



CREATING REST API'S WITH A CODE GENERATOR

Lappeenranta–Lahti University of Technology LUT

Software engineering and digital transformation, Master's thesis

2023

Valtteri Kokko

Examiners: Associate Professor Antti Knutas

Associate Professor Jouni Ikonen

ABSTRACT

Lappeenranta–Lahti University of Technology LUT

LUT School of Engineering Science

Software Engineering

Valtteri Kokko

Creating REST API's with a code generator

Master's thesis

2023

58 pages, 11 figures

Examiners: Associate Professor Antti Knutas, and Associate Professor Jouni Ikonen

Keywords: Code generation, API, OData, Rest, Model Driven Development, GraphQL, dotNet, Low-code, No-code, Swagger

The goal of this master's thesis work was to figure out what a web application programming interface (API) generator would look like in the target company's working environment and to create a proof-of-concept level solution of a code generator. This is a design science case study into one software company, that already utilizes a code generator in their workflow, and they are interested in adding an API generator component to it. Two artefacts were created for this study, one being a baseline API that uses OData and resembles the desired outcome of the generator and a code generator that creates APIs from JSON parameters, using the baseline version as a template. The study found that generating APIs is a viable way of developing web APIs that suite the target company's customers' needs. The results also show that it brings a lot of benefits, including efficiency gains. This is also supported by literature and previous studies.

TIIVISTELMÄ

Lappeenrannan–Lahden teknillinen yliopisto LUT

LUT Teknis-luonnontieteellinen

Tietotekniikka

Valtteri Kokko

REST rajapintojen luonti koodigeneraattorilla

Tietotekniikan diplomityö

2023

58 sivua, 11 kuvaa

Tarkastaja(t): Apulaisprofessori Antti Knutas ja Tutkijaopettaja Jouni Ikonen

Avainsanat: Koodi generointi, API, OData, Rest, Mallinnuslähtöinen kehitys, GraphQL, dotNet, vähä koodinen, kooditon, Swagger

Tämä diplomityön tavoitteena oli selvittää, miltä web rajapinta (API) koodi generaattori voisi näyttää kohde yrityksen ympäristössä ja kontekstissä, sekä kehittää prototyypin tason API koodigeneraattori. Tämän työn tutkimusmenetelmä on suunnittelutiede ja tapaustutkimus yritykseen, joka käyttää jo koodigeneraattoria päivittäisessä kehityspotkessaan ja ovat myös kiinnostuneita lisäämään API generaattori komponentin siihen. Tätä työtä varten kehitettiin kaksi artifaktia, toinen on generaattorin mallina ja lähtökohtana käytetty perus API, joka käyttää ODataa ja toinen on koodigeneraattori, joka tuottaa API:ä käyttäen JSON muotoisia parametreja. Tutkimus toteaa, että API:en generointi on mahdollista ja kannattavaa kohdeyrityksen asiakkaiden tarpeiden toteuttamiseksi. Tulokset myös osoittavat, että API:en generointi antaa monia hyötyjä, kuten parannuksia tehokkuuteen. Tutkimuksen tuloksia tukee myös aikaisempi kirjallisuus ja tutkimukset.

ACKNOWLEDGEMENTS

I will always be grateful to the important people in my life for supporting me during the making of this thesis and studies in general. It has been a wonderful 7 years in LUT.

Also big thanks to SC Software Oy for providing me the topic for this thesis and the time to work on it. It helped a lot.

