GENERATING TYPESCRIPT TYPES FROM OPENAPI SPECIFICATION FOR IMPROVED DEVELOPER EXPERIENCE

Master's thesis

Lappeenranta–Lahti University of Technology LUT

Master's Programme in Software Engineering and Digital Transformation

2025

Topi Ilmoni

Examiners: Antti Knutas, Associate Professor

Maija Heiskanen, M.Sc (Tech)

ABSTRACT

Lappeenranta–Lahti University of Technology LUT

LUT School of Engineering Science

Software Engineering

Topi Ilmoni

Generating Typescript Types from OpenAPI Specification for Improved Developer Experience

Master's thesis

2025

55 pages, 8 figures, 2 tables and 1 appendix

Examiners: Associate Professor Antti Knutas and Maija Heiskanen, M.Sc (Tech)

Keywords: OpenAPI, TypeScript, Developer Experience, code generation

The OpenAPI specification has become a popular tool as it can be used to generate documentation, tests and SDKs. The problem for the case company was the manual process of writing TypeScript types from OpenAPI specification which they wanted to automate for improved Developer Experience. According to previous research, Developer Experience defines how developers feel and perceive their job, and it directly influences productivity and project success. In this thesis, Design Science Research Method was used to produce a solution. Alternative approaches to producing the artifact were explored, ultimately leading to the decision to extend an open-source library. The library required various modifications to meet case application's requirements. An explorative survey was used to evaluate the artifact's effectiveness in improving Developer Experience. The artifact uncovered issues originating from the OpenAPI specification which were noted for future improvement. The survey indicated that Developer Experience was improved by faster feedback loops and reduction in the cognitive load. However, the survey had a small sample size and limited data collection timeframe.

TIIVISTELMÄ

Lappeenrannan–Lahden teknillinen yliopisto LUT

LUT Teknis-luonnontieteellinen

Tietotekniikka

Topi Ilmoni

TypeScript-tyyppien generointi OpenAPI-spesifikaatiosta kehittäjäkokemuksen parantamiseksi

Tietotekniikan diplomityö

2025

55 sivua, 8 kuvaa, 2 taulukkoa ja 1 liite

Tarkastajat: Apulaisprofessori Antti Knutas ja Diplomi-insinööri Maija Heiskanen

Avainsanat: OpenAPI, TypeScript, kehittäjäkokemus, lähdekoodin generointi

OpenAPI-spesifikaatiosta on tullut suosittu työkalu, koska sitä voidaan käyttää dokumentaation, testien ja SDK:iden luomiseen. Case-yrityksen ongelmana oli TypeScript-tyyppien kirjoittaminen OpenAPI-spesifikaatiosta manuaalisesti, ja he halusivat automatisoida tämän prosessin kehittäjäkokemuksen parantamiseksi. Aiemman tutkimuksen mukaan kehittäjäkokemus määrittelee, miten kehittäjät kokevat ja havaitsevat työnsä, ja se vaikuttaa suoraan tuottavuuteen ja projektien onnistumiseen. Tässä tutkielmassa käytettiin Design Science Research Method -menetelmää ratkaisun tuottamiseksi. Vaihtoehtoisia lähestymistapoja artefaktin tuottamiseen tutkittiin, mikä lopulta johti päätökseen laajentaa avoimen lähdekoodin kirjastoa. Kirjastoon tehtiin useita muutoksia, jotta se täyttäisi case-sovelluksen vaatimukset. Artefaktin tehokkuutta kehittäjäkokemuksen parantamisessa arvioitiin eksploratiivisella kyselytutkimuksella. Artefakti paljasti OpenAPI-spesifikaatiosta johtuvia ongelmia, jotka merkittiin ylös tulevaisuuden parannusta varten. Kysely paljasti, että kehittäjäkokemus parani nopeamman palautteen ja kognitiivisen kuormituksen vähenemisen ansiosta. Kyselyllä oli kuitenkin pieni otoskoko ja rajallinen tiedonkeruuaika.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Antti Knutas for the input regarding how to utilize the Design Science Research Method in the thesis. I would also like to thank my supervisor Maija Heiskanen for providing valuable feedback what approaches could be taken for the artifact.

ABBREVIATIONS

API	Application Programming Interface
DSRM	Design Science Research Method
DX	Developer Experience
GUID	Globally Unique Identifier
HTTP	Hypertext Transfer Protocol
ID	Identifier
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
REST	Representational State Transfer
ROI	Return on Investment
SDK	Software Development Kit
UI	User Interface
URL	Uniform Resource Locator
UX	User Experience
YAML	Yet Another Markup Language
npm	Node Package Manage

Table of contents

DECLARATIONS

AI usage

The author of the thesis, Topi Ilmoni, used the following AI-tools during the preparation of the thesis:

GitHub Copilot

a. Purpose of use: Code suggestions
b. Explanation of the use of the tool: During the development of the artifact, code suggestions were provided by Copilot. All suggestions were verified by the author.

Google Gemini

c. Purpose of use: Improving grammar
d. Explanation of the use of the tool: Gemini was used while reviewing the thesis to improve grammar. All suggestions were verified by the author.

The author, Topi Ilmoni, takes full responsibility for the content of this thesis and has reviewed and edited the content generated by the possible use of AI tools.

1 Introduction

A modern web application relies on a well-defined API to enable communication between frontend and backend systems. OpenAPI specification has become a widely used tool for describing RESTful APIs, which can be seen by its large presence in GitHub repositories [1]. However, for developers to unlock its full potential, it must be translated into frontend code. TypeScript has gained widespread adoption for building scalable and maintainable frontend applications, primarily due to its static typing and rich developer tooling [2]. The annual Stack Overflow developer survey shows that its popularity among professionals has increased significantly from 28.3% in 2020 to 48.8% in 2025 [3], [4]. The benefits of TypeScript can be fully realized when developers have accurate type definitions for the APIs they are consuming.

The case application's frontend contains approximately 950 manually typed API models. Frontend developers are responsible for ensuring that the TypeScript types they write exactly match the OpenAPI specification. An approach proposed in a previous study is a tool that automatically generates TypeScript types from the OpenAPI specification [5]. Such an automated tool would ensure that TypeScript types remain consistent with the OpenAPI specification. A study by Koren and Klamma noted that the motivating factor for implementing this kind of automation is the reduction of complexity and time required [6]. Another study by Donato and Qin shared the same consensus on the manual process being time-consuming and prone to human-error. Although they emphasized that in the case where automation of the process is achieved, the discrepancies in the OpenAPI specification can cause new problems [7].

1.1 Objectives and limitations

The goal of this thesis is to develop a tool that automatically generates TypeScript types from an OpenAPI specification. This tool is designed to improve the Developer Experience by automating the tedious task of writing type files manually. A key question is what to consider when choosing such a tool. The tool will be used by professionals in their daily work and then its effect on the Developer Experience will be evaluated.

The research questions of this thesis are:

Q1: What options are available for generating TypeScript types from an OpenAPI 3 specification?

Q2: What kind of factors need to be considered when choosing the tool?

Q3: Does the tool improve the Developer Experience?

2 OpenAPI specification

The OpenAPI Specification is a language-independent standard used for describing RESTful web services [8]. It is considered as a de facto standard [9]. It allows humans and machines to understand an API's capabilities without accessing its source code. An OpenAPI document is typically a YAML or JSON file and contains information such as:

- info: Metadata containing the title, version, and license.
- servers: URLs for different environments (internal, production, etc).
- paths: Endpoints and their supported HTTP operations (GET, POST, etc.).
- components: Reusable objects that are mainly model schemas.
- security: Authorization methods required to access the API.

The OpenAPI specification can be parsed to automatically generate code in any programming language. The most common use cases for this are:

- API documentation: Allows users to view documentation and test API calls directly.
- SDKs: Removes the need to manually write API communication methods.
- Automated tests: Verifies that the implementation matches the specification.

The API documentation approach offers a straightforward way to share API specification with developers and customers. The SDK generation provides a more complete solution for the users to connect to the API. The case company's current approach is to provide API documentation for developers and customers.

2.1 Erlang generation

In the first paper, Donato and Qin [7] addressed the challenge of manually validating OpenAPI-based HTTP requests within a 5G core network. Automated code generation presented a practical alternative to manual implementation, which is often time-consuming and susceptible to human error.

The authors used OpenAPI 3 generator to produce Erlang code, even though its Erlan-specific module was immature and poorly maintained. Their research focused on integrating this generated code for HTTP request validation into Ericsson's internal test environment. The integration was completed with only minimal changes, and the process revealed two bugs and several implementation inconsistencies. The outcome highlighted the generator's value by enforcing compliance with the specification and improving overall system quality.

A primary challenge revealed by the study was the poor quality of the input OpenAPI specifications. These documents contained errors, such as duplicate operation IDs or broken external references, which required manual intervention. The generator itself also had limited capabilities, particularly in its handling of advanced OpenAPI 3 features. The authors concluded that because of these shortcomings, the technology is ready for testing but not for production. The work led to several improvements in the Erlang generator. This included fixing bugs, reorganizing the code, improving features like regular expressions and Enums, and adding custom modules for more flexible integration.

2.2 Automated test cases

In the second paper, Ed-douibi et al. [10] proposed a model-driven approach for automatic test case generation for RESTful APIs. The challenge presented by the authors is the absence of a fully automated approach for generating comprehensive test cases for REST APIs, as existing tools lack fault-based tests and require manual data input. Their solution automates RESTful API testing by using a model-based process to transform an OpenAPI specification into an intermediate test model, which is then converted into executable test code.

The authors' approach first extracts an OpenAPI model and then enhances it with information gathered from provided examples and defaults. This enriched model is then used to generate a TestSuite model, which in turn is used to produce the final executable test code (such as JUnit tests). The solution can produce a test suite that includes cases with both correct and incorrect data, thereby providing more complete coverage.

The validation dataset was composed of 91 OpenAPI definitions, sourced from the APIs.guru collection. An average coverage of 76.5% for OpenAPI definition elements was achieved by the generated test cases. The validation revealed that 40% of the tested APIs failed, indicating underlying issues in either their OpenAPI definitions or server implementations. The observed failures included incorrect HTTP status codes, schema non-compliance in responses to valid data, and unexpected server responses to invalid data.

2.3 Generating user interfaces

In the third paper, Koren and Klamma [6] described a model-driven methodology to automate the generation of responsive web user interfaces from OpenAPI specification. The authors were motivated by the tendency of current tools to generate overly technical UIs, which are not suited for non-technical stakeholders. The primary goal was to reduce the complexity and time required for creating user interfaces. This would allow the designers and end-users to abstract away technical details and rapidly prototype applications.

The author's solution uses the Interaction Flow Modelling Language (IFML) as an intermediate presentation. The process begins by converting an OpenAPI specification into an abstract IFML model. This initial step leverages the structured nature of the OpenAPI document to systematically map its constructs to corresponding IFML view components such as Lists, Forms and Details Views. In the second step, the IFML model is transformed into a concrete frontend built with HTML5 and JavaScript. This solution highlights that a well-defined API is essential for any kind of automation.

The generated web frontend uses Web Components and follows Material Design guidelines to ensure usability across different devices. Although the tool automates much of the UI generation, it requires manual changes for context-specific details like hiding a password field. The need for manual adjustments shows that full automation is a difficult challenge. A key contribution of the study is the systematic mapping of API data structures to visual UI elements.

3 Developer Experience

Developer Experience (DX) examines the challenges in developer’s daily work and their feelings about their job [11]. A good DX minimizes friction, which improves developer focus and productivity, while a poor DX introduces friction that slows delivery, harms morale, and increases turnover [12]. DX is conceptually similar to user experience (UX) or customer experience (CX). However, while UX focuses on the end-user of the software and CX on the consumer of a product or service, the DX focuses on the developer's experience with their tools and processes [13].

