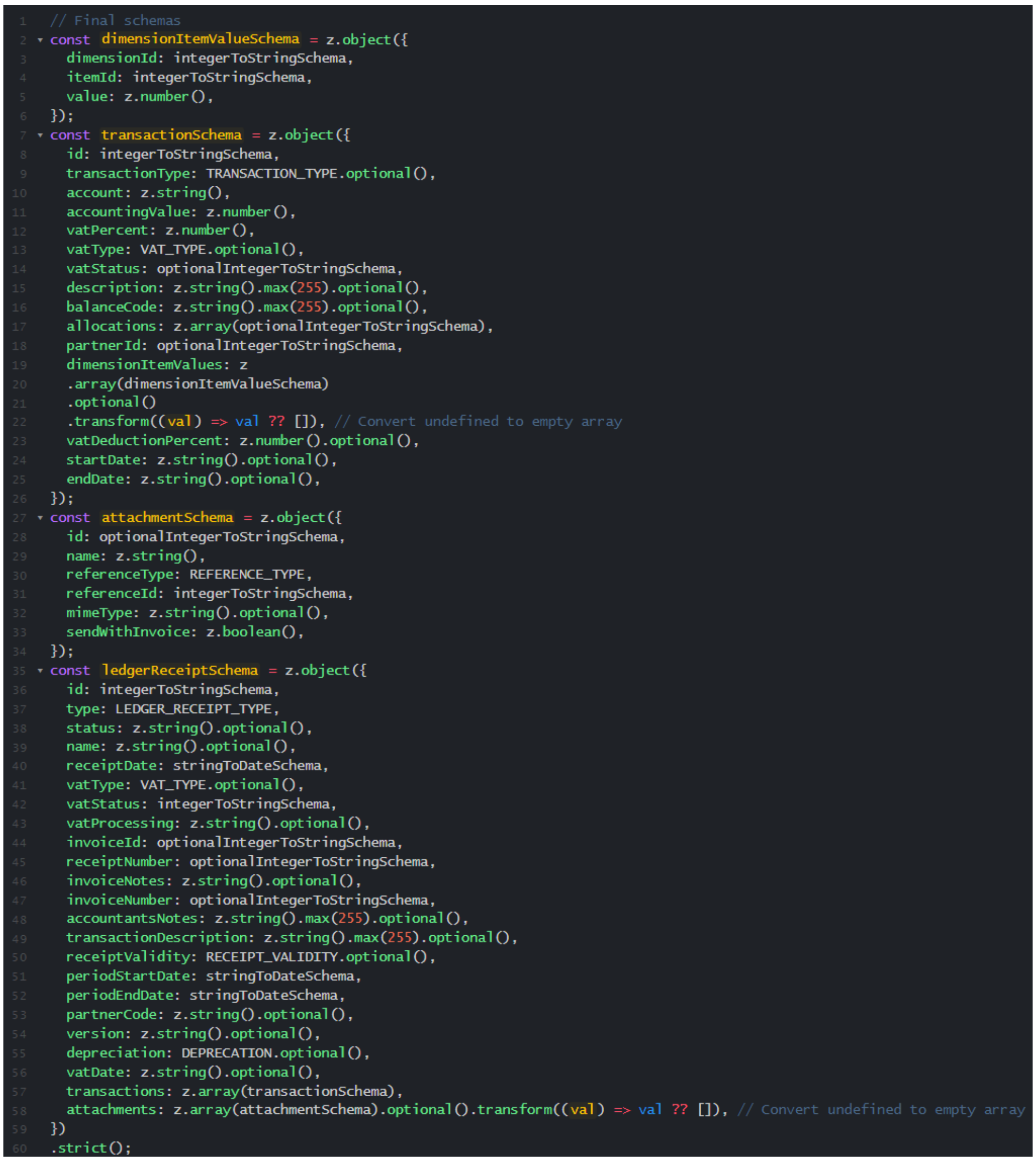
Several of the fields have a specific set of values they can take and are defined in the API documentation as such. These enumerations have been defined separately (Figure 5) and are reused throughout the schemas. When expanding on the system and using more API resources from this specific source this will offer great reusability.

An explicit ".strict()" check has also been added to the whole schema. This will throw an error as soon as an unknown property exists on the data structure. This allows the developer to catch this new information as soon as possible, eliminating the possibility of overlooked updates to the external API that could be of importance to the system.