Kuopio, 26.4.2023

SYMBOLS AND ABBREVIATIONS

MDD	Model Driven Development
UML	Unified Modelling Language
OData	Open Data Protocol
Rest	Representational state transfer
API	Application programming interface
DotNet	Technology developed by Microsoft
POC	Proof of Concept
CRUD	Create, Read, Update, Delete
DSM	Domain Specific Modelling
MDE	Model Driven Engineering
MDA	Model Driven Architecture
SOAP	A messaging protocol specification
RPC	Remote procedure call
HTTP	Hypertext Transfer Protocol
EDM	Entity Data Model
SOC	System on a Chip
MVC	Model-View-Controller
XML	Extensible Markup Language
NuGet	Microsoft Package Manager
UI	User Interface
URI	Uniform Resource Identifier

Table of contents

Abstract

Acknowledgements

Symbols and abbreviations

1	<i>Introduction</i>	8
1.1	Objectives	10
1.2	Limitations	11
1.3	Structure of the thesis	12
2	<i>Literature and background</i>	13
2.1	API's	13
2.1.1	REST	13
2.1.2	OData	15
2.1.3	GraphQL	16
2.2	Low code and No code	17
2.3	Code generation and domain specific modelling	19
2.3.1	Model driven development	19
2.3.2	Code generation.....	21
2.4	Research Methods	24
3	<i>Design and implementation of the API generator</i>	26
3.1	Designing the baseline API	26
3.1.1	OData as best-practice approach for the API.....	26
3.1.2	API requirements	28
3.2	API Implementation	31
3.3	Designing the generator	35
3.4	Implementation code generator	37
3.5	Evaluation of the implementation	39
4	<i>Findings</i>	46
4.1	RQ1: Is it viable to use a code generator for customer specific API creation?	46
4.2	RQ2: Does code generated APIs bring work efficiency benefits compared to coding them manually?	46
5	<i>Discussion and future research</i>	49

5.1	Discussion.....	49
5.2	Future research.....	52
6	Conclusion.....	53
	References.....	54

1 Introduction

Code generation is a method of software development, that has been used in one way or another as long as there have been higher level programming languages. In modern software development the usage of code generators is increasing but their popularity has not increased the way that was initially expected. The generators today are utilized from maintainability, extensibility, and reusability. (Sebastián et al., 2020) To enable any form of code generation in a way that a company can use it to generate software, model driven development (MDD) must be established as a development method. In a nutshell MDD is describing a system or a software using models and modelling, with for example Unified Modelling Language (UML). (Kelly and Tolvanen, 2008)

This thesis work sets out to research and solve how Application Programming Interfaces could be generated with a code generation in a company environment where code generation is already in use. This study is an action research case study where we try to find if it is viable to generate APIs and if they bring any efficiency benefits for the company compared to coding the APIs manually. This study will also produce a proof-of-concept level API code generator.

The idea for this thesis came from the target company of this work in summer of 2022 when SC Software Oy was recruiting new employees and showcased the way they approach software development for customized solutions. Their way of doing things was very different compared to many other companies and their competitors. They used a code generator in their day-to-day work. It has been a crucial part of their way of operating from design of the software to the implementation. The generator they use is quite comprehensive as is but there are parts that have not been properly implemented or even researched yet. There was a lot of discussion on what part of the generator should be upgraded or what could be added. Things like updating the authentication in the final generated product, migrating the generated code to newer version of dotNet and the final topic of this thesis, the generation

of Rest APIs using a form of best practices implementation using Open Data Protocol (OData).

The target company of this thesis is SC Software Oy. It is a software company founded in 2014 and has multiple offices around Finland. They develop customized software solutions and industry specific software products. The main selling point of their services is their high-productivity software development platform that allows them to deliver good quality products in very little time compared to traditional software development. The technology they use is close to low-code development, but it is not trying to be completely codeless solution. The development platform allows the developers to focus more on the implementation of business logic and customer requirements rather than medial repetitive tasks. This also means that the scope of the solution is not limited. (SC Software Oy, 2023)

The SC software code generator is a development platform that has been in development for multiple years now. It is constantly evolving and adapting to SC Software's and its customers' needs. The generator has been a key part of SC Software's success in their respective fields. It has allowed them to produce functional applications for their customers as early as they have defined what kind of system they want. Behind the generator is a model-based definition of the application that the generator uses to create the code. This includes the entire pipeline from database to the user interface.