3.1 Psychological dimensions

In the first paper Fagerholm and Munch [13] introduce their definition on DX. They argue that human factors are critical for software development success and that improving DX will lead to better project outcomes. The paper provides a conceptual framework that describes Developer Experience as having cognitive, affective and conative elements.

The cognitive dimension covers the aspects that affect how developers intellectually perceive the development infrastructure. These include using development tools and carrying out software tasks. A positive outlook on these aspects is noted to improve DX.

The affective dimension relates to the factors that affect how developers feel about their work. Feeling respected and included creates a sense of security. Being connected to others, teams, or routines also plays a role. A better DX is linked to positive emotions.

The conative dimension is comprised of the factors affecting how developers perceive the value of their contribution. Aligning a developer's personal goals with their project goals can increase their sense of purpose and motivation, which then results in a better DX.

3.2 Productivity and DX

In the second paper Noda et al [14] suggest that instead of focusing on traditional metrics like tracking output or task completion time, leaders should improve developer productivity by focusing on how developers feel about their work. They propose a framework where the DX is composed of three dimensions: feedback loops, cognitive load, and flow state.

Feedback loops are the time developers spend waiting for responses from systems like code compilations or test runs. Fast feedback loops help developers work efficiently by reducing obstacles. Slow feedback loops disrupt progress, leading to frustration and delays. Organizations can improve efficiency by speeding up tools like build and test processes and making human hand-offs smoother.

Cognitive load describes the amount of mental processing required for a developer to perform a task. The difficulty grows with complex tasks, unfamiliar frameworks, poorly documented code, and challenging systems. Organizations can reduce cognitive load by removing unnecessary obstacles, organizing code and documentation clearly, and offering simple self-service tools.

Flow state is defined as a mental state of energized focus, full involvement, and enjoyment. Achieving a flow state often leads to higher productivity, greater innovation, and professional growth. Flow state can be negatively affected by interruptions, delays, and unclear goals, which in turn hurts productivity and focus.

The authors suggest that an effective measurement of DX requires combining subjective feedback from developers with objective data from the systems they use. Surveys offer a quick way to gather subjective feedback from developers, but they must be carefully designed to yield accurate and reliable results.

3.3 DX Factors

In the third paper Greiler et al [15] looked at DX to understand what affects developer’s productivity, satisfaction and retention. The authors created a framework for improving DX based on semi-structured interviews with 21 developers to identify key factors influencing their work experience. The resulting framework organizes these influencing factors into four main categories:

- Development and release: Codebase health, tools and release process.
- Product management: Clear goals, reasonable deadlines and the ability to provide business value.
- Collaboration and culture: Supportiveness, psychological safety and knowledge sharing.
- Developer flow and fulfilment: Autonomy, stimulating work and career growth.

Authors state that these factors are often hindered by “Barriers to Improvement”, such as low prioritization, lack of ownership, and organizational politics. The developers overcome these with individual and team strategies, which include speaking up, creating transparency and using metrics. If the developers cannot attain the improvements, they use “Coping Mechanisms” like reduced engagement, focusing on personal projects or leaving their job. The authors also note that “Contextual Characteristics” like seniority and company goals influence which factors developers perceive as most important.

4 Research method

This thesis uses the Design Science Research Methodology (DSRM) as its research method. The DSRM is a structured approach specifically developed for conducting design science research in fields like Information Systems. DSRM involves first identifying a problem and then creating and evaluating an IT artifact as a solution. This methodology provides a defined process composed of the design and development of an artifact, demonstrating its utility, evaluating its performance, and sharing the findings. A key benefit of DSRM is its establishment of a shared framework and mental model for research structure and reporting for design science [16].

The methodology outlines six interconnected activities:

1. Problem identification and motivation
2. Definition of objectives for a solution
3. Design and development of the artifact
4. Demonstration of the artifact in use
5. Evaluation of the artifact's effectiveness
6. Communication of the research findings

Adopting this methodology provides a clear direction for the research process and a template for presenting the research outputs. The DSRM allows for flexible entry points, permitting to begin at step 1, 2, 3, or 4. These research entry points have been given names by the authors:

1. Problem-centred initiation
2. Objective-centred solution
3. Design & Development centred approach
4. Client/Context initiated

As the problem identification and motivation had already been done by the case company prior to this thesis, the work began at step 2. This approach is known as Objective-centred solution.

In the first step, the research problem is defined and the value of finding a solution is justified. The problem that the case company had identified will be presented in the chapter 4.1. In the second step, the quantifiable or qualitative objectives of the solution are inferred based on the problem definition. This is conducted on the chapter 4.2. In the third step, the artifact is designed and developed. This involves determining its functionality, structure and then building it. This is done on the chapter 4.3. In the fourth step, the artifact is applied to a problem instance and its effectiveness as a solution is demonstrated. This will be done on the chapter 4.4. In the fifth step, the artifact's effectiveness as a solution is examined and measured. This thesis uses a survey to analyse the artifact's impact on developer workflow, and it will be done on chapter 4.5. In the sixth step, the research findings are shared with relevant audiences, and it will be scoped out of this thesis.

4.1 Problem identification and motivation

The case company's product is a modern web application provided as a SaaS solution. The backend of the application is implemented in .NET Core. The frontend is split into multiple applications: Inferno and React based single-page implementations. The React-based applications are fully compatible with TypeScript and represent the future development standard for the project. Therefore, the requirements for the type generation tool will be based solely on these React-based applications.

The backend uses the NSwag [17] library to annotate the C# codebase and then automatically generate an OpenAPI documentation page. This page describes each API route, detailing its parameters, responses, and associated data models. It is used by both internal developers and external customers who use it to build integrations. The development team for the case application is organized into frontend and backend specialists. Once the backend developer has implemented a new data model, the frontend developer can use the OpenAPI documentation to learn about it and translate its properties into TypeScript equivalents. The manual nature of this process makes it susceptible to human error. Additionally, ensuring compatibility with API changes is an ongoing maintenance burden.

The case problem was first noted in a thesis by Heiskanen, which details the migration of a JavaScript codebase to TypeScript [5]. At the time, no existing open-source tool was found that could meet the case application's specific needs. In the absence of a suitable solution, the integration of an automatic type generation tool into the codebase was scoped out of that study.

A common problem with manually created models is that they can unintentionally diverge from the OpenAPI specification over time as changes are made [18]. A primary example of this problem is when a backend developer makes a small, non-breaking change to the data model without informing the frontend team. It is unclear how many of the case application’s frontend models are out of date. Therefore, an automated tool that generates these API models can increase both productivity and the DX. An evaluation of the tool can help demonstrate the practical benefits of automation.

4.2 Objectives

This thesis aims to develop a tool that allows developers to automatically generate TypeScript types for the frontend. The developed tool generates these types from the OpenAPI specification, which provides a language-independent description of the API interface. The tool's usage will be evaluated to assess its impact on Developer Experience and productivity.

Objectives of the code generation tool are following:

- Improved DX: The tool eliminates the manual process of writing and updating types. This allows developers to focus on the core application logic.
- Ensuring type safety and consistency: The tool keeps data structures aligned between the frontend and backend. Any change to the API schema is immediately reflected in the generated types, allowing errors to be highlighted and fixed pre-emptively.

The tool should reduce developers cognitive load by automating the creation of frontend types, which would otherwise need to be memorized and written manually. This automation would allow developers to focus their efforts on solving core application problems rather than on repetitive tasks. It would also shorten feedback loops, as generated types provide more immediate code validation which allows errors to be discovered and resolved earlier. By reducing cognitive load and flow interruptions, the tool could help the developers remain more focused on their tasks. This increased focus can lead to productivity gains and more effective problem-solving. The result is a more efficient development process [14].

4.3 Development

The development of the artifact will be organized into multiple phases. In chapter 4.3.1, different approaches that could be utilized for acquiring the solution are examined. In chapter 4.3.2 the chosen approach is used to find different candidates and select a solution. In chapter 4.3.3, the selected solution is modified to meet all specified requirements. The chapter 4.3.4 details the process of integrating automatically generated types into the codebase to replace previous manually written versions. The chapter 4.3.5 outlines the steps for publishing the tool and implementing it in the case application’s repository.

4.3.1 Tool acquisition strategy

The evaluation of various procurement strategies must be done when there’s a need for a new software component. Building and maintaining software components can be expensive, so it is critical to focus development effort on what creates the most value for customers. The primary strategies for acquiring a new software component are: In-house development, outsourcing, commercial-off-the-shelf and open source-software [19].

In-house development of the custom tool provides the distinct benefits of a perfect fit for the organization's needs, total control over its evolution, and no licensing costs. However, this approach is limited by several factors, including the high cost of experienced developers, the potential for unpredictable expenses, a slow and gradual development cycle, and a reliance on limited internal resources for maintenance and support [20]. It is important to determine if the tool will be developed solely for internal purposes or as a product that delivers value directly to customers. A clear return on investment (ROI) from the tool would justify a larger investment, making an in-house build a more viable option.

Outsourcing tool development presents both benefits and risks. The advantages include control over the tool's evolution and the ability to supplement the team's internal competencies [21]. On the other hand, the drawbacks fall into two main categories. Control risk involves a loss of independence in making decisions or managing resources. The second category is opportunism, where a vendor might act in a self-serving manner by misleading the client or creating barriers to switching competitors [22].

Buying a tool (Commercial Off-the-Shelf) provides immediate availability, quick response time from the vendor and allowing for fast deployment. The most notable downsides include licensing costs and the lack of control over the tool's development and source code [21]. Few commercial SDK generation products are on the market, notably Stainless [23] and Speakeasy [24]. Their core features include multi-language SDK code generation and a suite of tools for managing the lifecycle of the created packages. The investment required for these products is difficult to justify in the context of the case application's modest use case.

The primary benefits of using an open-source solution are noted to be the absence of licensing fees, access to a high-quality product, adherence to open standards, and the avoidance of vendor lock-in [25]. This approach leverages community support and contributions, although the value of this is dependent on how quickly the community addresses bugs and other problems. The success of extending an open-source tool depends on having a skilled in-house team to handle development and maintenance.

The selected tool in this study was required to meet the following functional and non-functional requirements:

- **Maintainability**: The tool must require minimal maintenance, as extensive support cannot be guaranteed after the completion of this thesis.
- **Granular file structure**: Generated files must follow flat folder structure, with one data model per file. This approach makes it easier to differentiate changes in pull requests, improving code review processes.

- **Consistent style**: Generated files must follow the existing file naming conventions of the codebase. A consistent code style across the project should be enforceable using linting and formatting tools.
- **Generated output**: The implementation focuses on generating model schemas (i.e., TypeScript types). It allows a drop-in replacement for the manually written types.
- **Overridability**: The tool must provide the ability to override specific models or properties. This is necessary to apply temporary workarounds for OpenAPI specification inconsistencies while permanent backend fixes are developed. It also allows frontend features to be developed before the backend is ready.

Considering the resource limitations, the most pragmatic approach for this use case was to leverage an open-source tool that aligned with the requirements or could be customized accordingly.