5.4.2 Improving implementation with real data

Once the initial implementation of the validation function is developed, it can be tested by using it to validate data fetched from the external source. All inconsistencies between the API documentation and the actual, real-world data will be caught by the schema validation function, alongside a descriptive error message, clearly guiding the developer as to what the faulty data looked like and what the expected state was. This allows for rapid iteration of the data validation function, ensuring that the structure of the data is known as precisely as possible without including any guesswork from the developer.

After running the initial validation schema against tens of thousands of real-life ledger receipts in Procountor, several errors were thrown. What this in practice means is that the API documentation was not completely transparent regarding the actual functionality of the API and how the data is returned.



```
// Final schemas
const dimensionItemValueSchema = z.object({
  dimensionId: integerToStringSchema,
  itemId: integerToStringSchema,
  value: z.number(),
});
const transactionSchema = z.object({
  id: integerToStringSchema,
  transactionType: TRANSACTION_TYPE.optional(),
  account: z.string(),
  accountingValue: z.number(),
  vatPercent: z.number(),
  vatType: VAT_TYPE.optional(),
  vatStatus: optionalIntegerToStringSchema,
  description: z.string().max(255).optional(),
  balanceCode: z.string().max(255).optional(),
  allocations: z.array(optionalIntegerToStringSchema),
  partnerId: optionalIntegerToStringSchema,
  dimensionItemValues: z
  .array(dimensionItemValueSchema)
  .optional()
  .transform((val) => val ?? []), // Convert undefined to empty array
  vatDeductionPercent: z.number().optional(),
  startDate: z.string().optional(),
  endDate: z.string().optional(),
});
const attachmentSchema = z.object({
  id: optionalIntegerToStringSchema,
  name: z.string(),
  referenceType: REFERENCE_TYPE,
  referenceId: integerToStringSchema,
  mimeType: z.string().optional(),
  sendWithInvoice: z.boolean(),
});
const ledgerReceiptSchema = z.object({
  id: integerToStringSchema,
  type: LEDGER_RECEIPT_TYPE,
  status: z.string().optional(),
  name: z.string().optional(),
  receiptDate: stringToDateSchema,
  vatType: VAT_TYPE.optional(),
  vatStatus: integerToStringSchema,
  vatProcessing: z.string().optional(),
  invoiceId: optionalIntegerToStringSchema,
  receiptNumber: optionalIntegerToStringSchema,
  invoiceNotes: z.string().optional(),
  invoiceNumber: optionalIntegerToStringSchema,
  accountantsNotes: z.string().max(255).optional(),
  transactionDescription: z.string().max(255).optional(),
  receiptValidity: RECEIPT_VALIDITY.optional(),
  periodStartDate: stringToDateSchema,
  periodEndDate: stringToDateSchema,
  partnerCode: z.string().optional(),
  version: z.string().optional(),
  depreciation: DEPRECATION.optional(),
  vatDate: z.string().optional(),
  transactions: z.array(transactionSchema),
  attachments: z.array(attachmentSchema).optional().transform((val) => val ?? []), // Convert undefined to empty array
})
.strict();
```

Figure 6: Finalised schema based on parsing thousands of ledger receipts.

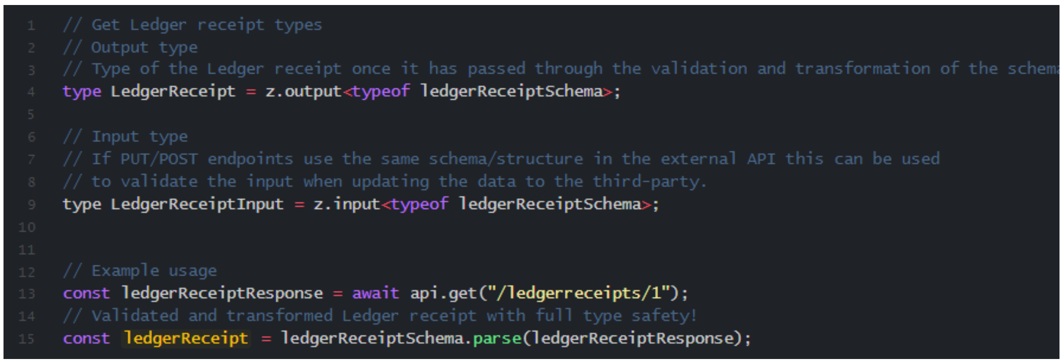
The schema was iteratively improved once each error was encountered, and the final version is presented in Figure 6. The most notable difference between the initial schema and the finalised version is the possibility for many of the properties of the data structure to be *undefined*. This has been handled in the validation schema by giving these fields the ".optional()" property.