The need for external application programming interfaces (APIs) for customers has been a feature that has been long asked for and was a clear choice for the topic of this thesis. Data is one of the most important things that is traded today, and many companies and products need to exchange data between each other. One solution to this is Rest APIs that provide easy access to needed data without exposing unnecessary or unwanted data in the process. For this work it was important to include a sort of best practices approach to APIs in the form of the OData standard. It specifies a lot of different practices to serving and consuming data through Rest APIs. Because this has been a frequently asked feature by customers and potential customers, some APIs have been already implemented. The problem with this is that with every customer the API looks about the same but needs to be written from scratch.

To tackle this problem the code generator needs a new feature that creates these APIs with given parameters. This is not as straight forward as one might think though, for the generated API needs to be made so that it can be modified by developers to meet the customers more specific needs and it needs to be regeneratable without losing the manually written code. The reason for this is that the generator also needs to be maintained and further developed to keep up with the rapidly evolving world of software. When the generator is updated, it can be used to update the previously generated API to match the new features or fixes and without losing the manually written code.

Code generation is a valuable tool to have in a company's toolbox as it brings a lot of different advantages and benefits to the development process. By generating code, quality is improved, efficiency of the developers is better and prototyping and demonstrating software is a lot easier. To make code generation work, MDD and Domain Specific Modelling (DSM) are the key. MDD and DSM are tools to model software or environments. By being able to model the software, we are easily able to produce or generate code based on that model. This model could be, for example made in UML or some other visual tool. Code generators are also not compatible with all projects, because the generator needs to be well defined to produce quality code that fits the desired purpose. If the generator does not have a model or the target is way too large and ill defined, the generator will not have any of the benefits that were just told. (Kelly and Tolvanen, 2008)

1.1 Objectives

The primary objective is to study Rest API generation with a code generator. The target company wants to know if it is viable at all to create APIs to suite their customers' needs or is it better to code them separately for all their customers. In addition to viability, they want to know what the possible outcomes from the generator might be like. What are the possible problems in API generation, how would they be overcome, what best-practice would be suitable for a project like this? Also, a big part of the current generator is the ability to regenerate the code but also allow manual customizations and additions to it. The generated

API code must also implement this feature and it is important to see how that would be possible.

Secondary goal is the creation of a proof-of-concept (POC) level code generator, that can generate Rest APIs. The goal here is a console application that takes in parameters and generates a mostly functional Rest API. The POC will be able to do most basic operations on a given resource the API uses. This means that the API has end points to get, create, update, and delete resources. In addition to these basic functions there will be an option to create custom operations and functions. These will be a key part in enabling the manual coding of the application.

This thesis study is qualitative research into code generation and Rest APIs. To be more specific, this is a design science case study into the way SC Software uses code generation in their workflow and how an API generator would benefit the already existing generator and working methods. The research questions for this study are as follows:

***RQ1:** Is it viable to use a code generator for customer specific API creation?*

***RQ2:** Does code generated APIs bring work efficiency benefits compared to coding them manually?*

The first question aims to answer the target companies need of wanting to map the possibilities in API generation and is the main focus of this thesis. The second question aims to enlighten the possible efficiency benefits and problems of API code generation when it is meant to be modified by developers after the initial generation of the code.

1.2 Limitations

The generator created for this thesis will be POC level application. It is not meant to generate production ready APIs and it will not be integrated into the existing generator as such. The generated API will also be a basic Rest API, only allowing basic CRUD operations and

simple custom features. Although the API is meant to follow the OData best practices, it will not include things like authentication, or any database connection as they are modules that can be changed and handled as separate parts from the API. Data will be served from a static resource and the API can only be used locally. The generator will also not use any existing resources of SC Software, because it was not seen as relevant for this work as the endpoints and resources the API uses must be defined separately of existing model and resources anyway. The data used by the API will also be simple and only contain simple or no links to other resources. There is no need to have any more complex data for this thesis work.