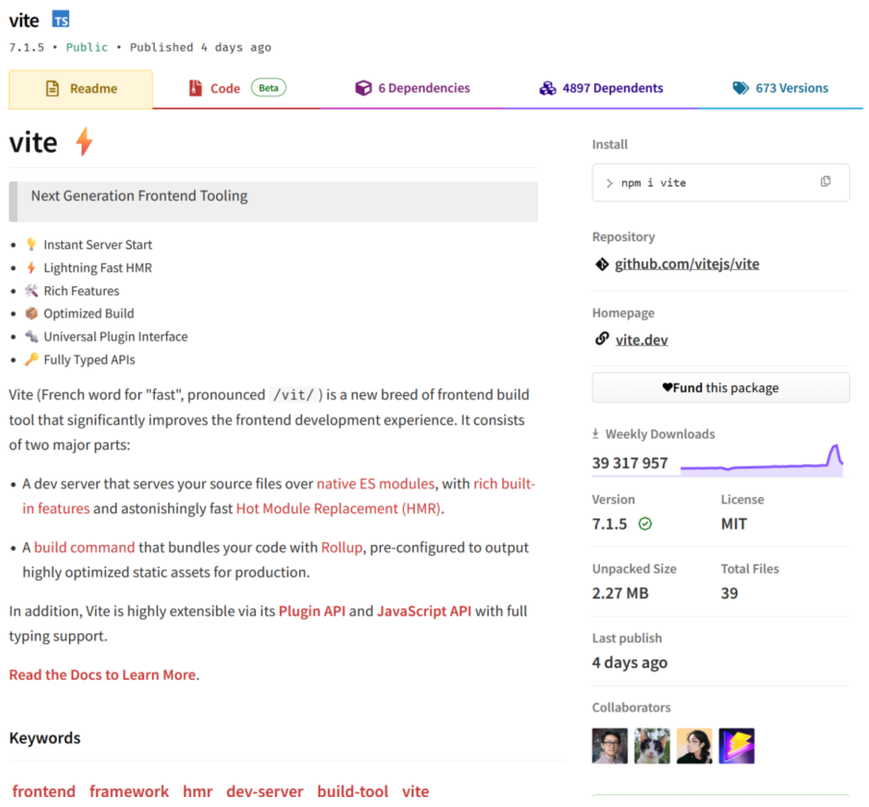
4.3.2 Evaluation of open-source packages

The appeal of an open-source solution lies in benefits, such as its minimal initial cost, its potential for customization, and its access to a broader community for support and security. A contributor can gauge an open-source project's health by its number of stars and forks, the quality of its documentation, recent commit activity, and the responsiveness and politeness of its maintainers [26]. The long-term viability of a package is a critical factor, as choosing one that will soon be deprecated introduces risk.

For this reason, a crucial first step was to specify the package's requirements and evaluate potential open-source solutions against them prior to committing to one. For the case application, the selected package was required to meet the following criteria:

1. Satisfies most of the requirements and can be customized to achieve rest of them
2. Active maintenance and support from a strong community
3. Permissive open-source license
4. Evidence of project health and longevity

Modern web frontend development relies heavily on the JavaScript ecosystem, which makes the npm [27] a required component. A dedicated webpage for each npm package displays essential information regarding its current status. Figure 1 shows the Vite package as an example.



vite TS
7.1.5 • Public • Published 4 days ago

Readme | Code Beta | 6 Dependencies | 4897 Dependents | 673 Versions

vite ⚡

Next Generation Frontend Tooling

- Instant Server Start
- Lightning Fast HMR
- Rich Features
- Optimized Build
- Universal Plugin Interface
- Fully Typed APIs

Vite (French word for "fast", pronounced `/vit/`) is a new breed of frontend build tool that significantly improves the frontend development experience. It consists of two major parts:

- A dev server that serves your source files over native ES modules, with rich built-in features and astonishingly fast Hot Module Replacement (HMR).
- A build command that bundles your code with Rollup, pre-configured to output highly optimized static assets for production.

In addition, Vite is highly extensible via its Plugin API and JavaScript API with full typing support.

Read the Docs to Learn More.

Keywords

frontend framework hmr dev-server build-tool vite

Install

```
> npm i vite
```

Repository
github.com/vitejs/vite

Homepage
vite.dev

♥Fund this package

Weekly Downloads
39 317 957

Version	License
7.1.5	MIT

Unpacked Size	Total Files
2.27 MB	39

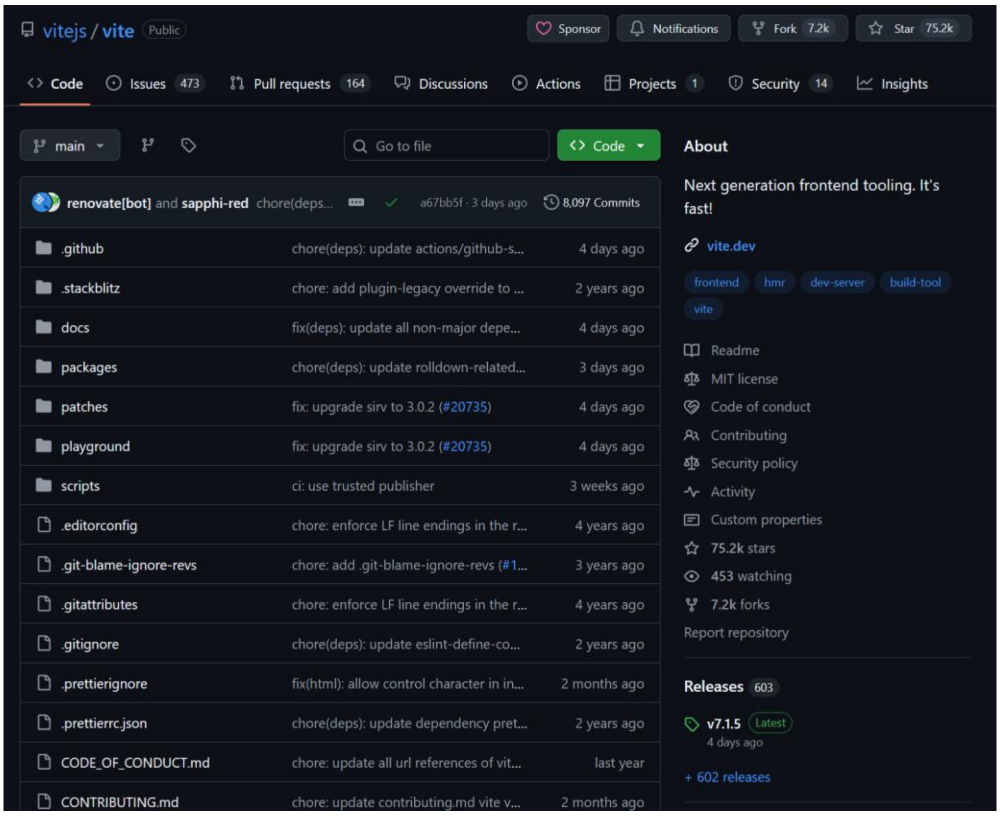
Last publish
4 days ago

Collaborators

Figure 1. Vite frontend tool npm package website

The package's name and an icon indicating its TypeScript usage are displayed in the top bar. TypeScript packages offer built-in documentation, since all data structures are explicitly defined within the source code. Since the adoption of TypeScript is now widespread among these types of tools, its use is considered a baseline rather than a distinguishing feature for comparison. The dependency count is a valuable metric shown in the tab, as fewer dependencies suggest lower maintenance risk and less performance overhead. The right sidebar contains information, such as weekly downloads which is an important measure of popularity and an indicator of the package's long-term health. The license of the package, shown in the sidebar, is a critical detail because it defines how the software can be legally utilized. Permissive licenses, such as MIT or Apache 2.0, are suitable because they place minimal restrictions on modification and distribution. The GNU GPL licenses should be avoided because its terms require any application using a GPL component to release its own complete source code, which is unacceptable for a proprietary SaaS product [28]. The last publish date serves as a metric for quickly assessing whether a package is still under active development. However, an accurate assessment of a package's development activity requires a look at its source control repository.

This thesis focuses more closely on GitHub, as it is the most common platform for open-source software. Figure 2 depicts the GitHub page for the Vite project. A look at the project's GitHub page can reveal important details such as the coding standards it follows and the typical response time of its community to bug reports.



renovate[bot] and sapphi-red chore(deps... a67bb5f · 3 days ago 8,097 Commits

.github	chore(deps): update actions/github-s...	4 days ago
.stackblitz	chore: add plugin-legacy override to ...	2 years ago
docs	fix(deps): update all non-major depe...	4 days ago
packages	chore(deps): update rolldown-related...	3 days ago
patches	fix: upgrade sirv to 3.0.2 (#20735)	4 days ago
playground	fix: upgrade sirv to 3.0.2 (#20735)	4 days ago
scripts	ci: use trusted publisher	3 weeks ago
.editorconfig	chore: enforce LF line endings in the r...	4 years ago
.git-blame-ignore-revs	chore: add .git-blame-ignore-revs (#1...	3 years ago
.gitattributes	chore: enforce LF line endings in the r...	4 years ago
.gitignore	chore(deps): update eslint-define-co...	2 years ago
.prettierignore	fix(html): allow control character in in...	2 months ago
.prettierrc.json	chore(deps): update dependency pret...	2 years ago
CODE_OF_CONDUCT.md	chore: update all url references of vit...	last year
CONTRIBUTING.md	chore: update contributing.md vite v...	2 months ago

About

Next generation frontend tooling. It's fast!

vite.dev

frontend hmr dev-server build-tool vite

Readme
MIT license
Code of conduct
Contributing
Security policy
Activity
Custom properties
75.2k stars
453 watching
7.2k forks
Report repository

Releases 603

v7.1.5 Latest
4 days ago

+ 602 releases

Figure 2. Vite frontend tool GitHub repository

The number of GitHub stars is displayed right in the top right corner, and it serves as a great indicator of a package's popularity. Metrics, such as commit history, open issues, and contributor count, provide insight into a project's overall size and activity level. The quality of a package's maintenance can be assessed by reviewing its “Pull Requests” and “Issues” sections, where prompt responses and active dialogue indicate a healthy project.

The four candidates for TypeScript type generation were “orval” [29], “openapi-typescript” [30], “@openapitools/openapi-generator-cli” [31], and “@hey-api/openapi-ts” [32]. The first step was to analyse these open-source tools. This involved documenting the pros and cons of each option to measure the suitability for the case application. Statistics related to each package can be seen in the Table 1.

Table 1. Statistics of the selected open-source packages

Name	**Last publish**	**Weekly downloads**	**Depen dencies**	**Commit Activity (Last 3 Months)**	**License**
orval	1 month ago	350 943	26	65 commits by 23 contributors	MIT
openapi-typescript	1 month ago	1 574 772	6	21 commits by 12 contributors	MIT
@openapitools /openapi-generator-cli	17 days ago	944 428	17	45 commits by 4 contributors	Apache-2.0
@hey-api/openapi-ts	1 day ago	462 252	9	418 commits by 30 contributors	MIT

A primary focus on SDK client generation was observed across all selected libraries. A single file containing all API models was generated by the “orval”, “openapi-typescript”, and “@hey-api/openapi-ts” packages. In contrast, a single file for each model was generated by the “@openapitools/openapi-generator-cli” package. However, its dependency on Java was considered a drawback. The “@hey-api/openapi-ts” library was found to be extensible with plugins, which would allow it to be adapted to custom requirements. The default implementation of this library contained several unnecessary features for the case application's use case, which would need to be removed.

All of the examined libraries were found to lack important features, necessitating modifications to meet the case application's requirements. A significant advantage of the “@hey-api/openapi-ts” library was its plugin system. A custom plugin for this library was thought to be able to provide the necessary functionality to meet all of the case application’s requirements. Consequently, this library was considered the most promising solution and was chosen as the foundation for the automatic type generation tool. The next chapter details the modifications required for the “@hey-api/openapi-ts” library to fulfill the requirements.