Note specifically the optional arrays that are returned from the API. These are already in the schema transformed from *undefined* to empty arrays.

Functionality-wise, *undefined* and an empty array mean the same in this case, but this pre-emptive conversion greatly improves the developer experience when working with such data, since no explicit checks on *undefined* values will have to be performed when working with this dataset in the system.

5.5 Using the validation schemas

The validation schemas are now completed and can be used to validate and parse the data that arrives to the system from the external API.



```
// Get Ledger receipt types
// Output type
// Type of the Ledger receipt once it has passed through the validation and transformation of the schem
type LedgerReceipt = z.output<typeof ledgerReceiptSchema>;

// Input type
// If PUT/POST endpoints use the same schema/structure in the external API this can be used
// to validate the input when updating the data to the third-party.
type LedgerReceiptInput = z.input<typeof ledgerReceiptSchema>;


// Example usage
const ledgerReceiptResponse = await api.get("/ledgerreceipts/1");
// Validated and transformed Ledger receipt with full type safety!
const ledgerReceipt = ledgerReceiptSchema.parse(ledgerReceiptResponse);
```

Figure 7: Example usage of the validation schemas and type extraction from them.

Input and output types are easily extracted from the validation schema (Figure 7), providing the developer with full type safety and a reusable type that can easily be extended or iterated on by improving and modifying the validation schemas when changes occur in the third-party system.

6 Conclusion

The first step to building a robust and uniform system is ensuring that the data that moves through the system is known. When working with third party applications, through APIs, data scraping, RPA (Robot process automation), or even just user input within the application under development, the importance of proper parsing or validation quickly becomes apparent.

While some of these issues can be partly mitigated by using strongly typed programming languages and having good knowledge of the third-party applications there will always exist cases when it is impossible to fully control the data that enters the system. This can happen for various reasons but most commonly it is impossible to fully be aware of the quirks and limitations of third-party applications. While there might exist relatively good documentation for APIs the information is seldom complete and will include discrepancies when it comes to data format or nullability, or even lack of information about certain fields in the data that is part of the request or response that the API provides. For user input, RPA solutions or data scraping the issues are much more prevalent since the assumptions are made completely by the developer since there often are no communication channels with the third party in these kinds of data integration solutions. In these cases, the use of a schema validation library becomes crucial for maintaining a robust and error-free backend and business logic in the system.

The use of a schema validation library for parsing, validating, and transforming the external data ensures that the data is fully typed and includes no unknowns when entering the backend of the application under development. This approach brings numerous improvements to the system that allow for a more robust, easily maintained, well-documented, and unform system. Some of the concrete improvements that this brings are the following:

- Explicitly defined schemas for third-party data pipes that act as documentation and single point of entry to the backend under development.

- Up-to-date documentation about the structure of the data that enters the system.
- Unified approach to how to define and document the structure of external data.
- Single point in the application where parsing and validation of external data is done, which ensures that the backend code of the system remains clean and can strictly focus on relevant business logic.
- Transformation of external data can be managed as a part of the parsing and validation pipeline, providing reusable types and functions to unify data from multiple different third parties.

The benefits of parsing or validation libraries quickly become apparent when dealing with large and complex data structures, and the benefits mentioned above are undeniable and instantly visible when implementing this approach in a system under development.

7 Sammanfattning

I modern programmering är det vanligt att förr eller senare vara tvungen att hantera data från externa system. Oberoende om dessa data härstammar från filer, externa API:s eller vagt integrerade gamla system, kommer man vid något skede att behöva ta itu med problemet att försöka komma underfund med hur strukturen av dessa data ser ut.

Det naiva sättet att ta itu med detta problem är att försöka gissa sig fram till hur ens data kommer att se ut, antingen på basis av tidigare erfarenhet eller någon existerande dokumentation som finns tillgänglig, och sedan validera dessa data vid olika tillfällen i systemets kod när man stöter på problem eller osäkerheter. Detta leder snabbt till mycket extra komplexitet och ostrukturerad kodbas som är svår att upprätthålla.