Other big limitation for this work was time. The target company gave 7 months to get this thesis completed. After the 7 months there is little to no time to commit to the coding of the generator. This thesis will also mostly focus on the generator SC Software uses and will not go deep into replacements for it. Other code generator platforms can be introduced in the thesis, but they will not be compared to the SC Software generator platform. Principles and theory that emerges from other resources and references will be reflected onto this case.

1.3 Structure of the thesis

After this chapter we go through literature including relevant topics, previous studies conducted relating to this study and the research method of this thesis. After literature we describe the process of creating the artifacts and use them to get data for our study. In the fourth chapter are the findings of this study and answers for the research questions. And lastly in the fifth chapter the discussion related to the findings of the study and possible future research that could be conducted in the future based on this thesis.

2 Literature and background

This thesis work is a combination of multiple different independent modules working together and there are a lot of different topics. This chapter focuses on the background of these different topics, including APIs, Low code – no code solutions, code generation and other relevant topics that enable the efficient use of the components of this study. This chapter also investigates previous studies relating to this thesis.

2.1 API's

APIs allow different software or software component to discuss with each other. Usually, they work with two components working independent from one another. These components are client and server. The client sends the server requests into the API and the server sends data back to the client in a response. There are multiple ways that an API can work. The four most common are SOAP, RPC, Websocket, and REST. Simple object access protocol (SOAP) is a protocol that was used a lot in the past. It uses XML as means to exchange data. Remote procedure calls (RPC) are a technique where server runs a procedure and delivers the output of that to the client. Websockets are modern JSON based API technology, that allows two-way communication with a client. The server can also return callbacks to its clients. The fourth way is the most popular at this moment, Representational state transfer (REST). In REST the client sends requests as data and the server uses this data as an input to do the internal operations to return data back. REST uses HTTP as to communicate between the server and client. (Amazon, 2023)

2.1.1 REST

REST is a resource oriented, meaning that all resources can be accessed by a request path. In REST the client sends a HTTP request to a server. This request consists of headers and data. The data can be sent in multiple different formats, such as JSON, HTML, plain text etc. JSON is the most popular, as it is easy to parse and understand by humans and machines.

The other part of the request, headers are an important part of the communication as it can contain cookies, authorization information, caching, and other important identifier information.

REST as itself is not a protocol or standard but a set of constraints that can be implemented in multiple different ways and for an API to be considered RESTful, it has to implement the following criteria:

- A client-server architecture that consists of clients, servers and resources that communicate with HTTP.
- Stateless communication that does not store info between requests and handles requests separate and unconnected.
- Cacheable data.
- Uniform interface between components that ensures that the transferred data is in standard form.
- A layered system of servers that are organized by type of server involved in getting a resource, that is invisible to the client.

As REST is only a set of guidelines, the developers implementing a REST API can only choose what parts of the definition they want to use. This allows the API to be as light or as heavy as needed. (Red Hat, 2020)

In a typical RESTful service, there are four universal verbs that can be used: GET, PUT, POST and DELETE. These verbs are the most important keywords when interacting with the services resources as they retrieve, update, insert and delete resources. (Halili and Ramadani, 2018)

REST is a lightweight solution for API implementations and the most popular way to use them. As big tech companies like Google, Facebook and Twitter using them in their cloud-based services, REST has the advantage of being easy to learn, implement and use as the big

companies need and produce material that others can use as well. RESTful architecture also easily scalable as well. The disadvantages of REST are that it is not very comprehensive, as it does not cover web service standards, like security and transactions and it is heavily dependent on request headers, like for storing authentication details. The use of headers is sometimes seen as clumsy and requires the API to use HTTP for communication. (Halili and Ramadani, 2018)

2.1.2 OData

For the API, that the code generator creates for this thesis OData was chosen as an easy way to create best-practice APIs. This allows us to focus more on what can be provided with the API rather than the technical implementation of an API as the focus of this work is more on the code generation rather than the API itself.