To provide more context for the choosing the type-only generation approach, it is beneficial to examine the more common client SDK generation approach. A client SDK simplifies the process of making API calls by automatically handling tasks like authentication, request formatting, and response parsing. This functionality is provided to developers through a set of automatically generated methods. This means developers no longer have to write each API request by hand. Automatic generation helps to enforce best practices and reduce the errors that come from manual coding [33]. These tools can generate clients for numerous programming languages (e.g., Python, Java, and C#) since the OpenAPI specification is language-independent. The benefits of client SDKs are not limited to the internal development team as they can also be provided to customers to help them build integrations.

The main challenge in implementing a client SDK for the case application lay in the significant refactoring required for its existing network functions. These functions would have been difficult to replace due to their support for numerous features and edge cases. Another key point was that an SDK intended for customers would have had to be published as official versioned package, which would have required ongoing maintenance from the development team. Due to this complexity, the client SDK generation approach was considered outside the scope of this study. This thesis instead focuses on the simpler type-generation approach. However, for a new application built from the ground up with more flexible requirements, the SDK client generation approach could be a more suitable choice.

4.3.3 Extending selected open-source package

The “@hey-api/openapi-ts” package was identified as a candidate that could meet the application's main requirements. The package showed indicators of longevity and relevance; its commit history dated back to 2020, and its download metrics surpassed 611,000 per week [32]. The library's primary strength was its extensibility, which is provided through a plugin system that allows new features to be built upon the existing foundation.

The package utilizes jiti library [34] to configure its various built-in settings. The configuration provides control over aspects, such as the OpenAPI input file and the output directory structure. It was determined that this configuration scheme could be extended via a custom plugin to add the options required for the case application. This configurability was seen as an advantage, because it would allow the tool to be deployed across multiple frontend repositories in the future.

The library provides core components for generating TypeScript types and SDKs, transforming data formats, and validating data. The “typescript” plugin was the most relevant part of the library for the use case, because its purpose is to generate the TypeScript types. The default behaviour generates a single “types.gen.ts” file containing all models as TypeScript type aliases.

The library is primarily used for SDK client generation. This functionality is managed by a dedicated plugin responsible for the generation process. The library provides SDK client generation for multiple targets, including Fetch, Angular, Axios, Next.js, and Nuxt. The SDK client generation feature was considered outside the scope of this thesis.

The transformer plugin is used to transform the structure of data during the application's runtime. A practical use for this is to parse date values that are transmitted as strings in a JSON payload and transform them into JavaScript Date objects. This feature was not utilized within this thesis, but its exploration is recommended for future work.

The validation plugin is compatible with a number of popular validation libraries, such as Valibot and Zod. If the data provided by an API cannot be guaranteed to be valid, the plugin can validate the response structure at the frontend runtime. The validation plugin was omitted from the tool because its features were not needed.

The official documentation warned that the plugin API was subject to breaking changes, which indicated that the system was not yet considered stable. This was an important consideration, as the unstable API could require extra maintenance work.

The base “typescript” plugin contained features that could be modified to meet the case application’s requirements. An analysis of the source code revealed that the library needed to be forked to utilize its internal functions. The required modifications are detailed in the following sections.

The implementation was first simplified by removing all non-essential plugins, so that the “typescript” plugin was the sole remaining component. The library was tested with the case application’s OpenAPI specification, with the goal of finding any surprising characteristics in the generated code. The most notable incompatibility that needed to be addressed was the generation of a single type file, as the case application requires one file per model. An investigation of the library's GitHub repository revealed an eight-month-old open issue that suggests adding a configuration setting that would allow the user to choose the desired output structure for the type file [35]. It may eventually be possible to deprecate the custom solution developed in this thesis in favour of an official implementation from the library. However, there was no indication of when this official feature might be developed. Another necessary change was to alter the library’s default behaviour to produce interface declarations rather than type aliases. By default, the library also had a feature enabled to generated “Read” and “Write” variants for models with read-only fields. A configuration option named “parser.transforms.readWrite” was used to disable this behaviour. There was also a need to alter the file naming scheme to conform to the codebase's standard of using the lowercase schema name for each filename. The default behaviour of the plugin was to generate both operations and schemas. As the scope of this thesis was limited to schemas, the operation generation feature was disabled.

The implementation of the custom plugin began by copying the contents of the “@hey-api/typescript” folder to be used as a foundation. The custom plugin was given the name “@hey-api/casesoftware”. The next step in the process was to add the plugin's configuration object and name to two files: “plugins/index.ts” and “plugins/types.d.ts”. Once registered, the custom plugin could be activated in the configuration file to replace the default “@hey-api/typescript” implementation.

The “typescript” plugin’s core functionality was located inside the “plugin.ts” file. The functions that turn types into their corresponding identifiers were a central part of this implementation. The supported types include array, boolean, enum, integer, number, never, null, object, string, tuple, undefined, unknown, and void. To create type aliases, the “objectTypeToIdentifier” function relies on the “compiler.typeInterfaceNode” function. The requirement to produce interface declarations was addressed by creating a new function, “compiler.interfaceNode”.

The base implementation contained a “plugin” object, which held a “createFile” function. This function was responsible for creating the “types.gen.ts” type file. The name of the generated file is determined by a “fileId” property passed to the function. A change was introduced to the implementation to support the generation of one file per schema. The modification ensures that every schema receives its own unique “fileId”, rather than using one constant identifier for all of them. This change resulted in the generation of multiple type files with names that adhere to the following format “modelname.gen.ts” The “.gen” suffix served to identify files that have been automatically generated. The library itself also provided a header comment to indicate a generated file's origin: “This file is auto-generated by @hey-api/openapi-ts”. This, along with the plan to store the output in a folder named “generated”, was deemed an adequate way to identify the generated files. Therefore, the library was altered to omit the “.gen” suffix, which ensured that the filenames aligned with the case application’s existing naming convention. The move towards a model-per-file structure, introduced the requirement to import sub-model dependencies from other files. The solution was straightforward because the type files were contained in a flat directory and followed a consistent naming structure.

There were known instances where data returned by the case application's API did not align with its corresponding OpenAPI specification. A prime example of this problem was the handling of null values. In some cases, properties were defined as nullable but never produced a null value, while in other instances, non-nullable properties occasionally returned null. Handling these discrepancies required a feature that could override the automatically generated models. This method was intended only as a temporary workaround, as the long-term solution was to correct these discrepancies in the backend itself. Another reason to develop the override feature was to support parallel development, which allows work on the frontend to start before the backend was complete.

Several different approaches were considered for achieving the type overriding feature. The tool's configuration object could be extended with a new property for defining how to override models and their properties. The downside of this approach would have been that the configuration file could have grown to hundreds or even thousands of lines, making it difficult to maintain.

Another considered method was to use TypeScript's own syntax to extend or change the generated types. This approach would involve the developer creating an override file which shares the same name as the equivalent generated file. This override file contains an interface that can then be extended or replaced to correct inconsistencies. The need for a separate override file was initially seen as a potential drawback. This was because creating a new file and using it to replace the generated model could require extensive changes to the import paths inside files that use the model. This would have resulted in large and difficult-to-review pull requests.

To address this drawback, the proposed solution was to generate a default override file for every model. The codebase would always import from these override files, whether they contain modifications or not. If a model does not require any changes, its override file would simply be an empty interface that extends from the generated model. The override file for an Enum schema would be a type alias that references the original generated type.

The benefit of this approach was that import paths never need to be changed. Applying an override only requires editing the specific override file, which leaves the rest of the code untouched. A feature was added to the tool, that would create an empty interface or type alias when a new schema was detected. This ensured that developers do not have to manually create overwrite files for any new models appearing in the OpenAPI specification.

A nullability issue was found in the OpenAPI specification where some sub-models were incorrectly defined as non-nullable, despite the actual API sometimes returning them as null. This inconsistency can cause frontend errors as the sub-model can be unexpectedly undefined. A new configuration option was added as a temporary workaround to make all sub-models nullable. A discussion with the backend team revealed that adjustments to the Swagger annotations could be made to properly define the nullability of the sub-models. Implementing this fix was planned but fell outside the scope of the thesis.

A few models were generated as empty interfaces due to their definitions in the OpenAPI specification. An investigation found no usage of these models within the case application. The base library's parser feature was used to remove the unneeded models from the generated output. This configuration option could be used to ignore problematic schemas in the future, which would prevent frontend type-checking errors. The origin of these empty models would be investigated in the future in collaboration with the backend developers.

The ability of a C# string to hold a null value led to it being marked as nullable in the OpenAPI specification. This required frontend developers to write null checks even for properties that were known to be non-nullable. For instance, the GUID property of a model was always a defined string, yet it was marked as nullable in the generated specification. A configuration option was added that overrides the nullability of specified properties, setting the nullability value to false.

The case application should rely on the OpenAPI specification as its source of truth. For this reason, the number of overrides had to be counted to measure how much the frontend types differ from the specification. This would make it possible to notice issues with the OpenAPI specification and discuss with the backend team regarding improvements. A new feature was developed that scans all model files in the overwrites folder and checks if they have been modified. An overwrite file was identified as modified if the type it contained was not the default empty structure. The feature leveraged the Node.js file system module to access files in the "overwrites" folder and the TypeScript compiler API to inspect their contents. The filenames of all modified files would be logged to the console every time the tool is run.

4.3.4 Implementing the generated TypeScript types

The case application's repository contains four distinct applications: web1, an older Inferno-based web application; web2, a more recent React-based application; trial, a small application for customers creating new environments; and mobile, a React Native application for mobile devices. There is also a React-based administrator application, but it is in a separate repository and is not covered in this thesis. The main mono-repository contains functionality that is shared across all applications. These common utilities are used for tasks like data storage and manipulation. The testing files are implemented in JavaScript, which means they cannot use API models. However, the mock data used in testing is kept in TypeScript files and API models are used to validate their shape. A simplified overview of the frontend architecture is presented in Figure 3.

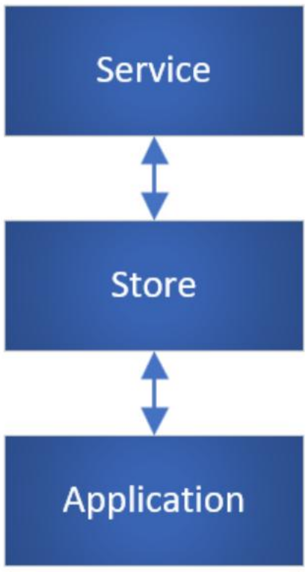


Figure 3. Architecture diagram of the case application

The service layer contains the network functions that are used to make calls from the frontend to the backend API. Therefore, it is the first layer in the frontend that interacts with the data models from the API. The store layer's primary responsibility is to store, manipulate, and distribute data to the application's visual components. This part of the architecture is notable for its complexity. It implements the various rules that allow data to be manipulated by users through the application layer. The application layer is responsible for composing and rendering the user interface.

The custom data manipulation framework is built with the Flux architecture and relies heavily on object-oriented principles. The framework utilizes inheritance by offering multiple base classes in the service and store layers. These classes encapsulate common functionalities for the classes that extend them. TypeScript generics are used heavily in base classes, which makes the code more reusable and decoupled. The domain classes inherit from base classes, and they contain business logic which requires the use of API models. Almost every API model has a corresponding service and store domain class.

Most of the older codebase was written in JavaScript, which meant it could not utilize the generated TypeScript API models. The process of refactoring the older codebase to TypeScript was still ongoing at the time, but the most important base and domain classes had already been converted. All new service and store files were written in TypeScript, which allowed them to use the generated API models. The codebases for the web2 and trial were written in TypeScript which meant that they could get the most out of the generated API models.

A common architectural pattern was that classes and components across different layers operate on the same data entity. This meant that to fully implement the generated types, the focus could not only be limited to modifying the modern web2 application.