Man kan kringgå dessa problem genom att tolka alla externa data som kommer till systemet, så att datans struktur alltid är känd. Genom att tillämpa tolkningsfunktioner när extern information kommer in i systemet kan man strukturera kodbasen bättre och fokusera all logik kring sådana data på ett och samma ställe för att förbättra dokumentation och underhållbarhet, samt hålla resten av ens kod och logik simpel och fri från annan logik.

Målet med denna avhandling är att undersöka de problem som ofta uppstår när man arbetar med komplexa externa data, samt hur tolkningsfunktioner kan tillämpas för att förbättra hanteringen av dessa data. Individer eller mjukvaruproduktionsteam som måste hantera externa data med komplexa datastrukturer kommer förhoppningsvis att finna något nytt och lärorikt i denna avhandling som kan förbättra deras produktivitet, samt robustheten av system under utveckling.

Fokuset i avhandlingen kommer att vara på hur en implementation av ett tolkningsbibliotek kan se ut och vad dess tillämpning konkret kan ge för nytta. Det är möjligt att göra en implementation av detta i vilket modernt programmeringsspråk som helst, men fokuset här kommer att ligga på TypeScript och tolkningsbiblioteket Zod. Anledningen till dessa val är att

TypeScript, som ett statiskt typat programmeringsspråk, gagnar av användning av ett tolkningsbibliotek då typer av dess tolkningsfunktioner kan återanvändas genom hela systemets kodbas efter att tolkningsscheman har definierats. Zod erbjuder utvecklaren enkel syntax att definiera valideringsfunktioner med, och nästan oändliga möjligheter för raffinemang och transformationer av data, som möjliggör behandling av vilken som helst typ av data. Externa data från API:s kommer att vara främst i fokus, men de lärdomar och praxis som förklaras här kan förstås också tillämpas på data från andra källor.

System är ofta beroende av flera diverse typer av data som har olika krav på integriteten av deras struktur. Det finns data som är kritiska för den övergripande funktionaliteten av systemet, som kräver mycket precis definition, samt mindre kritiska datakomponenter som kanske bara flödar genom systemet utan mycket behandling. För att optimera robustheten medan man håller utvecklingstiden och komplexiteten låg är det viktigt att skilja på dessa olika datatyper som finns i ens system.

Då man arbetar med data från externa system är strukturen sällan fullt känd, och dokumentation är oftast bristfällig eller helt och hållet obefintlig. Detta leder till att det inte är möjligt att veta hur data kommer att se ut då de kommer in i systemet, vilket kan leda till problem senare i utvecklingen eller då systemet redan har varit i användning i en längre tid, då något okänt format plötsligt kommer in i systemets back-end kod och businesslogik. Dylika problem, som är svåra att förutspå och som kan uppstå sent i utvecklingen eller då systemet redan är i full användning, är dyra och tidskrävande att reda ut och åtgärda. Ju tidigare problemen hittas, desto bättre och robustare system har man åstadkommit att utveckla.

Förutom otillräcklig kunskap om externa datas struktur innehåller data ofta även skillnader i hur information visas eller dess format, vilket kan bero på implementationen i fråga, som kan skilja sig från hur data ser ut i programmeringsspråket i det egna systemet. Ofta förekommande exempel på sådana problem är formatet på datum eller tidfält, hur tomma fält, såsom “null” eller “odefinierat” hanteras, samt olika format på numeriska fält som måste hanteras innan de kan hanteras korrekt i det aktuella systemet.

Dessa problem kan lösas med hjälp av tolkningsbibliotek, genom att definiera tolkningsscheman för dessa okända externa datastukturer, som ser till att strukturen på de data som kommer in i ens system är entydig och klart definierad. Detta tillvägagångssätt ger flera konkreta förbättringar som direkt syns i utvecklingen av systemet, exempelvis:

- Automatisk dokumentation på externa data och deras struktur
- Tolkning, validering och transformering av externa data sker på ett och samma ställe, direkt då data kommer in i systemet.
- Resten av kodbasen hålls ren, dataspecifik logik hålls skild från businesslogik.
- Enhetligt tillvägagångssätt på hur man ska definiera och behandla strukturen av externa data.
- Transformering av externa data kan effektiviseras och dess funktionalitet återanvändas för att förena data från flera olika externa system.