OData is a comprehensive set of best practices, that help create and consume REST APIs. It is also standardised and covered by ISO/IEC and OASIS standard. OData is a simple way to abstract the usage of an API and the conventions it describes makes it easy to create APIs and focus more on the applications business logic or other important parts. For example, OData covers request and response headers, status codes, HTTP methods, URL conventions, media types, payload formats and query options. (OData, 2022)

One of OData's main selling points is the query language it introduces into an API that uses OData. It has a lot of different filtering options, like filter by keyword, order by, count, etc. For example, an OData HTTP GET request might look like this:

```
https://services.odata.org/v4/TripPinServiceRW/People?$top=2 & $select=FirstName, LastName & $filter=Trips/any(d:d/Budget gt 3000)
```

This request returns the first 2 persons resources who have registered in at least one trip that costs more than 3000, and only display their first name and last name (OData, 2022). Behind

this query on the server, there is an Entity Data model (EDM), that allows these kinds of requests to be made and understood.

A study from 2018 it was said that while OData services are easy to consume, they still require a lot of understanding and tedious manual work to create. They also created a code generation tool for creating OData APIs. (Ed-douibi et al., 2018) The topic of their work suggests that this is a not a problem only in this thesis work but also recognized elsewhere. The study also gives ground to the viability about usage of code generation tools regarding APIs as Ed-douibi, et al. managed to create APIs supporting OData this way.

2.1.3 GraphQL

A very similar alternative to OData is GraphQL that was first developed by Facebook. It is a query language for APIs and server-side runtime for executing queries using a typed system custom defined to suited data. GraphQL is seen as an alternative to RESTful APIs and is gaining popularity as big tech companies like Microsoft, Twitter, Amazon and others are adopting it to their systems and applications. (GraphQL, 2022)

The main difference between GraphQL and REST APIs is in where REST gives a list of endpoints that a client can call, GraphQL export a database schema that the clients can make queries to. This schema is a multigraph that contains objects. Those objects can have references to other objects. GraphQL also allows the client to decide what data they want from the API. In traditional REST APIs one endpoint returns a fixed JSON document that contains all the fields of an given object. Usually the client does not need all that information. For example in GraphQL, if the server has Customer objects that contain a large amount of fields, but we only need the name of the customer we can formulate a GraphQL query that returns only the name and nothing else. This way a lot of client and server resources can be saved. (Brito et al., 2019)

In the context of code generation, GraphQL has not been researched very thoroughly as of yet but a study from 2018 researched if it is feasible to generate GraphQL wrappers for REST-like services. The reasoning behind this was that current REST APIs don't utilize the multiple new features and advantages of GraphQL and that if anyone wanted to use GraphQL they would have to make wrappers that translate the queries for the target service. They created a prototype generator that outperforms other similar existing solutions and is able to create GraphQL wrappers for 89.5% of APIs, with certain limitations. (Wittern et al., 2018)

2.2 Low code and No code

Low code and no code solution platforms are a rising method to develop new software solutions and applications. They allow the development of software by minimizing the need to code anything by hand. Low code platforms allow fast continuous and test-and-learn way to software development. (Sanchis et al., 2019) The main difference between low code and no code is straightforward: no code solutions require no manual coding and low code requires some coding (Hurlburt, 2021). As the SC Software code generator is more similar with a low code solution than no code solution, this section of literature review will focus more on that but will briefly cover no code solutions as well.

Low code can be seen as a predecessor or even same as model driven development (MDD) or model driven engineering (MDE), as stated by Cabot (2020). He also states that low code has a lot of potential from previous and current users of modelling tools but also from new interested parties, that have never been a part of MDE or MDD. This means that low code also brings MDD and MDE to new domains and communities, which drives it even further. MDD, MDE will be covered further in chapter 2.3.