The case application's repository had been in continuous development for more than a decade by a team of approximately 5-10 frontend developers. The scale of the repository was significant, consisting of over 4,500 TypeScript files that account for more than 200,000 lines of code. There were approximately 950 manually created API models in use. It was unclear how closely these models aligned with the current OpenAPI specification. The implementation of the new types could introduce errors in any existing file. Therefore, a manual review of all affected TypeScript files was necessary to verify their functionality.

Since all the data models in the system were highly interconnected, it was determined that the implementation of the generated types could not be done incrementally and had to be performed all at once. An incremental migration would have left the codebase in a hybrid state, where incompatibilities between the manual and generated models would have caused type errors. The main requirement for the implementation was that no changes be made to the application's existing functionality. Therefore, the plan was to minimize changes to the current JavaScript functionality and rely on TypeScript syntax to solve any resulting type issues. Any bug fix that would have changed the application's functionality was excluded from the scope of the thesis.

To obtain the types needed for the implementation, the tool had to be configured. The default library implementation offered multiple approaches for Enum generation: TypeScript string unions, TypeScript Enum declarations, and JavaScript Enum objects. The use of TypeScript

Enums is a subject of debate in the development community, since they have been known to cause typing issues [36]. TypeScript string unions were chosen for the implementation, as they are removed during the build process, leaving no trace in the final output. The developed custom configuration options were utilized to enforce specific nullability rules, such as making sub-models nullable and “guid” properties non-nullable, and to ignore schemas that were identified as unused, like “ClientAction” and “ResourceAllocationTotalModel”. Running the generation tool against the main OpenAPI specification resulted in the creation of approximately 950 model files, each with a corresponding override file. The generated models themselves did not contain any type errors. However, integrating these models into the existing codebase introduced 5668 linting and 1784 type-checker errors. Most of the errors could be classified as either incorrect import statements or mismatches in the model's expected shape.

The errors related to imports were expected, since the new generation method results in a different file structure. In the previous implementation, all API models resided within the “types/src/models” folder. The new implementation introduced three subdirectories within the “models” folder: “generated”, “overwrites”, and “manual”. It was found that some models required by the codebase were absent from the OpenAPI specification. These external models for features such as authentication and user tracking still require manual maintenance and were therefore placed in the "manual" folder. The task of modifying the import paths from “types/src/models/” to “types/src/models/overwrites/” was simplified by the Webstorm IDE's built-in refactoring tools. The find-and-replace operation was intentionally scoped to the "types" and "src" folders to avoid unintended changes to other instances of the string "models".

The backend models followed two different patterns: an older method with one model per entity, and a newer one with two models (an input and an output). Input models were typically used for API requests that submit data, such as POST and PATCH, while output models defined the data shape returned in API responses. All new models were being written with the input/output pattern, but a substantial portion of the older codebase still used the single-model approach. The long-term objective of the development team was to refactor the older models to the new input/output pattern. The refactoring had to be approached with caution because some of these models were exposed external customers and partners via a public REST API. Any breaking changes introduced to this public API could have negatively impacted these external parties. This situation had led to a practice where frontend developers had manually created input and output versions for some of the legacy single-models. This practice caused errors because the frontend code's import statements were expecting a dual-model structure, while the new tool generated a single-model structure. This discrepancy could not be fixed by a simple import path change due to differences in the models' properties. Another source of import errors came from naming inconsistencies between the manually created models and the OpenAPI specification. These inconsistencies had been caused by either a simple typo or by an intentional alteration of the model's name:

Simple typographical errors

- ChangeUserLicensesResultModel vs. ChangeUserLicencesResultModel
- RealtimeRegistrationInputModel vs. RealTimeRegistrationInputModel
- WorkWeekSettingsModel vs. WorkweekSettingsModel

Different naming conventions

- FeedbackModel vs. ProductboardFeedbackOutputModel
- UserType vs. LicenseUserType
- AuthToken vs. FreshChatTokenModel

Wrong terminology (Project instead of Case)

- ProjectRevenueRecognitionInputModel vs. CaseRevenueRecognitionInputModel

The lack of a predictable pattern in these discrepancies required a manual review of every TypeScript file in the codebase. It was determined that the most logical order was to first address the issues located in the service layer, then the store layer, and finally the application layer. Since the layers are dependent on one another, fixing the service layer first ensured that its issues did not affect the store layer. In the same way, this prevented problems in the store layer from impacting the application layer.

Although the initial goal was to convert the entire codebase to use generated types, it became apparent during implementation that this was not feasible for all models.

The case application had a custom reporting framework that enabled users to generate various data views in the user interface, such as lists and graphs. The current OpenAPI specification implementation did not include schema definitions for this feature and as a result, no corresponding models were generated. Due to this limitation, it was decided that the reporting framework would continue to rely on the existing manually written models.

Custom properties were another feature that was not correctly represented in the OpenAPI specification. This feature allowed users to add custom user-defined properties to models. The dynamic nature of the backend models in this approach yielded an inaccurate OpenAPI specification, which in turn generated incorrect frontend types. Therefore, the models for custom properties were also left to be manually maintained.

An inspection of the “models” folder revealed that some files were not API schemas, but rather runtime types used for the frontend. This included Enums for things like the list of addons and integrations available in the case application. The folder was also found to contain types for network request URL parameters. In the future, the tool could potentially be configured to create these URL parameters automatically. The files mentioned above were removed from the “models” folder because they were not API models.

Once errors in the service and store layers were fixed, the focus shifted to the application layer. Since the trial application depended exclusively on manually written external models, it was excluded from the conversion process. The mobile application had had a dedicated development team for many years, leading to different practices compared to the web team. This was evident in that they did not use the shared, manually written API models from the “types” package, and instead had chosen to write inline models within the mobile application’s files. Due to the author's inexperience with React Native, modifying the inline models was determined to be outside the scope of the thesis. The older web application was mostly written in JavaScript. Although about 10% of the web files were written in TypeScript, the “any” type was often used in them to avoid type-checking. This resulted in a simple migration process, as a limited number of files used the manual types from the “types” package. The newest web2 application was completely written in TypeScript and, therefore it formed the largest portion of the migration work. The most common type-related issues were cases where an object was accessed without a proper nullability check. Safe object access in these cases was achieved using JavaScript's optional chaining and nullish coalescing syntax.

The generated types were integrated into the case repository via a single pull request. This resulted in 18293 line additions, 8435 line removals and the changes spanned a total of 3,577 files. All frontend developers of the case application were invited to the review process to confirm these changes. The GitHub web interface imposes a limit of 100 file changes that can be reviewed at one time. This restriction posed a challenge to the review process. It was therefore crucial to avoid any JavaScript changes that could alter the application's behaviour beforehand. Any bugs that would require JavaScript changes would be addressed in future sprints, once developers became more familiar with the new models.

4.3.5 Integrating the tool into the case application

A key consideration was how and when the tool should be executed. The case application had multiple environments (local, internal, staging and production) whose OpenaAPI specifications could have minor differences depending on the time. Discussions with developers led to a decision that this tool should run in the local environment, when developers execute the "start" script of the development server. The new generated models would be versioned in Git, using the same convention established for the manual ones. This approach allows the developers to quickly notice any changes that have happened to the OpenAPI specification as they are highlighted as git diffs.

It was important to make the developed tool as modular as possible. The chosen approach was to deploy the automatic code generation tool into its own repository and publish it as a private GitHub package. This approach would allow the tool to be used across all frontend applications, since the administrator application was located in a separate repository. Additionally, this decision separated the tool's maintenance from the rest of the codebase, which provided flexibility for future replacement. For example, if one of the previously mentioned open-source packages were to meet the requirements in the future, the maintenance burden on the development team could be reduced by replacing the custom implementation.

There are various strategies for versioning packages. The npm documentation recommends using the semantic versioning standard [37]. Semantic versioning structures version numbers as “MAJOR.MINOR.PATCH”. The major version denotes backward compatibility breaking changes, minor version adds backward compatible features, and the patch version applies backward compatible bug fixes [38]. This system communicates the scope of changes in each version, helping developers manage updates and required code modifications.

Developers are responsible for applying correct version numbers, as only they can accurately determine the nature of their changes. Determining a correct version number for a release can be hard if there are multiple commits inline and no system in place to categorize them. The chosen approach was therefore to use the "semantic-release" library to automate the publishing workflow and reduce the need for manual intervention [39]. This eases the workload for maintainers, allowing them to focus on writing code instead of managing releases and version numbers.

The “semantic-release” library enforces “Angular Commit Message Convention” by default. The “Angular Commit Convention” defines a structure for commit messages to be written as follows: type(scope): subject [40]. The “type” specifies the category of the change, such as “feat” for a new feature or “fix” for a bug fix. The optional “scope” specifies which part of the codebase is affected. The “subject” is a concise summary of the commit, written in the imperative mood. A breaking change is signalled by including a “BREAKING CHANGE:” section in the footer of the commit message. The case organization's existing commit message convention required that a JIRA ticket number was included in the message. Following the Angular convention, the ticket number belongs in the subject. This results in a commit message with the following format: “feat(core): SID-010 add logging support”.

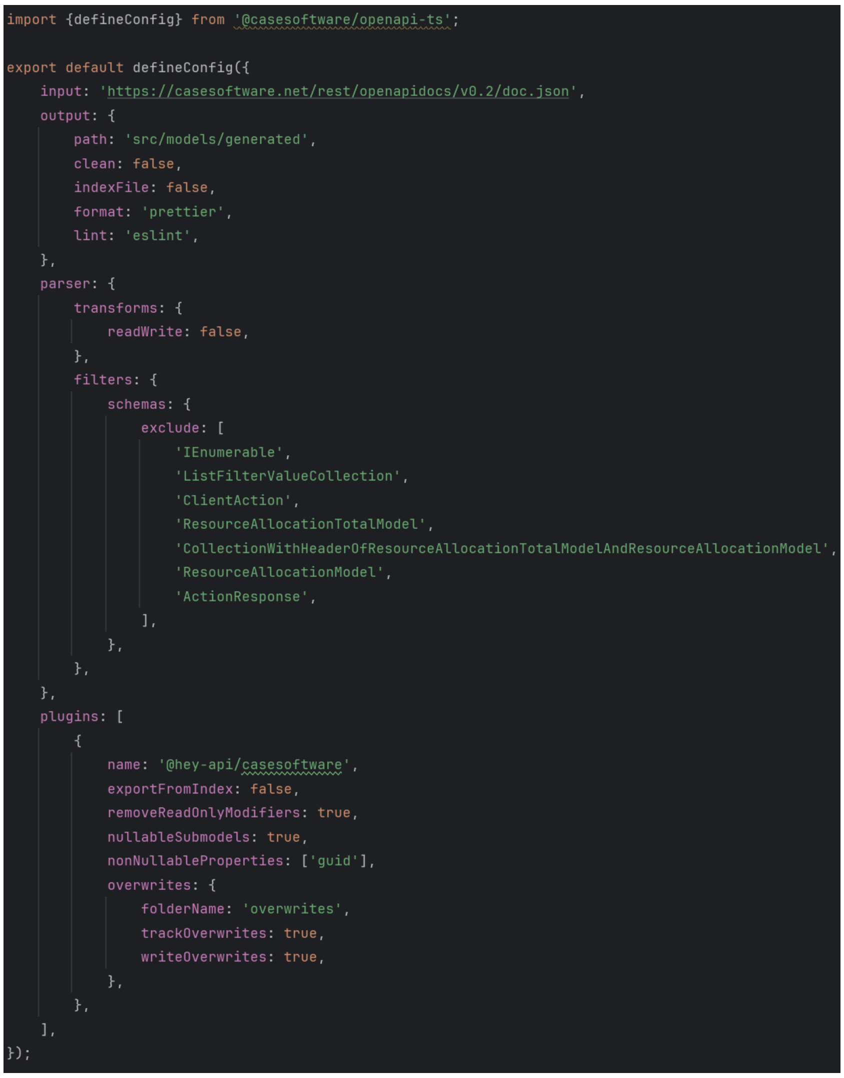
Adhering to a commit convention like the one previously mentioned enables automatic versioning. The “semantic-release” library uses this convention to generate a changelog for each new release. This provides developers with a clear summary of what has changed in any given version. The primary challenge of this kind of commit convention is ensuring its correct application, as errors can result in breaking changes appearing in what are supposed to be backward-compatible releases [38].