Fördelarna av användning av tolknings- och valideringsbibliotek blir snabbt tydliga då man måste behandla stora, komplexa datastrukturer som härstammar från externa system. Implementation av detta tillvägagångssätt förbättrar inte bara robustheten av systemet, utan fungerar också som automatisk, aktuell dokumentation som förbättrar förståelsen och kommunikationen mellan utvecklare i större team. Förutom dessa fördelar hjälper den också utvecklarna att arbeta enligt samma mönster och bästa praxis för att hålla strukturen bekant och förståelig för alla inblandade parter.

8 Bibliography

[1] J. Rinta-Filppula, IS STATIC TYPE CHECKING WORTH IT?, April 2021. https://trepo.tuni.fi/bitstream/handle/10024/131013/Rinta-FilppulaJaakko.pdf;jsessionid=A3C4F83CD89073600F8EC96B8522E1E1?sequence=2 (Last read: 23.10.2023)

[2] V. Nurminen, Unification of form validation implementations in web clients and servers, December 2022. https://aaltodoc.aalto.fi/bitstream/handle/123456789/119373/master_Nurminen_Valtteri_2023.pdf?sequence=1&isAllowed=y (Last read: 23.10.2023)

[3] Zod, https://zod.dev/ (Last read: 21.10.2023)

[4] J. C. Westland, The cost of errors in software development: evidence from industry, February 2000. https://pdf.sciencedirectassets.com/271629/1-s2.0-S0164121200X00970/1-s2.0-S0164121201001303/main.pdf?X-Amz-Security-Token=IQoJb3JpZ2luX2VjEAUaCXVzLWVhc3QtMSJGMEQCIAtwkNgG9YyOuax9xLz1lSwlT%2BEcyzN5SO5apwbsSGBKAiBjB5dIVbffaGyES7WBsG5DVaRO4EnUGIQfUok2BCb2xi (Last read: 23.10.2023)

[5] E. Engheim, Medium, January 2020. https://erik-engheim.medium.com/the-many-advantages-of-dynamic-languages-267d08f4c7 (Last read: 26.02.2024)

[6] Nullish value, https://developer.mozilla.org/en-US/docs/Glossary/Nullish (Last read: 23.10.2023)

[7] Oracle documentation, decimal and thousands separators, https://docs.oracle.com/cd/E19455-01/806-0169/overview-9/index.html (Last read: 24.10.2023)

[8] IEEE 754, https://en.wikipedia.org/wiki/IEEE_754 (Last read: 21.10.2023)

[9] ISO 8601 — Date and time format, February 2017. https://www.iso.org/iso-8601-date-and-time-format.html (Last read: 21.10.2023

[10] kingtorin, OWASP: SQL Injection, https://owasp.org/www-community/attacks/SQL_Injection (Last read: 02.12.2023)

[11] Y. Montoya, Vaadata, October 2023. https://www.vaadata.com/blog/rce-remote-code-execution-exploitations-and-security-tips/#:~:text=Remote%20Code%20Execution%20(RCE)%20is,remotely%20with%20no%20physical%20access. (Last read: 02.12.2023)

[12] Stack Overflow Developer Survey 2023, June 2023. https://survey.stackoverflow.co/2023/#technology-most-popular-technologies (Last read: 21.10.2023)

[13] Github, https://github.com/ (Last read: 21.10.2023)

[14] Yup Github, September 2014. https://github.com/jquense/yup (Last read: 21.10.2023)

[15] Pydantic, https://docs.pydantic.dev/latest/ (Last read: 02.12.2023)

[16] Cerberus, https://docs.python-cerberus.org/ (Last read: 02.12.2023)

[17] FasterXML/jackson: Main Portal page for the Jackson project, https://github.com/FasterXML/jackson (Last read: 21.10.2023)

[18] google/gson: A Java serialization/deserialization library to convert Java Objects into JSON and back, https://github.com/google/gson (Last read: 21.10.2023)

[19] Oracle Java tutorials: Generic types, https://docs.oracle.com/javase/tutorial/java/generics/types.html (Last read: 02.12.2023)

[20] Wikipedia: Object-relational mapping, https://en.wikipedia.org/wiki/Object%E2%80%93relational_mapping (Last read: 02.12.2023)

[21] Procountor, https://procountor.fi/ (Last read: 22.10.2023)

[22] Procountor API, https://dev.procountor.com/api-reference/ (Last read: 22.10.2023)