The build workflow for the package utilized the default tools of the forked “@hey-api/openapi-ts” library. After authentication for the GitHub Package Registry was configured, the package was built and published, making it available for the case application.

As previously mentioned, the case repository is structured as mono-repository, which contains applications and shared utility packages for them to use. The generated types were located in the “types” package in this mono-repository. Therefore, the configuration of the automated type generation tool was done in this package. The library’s jiti configuration file was used to configure the tool according to the specifications mentioned in the previous chapter. This allowed the developers to manually execute the tool by using “npx openapi-ts” command. The plan was to tie the tool to the case applications development server “start” script. This necessitated the creation of a Gulp.js task named “generate-types” that uses the library’s exposed “createClient” function to be called to generate the types. During the development of this feature, a limitation was identified in the “@hey-api/openapi-ts” library regarding its support for mono-repositories. This issue arose when the tool did not find the linter and formatter dependencies in the execution context. In the case repository, the formatter and the linter dependencies were located in the root folder which caused issues. For the time being this was fixed by adding ESLint and Prettier as dependencies in the sub folders. The default logging messages from “@hey-api/openapi-ts” library, such as "Running formatter" and "Running linter," were considered too generic. They were therefore given an “openapi-ts:” prefix to clearly identify them as output from the type generation tool during “start” script execution.

4.4 Demonstration

The tool was installed from its private GitHub Package registry to mono-repository’s “types” package. The next step involved configuring the tool in accordance with the case application's requirements. The configuration file is pictured in Figure 4. As detailed in the development chapter, many of the tools custom configuration options were created to circumvent inconsistencies of the case application’s OpenAPI specification. Improving the OpenAPI specification itself would have required changes to the backend codebase, which was considered outside the scope of this master’s thesis.

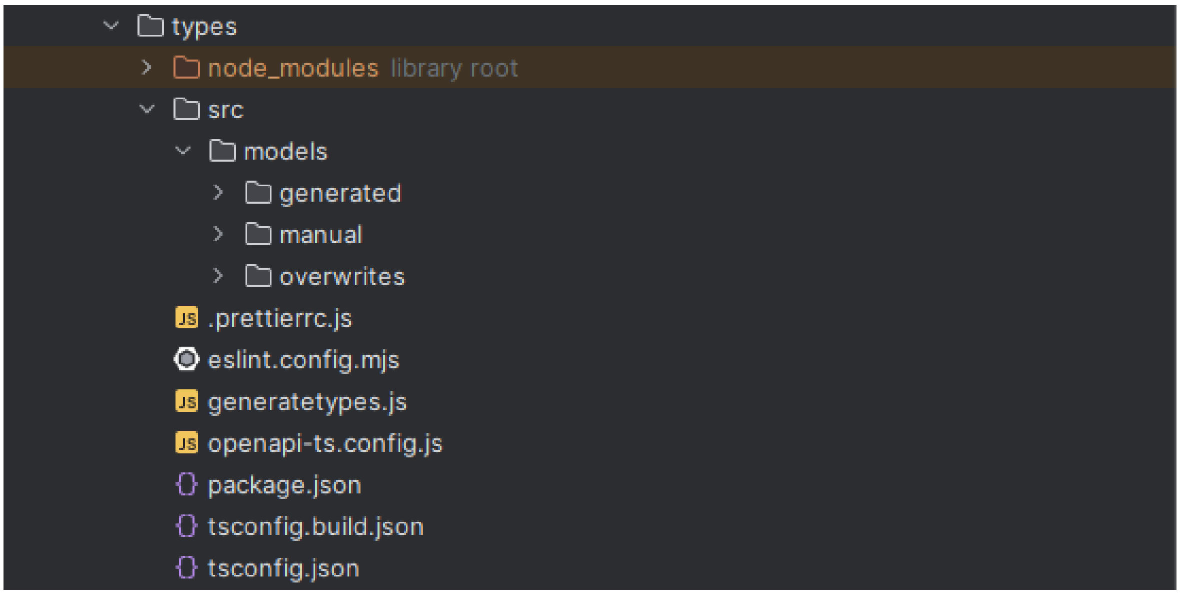


```
import {defineConfig} from '@casesoftware/openapi-ts';

export default defineConfig({
    input: 'https://casesoftware.net/rest/openapidocs/v0.2/doc.json',
    output: {
        path: 'src/models/generated',
        clean: false,
        indexFile: false,
        format: 'prettier',
        lint: 'eslint',
    },
    parser: {
        transforms: {
            readWrite: false,
        },
        filters: {
            schemas: {
                exclude: [
                    'IEnumerable',
                    'ListFilterValueCollection',
                    'ClientAction',
                    'ResourceAllocationTotalModel',
                    'CollectionWithHeaderOfResourceAllocationTotalModelAndResourceAllocationModel',
                    'ResourceAllocationModel',
                    'ActionResponse',
                ],
            },
        },
    },
    plugins: [
        {
            name: '@hey-api/casesoftware',
            exportFromIndex: false,
            removeReadOnlyModifiers: true,
            nullableSubmodels: true,
            nonNullableProperties: ['guid'],
            overwrites: {
                folderName: 'overwrites',
                trackOverwrites: true,
                writeOverwrites: true,
            },
        },
    ],
});
```

Figure 4. The configuration file of the type automation tool

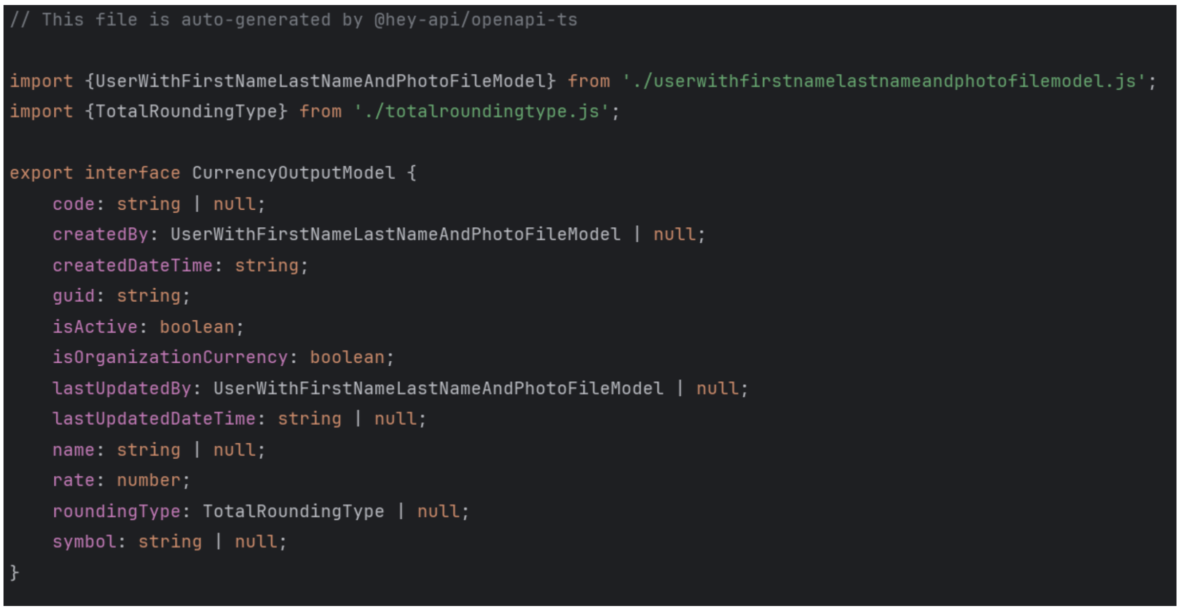
The type generation tool needed to be executed each time the development server was run. The case application utilized Gulp.js to manage build processes. The gulp task named “start” launched the Webpack development server. The “start” script ran supplementary tasks for development work, such as downloading translations and processing stylesheets. The type generation tool was added to this “start” script as its own supplementary task. The creation of the “generated” and “overwrites” folders followed the path and naming scheme defined by the user in the configuration file. The "manual" folder is used for storing types that cannot be generated from the OpenAPI specification, and it was not created by the tool. The application was configured with Prettier and ESLint tools, which required their configuration files to be in the execution folder. These tools ensured that the generated types adhered to the coding style defined by the configuration files in the mono-repository's root folder. The folder structure is detailed in the Figure 5.



```
types
  node_modules  library root
  src
    models
      generated
      manual
      overwrites
  .prettierrc.js
  eslint.config.mjs
  generatetypes.js
  openapi-ts.config.js
  package.json
  tsconfig.build.json
  tsconfig.json
```

Figure 5. Folder structure of the “types” package

The case application’s “CurrencyOutputModel” provides a good example of a backend-defined model. The backend provides the OpenAPI specification that defines a model, such as “CurrencyOutputModel”. The type generation tool consumes the OpenAPI specification to create a two-file system: a base file that holds the generated TypeScript type and a separate overwrite file used to define any custom frontend developer overrides. The generated model file is illustrated in Figure 6, while the corresponding overwrite file is depicted in Figure 7.

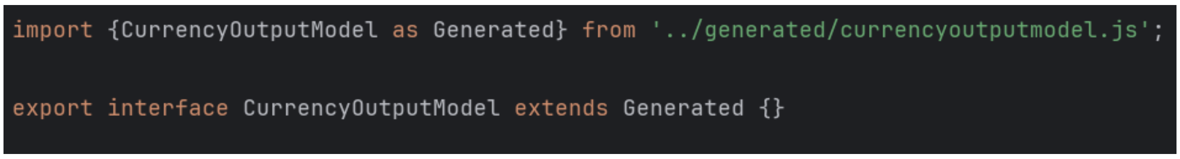


```
// This file is auto-generated by @hey-api/openapi-ts

import {UserWithFirstNameLastNameAndPhotoFileModel} from './userwithfirstnamelastnameandphotofilemodel.js';
import {TotalRoundingType} from './totalroundingtype.js';

export interface CurrencyOutputModel {
    code: string | null;
    createdBy: UserWithFirstNameLastNameAndPhotoFileModel | null;
    createdDateTime: string;
    guid: string;
    isActive: boolean;
    isOrganizationCurrency: boolean;
    lastUpdatedBy: UserWithFirstNameLastNameAndPhotoFileModel | null;
    lastUpdatedDateTime: string | null;
    name: string | null;
    rate: number;
    roundingType: TotalRoundingType | null;
    symbol: string | null;
}
```

Figure 6. Automatically generated model file

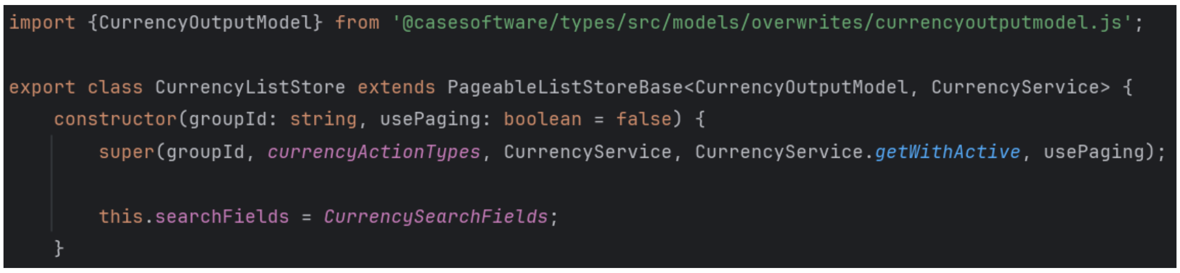


```
import {CurrencyOutputModel as Generated} from '../generated/currencyoutputmodel.js';

export interface CurrencyOutputModel extends Generated {}
```

Figure 7. Automatically generated overwrites file

This overwrite file can then be used within the application to ensure the entire codebase remains type-safe. The Figure 8 illustrates how the generated type specializes the generic PageableListStoreBase class. The type provides accurate code completion and potential bugs are caught by the type checker before they impact the application.



```
import {CurrencyOutputModel} from '@casesoftware/types/src/models/overwrites/currencyoutputmodel.js';

export class CurrencyListStore extends PageableListStoreBase<CurrencyOutputModel, CurrencyService> {
    constructor(groupId: string, usePaging: boolean = false) {
        super(groupId, currencyActionTypes, CurrencyService, CurrencyService.getWithActive, usePaging);

        this.searchFields = CurrencySearchFields;
    }
```

Figure 8. Application file using the overwrites file

The developer's IDE needed to be configured to exclude the “generated” folder from its auto-import functionality, because all imports were intended to be made exclusively from the “overwrites” files.

4.5 Evaluation

The evaluation phase used an explanatory survey to obtain insights into the Developer Experience. This type of survey is designed to explain why a population exhibits certain behaviours or preferences by examining the relationships between different techniques or variables [41]. An explanatory survey was chosen specifically for this thesis to explain why and how developers' workflow and perceptions are impacted by switching from a manual to an automated process.

The questionnaire is composed of five sections. The first section outlines the purpose of the thesis for the participants and is omitted from the Table 2. The second section collects professional background information from the participants. The third section establishes a baseline by exploring the manual workflow and the fourth is dedicated to evaluating the new automated type generation tool. Finally, the fifth section gathers qualitative feedback to understand what could be improved.

In the section 2 participants answer using single-choice questions about their background. The Likert scale was used in sections 3 and 4 to gather feedback on the workflow. The Likert scale is a standard scientific tool that simplifies the answering process for the participants and makes the subsequent evaluation easier [42]. The scale goes from 1 to 5, where 1 stands for “Completely Disagree” and 5 stands for “Completely Agree”. In the last section participants answer with open-ended questions regarding the tool and process around it. Table 2 shows the survey structure in more detail.

The thesis used convenience sampling because the participant pool consisted of only 10 frontend developers. Developers were invited to participate in the survey on the basis that they had previously written API models manually and then had experienced using automatically generated types in their workflow. The survey was shared with developers through Google forms.

Table 2. Survey questions

Question	**Answering method**
Section 2: Background	
1. How many years of experience do you have in professional software development?	Single choice (0-4, 5-9, +10)
2. How would you rate your expertise with TypeScript?	Single choice (Beginner, Intermediate, Advanced, Expert)
3. How frequently have you interacted with OpenAPI specification?	Single choice (Never, Rarely, Sometimes, Frequently)
Section 3: Manual workflow	
4. Manually creating types from the OpenAPI spec was time-consuming process.	Likert scale
5. I found the manual process to be mentally demanding.	Likert scale
6. It was easy to keep the manually-written types synchronized with changes in the API specification.	Likert scale
7. I was satisfied with the manual process of creating types	Likert scale
Section 4: Automated workflow	
8. Integrating the automated type generation tool into my development workflow was straightforward.	Likert scale
9. The tool handles updates to the types from the OpenAPI specification effectively.	Likert scale
10. The TypeScript types generated by the tool were accurate and high-quality.	Likert scale
11. I am satisfied with the experience of using the automated tool.	Likert scale
Section 4: Open-ended questions	

12. What did you like the most about using the automated tool?	Open-ended question
13. What did you like the least about the automated tool? (What could be improved?)	Open-ended question
14. Do you have any other comments or suggestions regarding the tool or the process?	Open-ended question

4.6 Results

The survey was completed by 5 members of the frontend development team. The number of answers is not large enough to support a quantitative analysis, which means that no proper scientific conclusions can be drawn from sections 3 and 4 of the survey. The qualitative portion of the survey yielded several valuable insights regarding future improvements.

The developers who responded to the survey had a variety of experience levels. The respondents also reported a variety of skill levels in TypeScript and their exposure to OpenAPI specification.

Feedback regarding the manual process indicated that participants found it to be time-consuming (average rating: 3.8). On the contrary, the process was not perceived as particularly mentally demanding (average rating: 2.8). The respondents strongly disagreed with the statement that it was easy to keep the types synchronized with the API specification (average rating: 1.8). There was a low overall satisfaction with the manual process (average rating of 1.8).

In contrast to the manual process, the automated tool received highly positive feedback. Respondents gave a perfect score to the ease of integration (average: 5.0) and strongly agreed that the tool effectively handles updates (average: 4.8). The quality of the generated types and the overall satisfaction with the tool both received high ratings (average: 4.6 for both). The full results can be seen in Appendix 1.

5 Discussion

This chapter presents the insights gained during the thesis. First, survey results are examined and conclusions drawn from the feedback. Then, the findings and considerations regarding the artifact are discussed. The chapter concludes with a discussion of future research opportunities.

5.1 Survey

The survey was limited by the data collection timeframe. Due to the thesis's time constraints, the survey was conducted after developers had used the automation tool for only two weeks. To obtain more reliable results, a longer evaluation period of 3-6 months would have been preferable. Another aspect that could have improved the evaluation phase would have been to include objective metrics from the system, as suggested by Noda et al [14].

The qualitative part of the survey indicated that the tool improved the Developer Experience of the case application's frontend developers. This result can be contributed to faster feedback loops and reduction in the cognitive load [14]. The quantitative open-ended questions provided ideas for improving the tool.

One developer wrote about the issue that the use of override files obscures the model's structure within the IDE's type-hinting system. This forces developers to perform extra steps, such as navigating to the generated base file to see a model's properties. A definitive solution to this issue requires further investigation. Two potential approaches could be to either leverage TypeScript features or to develop a custom IDE plugin.

Another developer noticed an issue between the tool and git. After the types are generated, they are temporarily in an unformatted state. This process creates a time window where, from the perspective of the version control system, the generated files are in a "modified" state before the linting and formatting operations have fully completed. The linting and formatting libraries are deliberately excluded from the tool's dependencies in order to leverage the dependencies already present in the root repository.

The legacy backend architecture's inaccurate data models were also seen as a source of frustration. This issue was highlighted during the tool's development by the two different data modelling approaches found in the codebase: a legacy single-model pattern versus the newer input/output pattern. Addressing this issue requires dialogue with backend developers and could lead to substantial refactoring work on the backend.

5.2 Artifact

The development phase had a slow start, despite having well-defined requirements and objectives. The main reason was the learning curve associated with the open-source library's codebase. This challenge, which relates to a learning cost in open-source development was also noted by Stol and Babar [25]. Implementing the primary requirements, such as multi-file output and interface declaration generation, resulted in a better understanding of the library's internal structure. This sped up the pace of later development. The need for specialized expertise can be considered a risk for the future maintenance of the tool, as this expertise is currently concentrated within a single developer.

The objectives for the artifact were defined in the chapter 4.2. The objective to improve DX by eliminating a manual process was achieved successfully. The other objective was to ensure type safety and consistency between frontend and backend. This was partly achieved because it's currently hindered by the inconsistencies in the OpenAPI specification itself. Future work needs to focus on improving the OpenAPI specification so that developers don't have to create workarounds to these inconsistencies.

In the future, it would be worth keeping an eye on a library that meets most of the tool's requirements and adopting it. This would reduce the maintenance burden of the case application's development team. The most important feature that is missing in "@hey-api/openapi-ts" library, is the support for model-per-file generation output. The current preference for generating interface declarations over type aliases could be reconsidered in the future, as the impact of this choice is likely negligible.

It is unlikely that the custom overriding feature would be added to any open-source library, as it was built for a particular use case. If future improvements to the backend codebase lead to a higher-quality OpenAPI specification, it is likely that fewer manual overrides for models would be needed. However, the override files also serve another purpose in scenarios where a frontend feature must be developed before its corresponding backend API is complete. An alternative solution to this overriding feature is worth investigating. If no alternative approach is found, a future role for the custom implementation could be limited to solely generating override files. The need for manual adjustments in the generated models follows a similar problem that Koren and Klemma encountered in the generated user interfaces [6].

Even if in the future the custom plugin is replaced by an official version, the work completed in this thesis remains valuable. The work in the implementation chapter carried out a lot of frontend codebase changes that made it possible to use auto-generated types.

During the thesis, several ideas for improving the backend codebase were also gathered. Implementing these ideas could lead to a higher-quality OpenAPI specification. A post-thesis meeting involving both frontend and backend developers is needed to define a plan for implementing fixes. These improvement ideas should be prioritized according to their estimated implementation difficulty. It is clear that refactoring the legacy single-models to the dual input/output pattern will be a substantial undertaking. Based on a previous short discussion with backend developers, making the sub-models nullable in the specification could be the easiest to implement. These kind of issues caused by inconsistencies in the OpenAPI specification were also noted in a study by Donato and Qin, [7].

5.3 Future work

A more detailed examination of the SDK generation approach is recommended for future research. In addition to providing value to internal developers, this approach can also be valuable to customers, as an official SDK simplifies creating integrations with the product. Companies like Spotify [41] and Monday.com [42] have already been successful in this approach. The case application already contains a solution to API version management, which can be leveraged to accelerate the development of this approach. However, a significant amount of refactoring would still be required in the service layer to make it compatible with the automatically generated network methods.

6 Conclusions

In this thesis, the Design Science Research Method was used to create an automatic type generation tool for generating TypeScript types from OpenAPI specification for improved DX. An evaluation of various methods for obtaining the solution led to the decision to extend an open-source package to meet the case application's requirements. The tool's success in improving the Developer Experience was then evaluated using a survey, which confirmed a positive impact on DX.

The first research question was regarding what options are available for generating types from OpenAPI specification. There were many commercial and open-source options for generating full-fledged SDKs, but none for generating only TypeScript types. This led to the decision to extend an open-source package to meet the case application's requirements. It was noted that for a new application with flexible requirements, the SDK client generation approach could be a more suitable choice.

The second research question was about what kind of factors need to be considered when choosing tool. The target codebase must be analysed to determine the necessary requirements for the tool. A make-or-buy analysis can help determine where an organization should focus its development efforts based on its available resources.

The third research question was regarding whether the tool improved the DX. The survey indicated an improvement in DX which came from faster feedback loops and reduction in the cognitive load. However, the survey's small sample size and limited data collection timeframe challenge its validity.

Future design science research in this area could investigate possibilities of generating SDK clients from OpenAPI specification. There is currently limited research on this topic, which means it could provide valuable information to the scientific community and the software industry.

References

[1] F. Di Lauro, 'GitHub-Sourced Web API Evolution: A Large-Scale OpenAPI Dataset', in *Web Engineering*, vol. 14629, K. Stefanidis, K. Systä, M. Matera, S. Heil, H. Kondylakis, and E. Quintarelli, Eds, in Lecture Notes in Computer Science, vol. 14629. , Cham: Springer Nature Switzerland, 2024, pp. 360–368. doi: 10.1007/978-3-031-62362-2_26.

[2] S. Kushwah, C. Bansal, S. Rathore, V. Kate, D. Bargal, and D. Vishwakarma, 'TypeScript: An Open-Source Programming Language with Options for Robust Development and Large-Scale Applications', in *2024 International Conference on Advances in Computing Research on Science Engineering and Technology (ACROSET)*, Indore, India: IEEE, Sept. 2024, pp. 1–5. doi: 10.1109/ACROSET62108.2024.10743426.

[3] 'Stack Overflow Developer Survey 2020', Stack Overflow. Accessed: Oct. 07, 2025. [Online]. Available: https://survey.stackoverflow.co/2020?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2020

[4] 'Technology | 2025 Stack Overflow Developer Survey'. Accessed: Oct. 07, 2025. [Online]. Available: https://survey.stackoverflow.co/2025/technology#most-popular-technologies-language-prof

[5] M. Heiskanen, 'Migrating a large JavaScript web UI to TypeScript to improve Developer Experience'.

[6] I. Koren and R. Klamma, 'The Exploitation of OpenAPI Documentation for the Generation of Web Frontends', in *Companion of the The Web Conference 2018 on The Web Conference 2018 - WWW '18*, Lyon, France: ACM Press, 2018, pp. 781–787. doi: 10.1145/3184558.3188740.

[7] D. DONATO and H. QIN, 'Using openapi 3 specifications of the 5g core to generate validators in erlang', 2019.

[8] 'OpenAPI Specification - Version 3.1.0 | Swagger'. Accessed: June 12, 2025. [Online]. Available: https://swagger.io/specification/

[9] S. Casas, D. Cruz, G. Vidal, and M. Constanzo, 'Uses and applications of the OpenAPI/Swagger specification: a systematic mapping of the literature', in *2021 40th International Conference of the Chilean Computer Science Society (SCCC)*, La Serena, Chile: IEEE, Nov. 2021, pp. 1–8. doi: 10.1109/SCCC54552.2021.9650408.

[10] H. Ed-douibi, J. L. Canovas Izquierdo, and J. Cabot, 'Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach', in *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, Stockholm: IEEE, Oct. 2018, pp. 181–190. doi: 10.1109/EDOC.2018.00031.

[11] Atlassian, 'What is developer experience?', Atlassian. Accessed: Oct. 20, 2025. [Online]. Available: https://www.atlassian.com/developer-experience

[12] 'What is developer experience? Complete guide to DevEx measurement and improvement (2025)'. Accessed: Oct. 20, 2025. [Online]. Available: https://getdx.com/blog/developer-experience/

[13] F. Fagerholm and J. Munch, 'Developer experience: Concept and definition', in *2012 International Conference on Software and System Process (ICSSP)*, Zurich, Switzerland: IEEE, June 2012, pp. 73–77. doi: 10.1109/ICSSP.2012.6225984.

[14] A. Noda, M.-A. Storey, N. Forsgren, and M. Greiler, 'DevEx: What Actually Drives Productivity: The developer-centric approach to measuring and improving

productivity', *ACM queue*, vol. 21, no. 2. ACM, New York, NY, USA, pp. 35–53, 2023.

[15] M. Greiler, M.-A. Storey, and A. Noda, 'An Actionable Framework for Understanding and Improving Developer Experience', *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 1411–1425, Apr. 2023, doi: 10.1109/TSE.2022.3175660.

[16] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, 'A Design Science Research Methodology for Information Systems Research', *J. Manag. Inf. Syst.*, vol. 24, no. 3, pp. 45–77, Dec. 2007, doi: 10.2753/MIS0742-1222240302.

[17] R. Suter, *RicoSuter/NSwag*. (Oct. 14, 2025). C#. Accessed: Oct. 16, 2025. [Online]. Available: https://github.com/RicoSuter/NSwag

[18] D. Sferruzza, 'Top-down model-driven engineering of web services from extended OpenAPI models', in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Montpellier France: ACM, Sept. 2018, pp. 940–943. doi: 10.1145/3238147.3241536.

[19] M. Borg *et al.*, 'Selecting component sourcing options: A survey of software engineering's broader make-or-buy decisions', *Inf. Softw. Technol.*, vol. 112, pp. 18–34, Aug. 2019, doi: 10.1016/j.infsof.2019.03.015.

[20] S. Aitzaz, G. Samdani, M. Ali, and M. Kamran, 'A Comparative Analysis of In-house and Outsourced Development in Software Industry', *Int. J. Comput. Appl.*, vol. 141, no. 3, pp. 18–22, May 2016, doi: 10.5120/ijca2016909578.

[21] K. Petersen *et al.*, 'Choosing Component Origins for Software Intensive Systems: In-House, COTS, OSS or Outsourcing?—A Case Survey', *IEEE Trans. Softw. Eng.*, vol. 44, no. 3, pp. 237–261, Mar. 2018, doi: 10.1109/TSE.2017.2677909.

[22] L. Loh and N. Venkatraman, 'An Empirical Study of Information Technology Outsourcing: Benefits, Risks, and Performance Implications'.

[23] 'Stainless - Generate best-in-class SDKs'. Accessed: July 12, 2025. [Online]. Available: https://www.stainless.com/

[24] S. Team, 'Generate MCP servers & SDKs from OpenAPI', Speakeasy. Accessed: July 13, 2025. [Online]. Available: https://www.speakeasy.com/

[25] K.-J. Stol and M. A. Babar, 'Challenges in using open source software in product development: a review of the literature'.

[26] H. S. Qiu, Y. L. Li, S. Padala, A. Sarma, and B. Vasilescu, 'The Signals that Potential Contributors Look for When Choosing Open-source Projects', *Proc. ACM Hum.-Comput. Interact.*, vol. 3, no. CSCW, pp. 1–29, Nov. 2019, doi: 10.1145/3359224.

[27] 'npm | Home'. Accessed: Aug. 16, 2025. [Online]. Available: https://www.npmjs.com/

[28] R. Viseur, 'A FLOSS License-selection Methodology for Cloud Computing Projects':, in *Proceedings of the 6th International Conference on Cloud Computing and Services Science*, Rome, Italy: SCITEPRESS - Science and Technology Publications, 2016, pp. 129–136. doi: 10.5220/0005775901290136.

[29] 'orval', npm. Accessed: Aug. 23, 2025. [Online]. Available: https://www.npmjs.com/package/orval

[30] 'openapi-typescript', npm. Accessed: Aug. 23, 2025. [Online]. Available: https://www.npmjs.com/package/openapi-typescript

[31] '@openapitools/openapi-generator-cli', npm. Accessed: Aug. 23, 2025. [Online]. Available: https://www.npmjs.com/package/@openapitools/openapi-generator-cli

[32] ‘@hey-api/openapi-ts’, npm. Accessed: Aug. 23, 2025. [Online]. Available: https://www.npmjs.com/package/@hey-api/openapi-ts

[33] liblab INC, ‘SDK generation: what it is, how it works & best practices | Generate SDKs for your API with liblab’. Accessed: Oct. 16, 2025. [Online]. Available: https://liblab.com/blog/sdk-generation

[34] *unjs/jiti*. (Oct. 04, 2025). TypeScript. UnJS. Accessed: Oct. 06, 2025. [Online]. Available: https://github.com/unjs/jiti

[35] hey-api, ‘Generate types into multiple files instead of a single `types.gen.ts` · Issue #1274 · hey-api/openapi-ts’, GitHub. Accessed: Oct. 06, 2025. [Online]. Available: https://github.com/hey-api/openapi-ts/issues/1274

[36] ‘TypeScript enums: use cases and alternatives’. Accessed: Oct. 06, 2025. [Online]. Available: https://2ality.com/2025/01/typescript-enum-patterns.html

[37] ‘About semantic versioning | npm Docs’. Accessed: Oct. 12, 2025. [Online]. Available: https://docs.npmjs.com/about-semantic-versioning

[38] S. Raemaekers, A. Van Deursen, and J. Visser, ‘Semantic Versioning versus Breaking Changes: A Study of the Maven Repository’, in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, Victoria, BC, Canada: IEEE, Sept. 2014, pp. 215–224. doi: 10.1109/SCAM.2014.30.

[39] *semantic-release/semantic-release*. (Oct. 12, 2025). JavaScript. Semantic Release. Accessed: Oct. 12, 2025. [Online]. Available: https://github.com/semantic-release/semantic-release

[40] angular, ‘angular/contributing-docs/commit-message-guidelines.md at main · angular/angular’, GitHub. Accessed: Oct. 12, 2025. [Online]. Available: https://github.com/angular/angular/blob/main/contributing-docs/commit-message-guidelines.md

[41] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2024. doi: 10.1007/978-3-662-69306-3.

[42] R. Likert, ‘A technique for the measurement of attitudes.’, *Arch. Psychol.*, vol. 22 140, pp. 55–55, 1932.

Appendix 1. The survey result summary

Sections and Questions	**Answering method**	**Results**	
Section 1.			
1. Question	Single choice (0-4, 5-9, +10)	0-4: 2, 5-9: 2, +10: 1	
2. Question	Single choice (Beginner, Intermediate, Advanced, Expert)	Intermediate: 2, Expert: 2, Advanced: 2	
3. Question	Single choice (0-4, 5-9, +10)		
Section 2.		**Average**	**Median**
4. Question	Likert scale	3.8	4.0
5. Question	Likert scale	2.8	3.0
6. Question	Likert scale	1.8	2.0
7. Question	Likert scale	1.8	1.0
Section 3.			
8. Question	Likert scale	5.0	5.0
9. Question	Likert scale	4.8	5.0
10. Question	Likert scale	4.6	5.0
11. Question	Likert scale	4.6	5.0
Section 4.			

12. Question	Open-ended question	“The automated system streamlined the management and updating of models, ensuring they consistently align with the REST API specifications, which enhances efficiency during development. Additionally, when a new model is needed, there's no need to manually search for it under various names or create it if unavailable; the automatically generated models are readily accessible which saves a lot of development time.” “Fire & forget type of deal - however I have not had to deal with models generated by the tool much lately so take it with a grain of salt.” “I didn't have to go to see the specs and manually write the types, create the files, etc. This tools saves me from a lot of annoying manual work. Overriding the types is easy.” "It decreases workload a lot, because manual process was quite ""monkey"" work.”
13. Question	Open-ended question	“We use JetBrains WebStorm as the primary IDE for UI development. Now the hints that the IDE shows by hovering API models are not very useful as they all just show that the model extends the generated model. If I want to see which properties the model has, I need to jump to the overwritten model and then to the generated one. For example, if I hover the ProjectOutputModel I get ""export interface ProjectOutputModel extends ProjectOutputModel {}"" as the hint which does not tell me anything about the structure of ProjectOutputModel. Not sure how this could

		be improved but that's pretty much the only downside I have noticed so far.” “It is bit confusing that it is generating models when starting local development script and shows quite many changes to Git until script finished.”
14. Question	Open-ended question	“The tool seems good and will eliminate manual work needed by the UI developers. I think that in order to have more meaningful comments/suggestions about the process (as in how we use the tool in our development flow), we need to have the tool in use for some time longer.” “Regarding the question ""The Typescript types generated by the tool are accurate and high-quality."" I would like to explain why I gave 4 out of 5. The types generated by the tool are accurate in a sense that they are exactly what are in the OpenApi specs. But, due to historical reasons and backend architecture etc., those types are not always correct. However, that is not a fault of the type generation tool and the tool enables overriding the incorrect types smoothly, so I don't think it even could do any better at the moment.” “It is bit confusing that it is generating models when starting local development script and shows quite many changes to Git until script finished.”