Low code solutions are easy to use and fast to make use of as they are pre-configured and built to be used as such. They utilize visual user interfaces to be more user friendly to those who are not qualified to work with just code or other technological tools, like databases. Even with very little technical knowledge, most people could create business applications

with relative ease. This allows the companies to develop applications with fewer resources and this way achieve rapidity and agility in their development. (Sanchis et al., 2019) A good example of rapidity and agility was research done by Laura Fadjukoff, who discovered that with the SC Softwares code generator the same application could be made 86% faster than coding the application manually from scratch. This is a good example of the power of low code solutions when utilized in the correct domain. (Fadjukoff, 2021)

Low code platforms have a lot of advantages and a few limitations as discovered by Sanchis, et al. (2019) They list privacy of development, rapidity, cost reduction, complexity reduction, ease of maintenance, involvement of business profiles and minimization of unstable or inconsistent requirements as the main advantages of low code solutions. The main limitations on the other hand are scalability, meaning that low code solutions are mainly for smaller applications; fragmentation of the company and the different vendors and their specific programming models; and software-only systems having expectations from the experts using the platform wanting to use their knowledge with the right level of abstraction. (Sanchis et al., 2019)

In addition to the previously mentioned advantages of low code solutions, low code and no code solutions bring additional benefits to small and medium sized companies as found by Bhattacharyya & Kumar (2021). They found that with cost effective and easily maintained low code application these companies can compete and gain advantage over larger corporations. (Bhattacharyya and Kumar, 2021)

A study from 2021 into sustainable public sector software also mentioned that low-code solutions could make the software business more sustainable. They say that the public sector is a large consumer of software and most of the software is tailored to their needs with no thoughts given to reuse and sustainability of them in the long run. A tool like low code could make the public sector software development more sustainable. (Setälä et al., 2021)

2.3 Code generation and domain specific modelling

The main goal of this thesis is to create an API code generator that will be like the existing generator SC software uses in their workflows. That generator uses a visual model that is converted into XML and further into working code. This part of the literature review will investigate what is code generation, the existing studies regarding that and how domain specific modelling is relevant in this thesis and code generation in this context.

2.3.1 Model driven development

MDD and domain specific modelling (DSM) are at the center of development in SC software. They are used as a baseline to create application code with their code generator. DSM means raising the abstraction level of the current programming languages by specifying the solution using the problem domain concepts and the final applications are generated from these high-level specifications (Kelly and Tolvanen, 2008). DSM is the result of seeking new ways to improve productivity in software development by adding abstraction. These days the additions to programming languages do not bring the kind of efficiency gains as they used to years ago with the move from assemblers to third generation languages. Back then the improvements were on the level of effectively writing the same functionality with one line instead of several before. (Kelly and Tolvanen, 2008)

The code generator of SC Software is a good example of DSM that Kelly & Tolvanen (2008) have been researching. They state that DSM is domain specific, and this means that the modelling language and the generator need to be restricted to certain kinds of applications to achieve the best results. SC Software's generator is a prime example of this.

DSM also provides a lot of benefits to software development. It significantly increases productivity, especially in product families and using a specific tools, such as MetaEdit+ that enables easy tools for DSM. By using DSM and appropriate tools companies can reduce

the needed work by a significant amount. One research found that the productivity gains can be as much as 5-10-fold and a similar decrease in time needed for new developers to become productive. (Tolvanen and Rossi, 2003) The benefits of DMS have also been described by Kelly & Tolvanen (2008). They state that DSM brings several benefits over general purpose and manual approaches. These benefits are increased productivity, improved quality, and shared expert knowledge. They also emphasize the benefits in productivity, and the commercial benefits that DSM brings along, like fast technological development and short product lifespan in terms of reduced time to market. With the automation DSM enables, companies can achieve better quality. This comes from less generated testing, because errors can be noticed a lot earlier in the development process. The importance of code quality also comes into play when the software enters the maintenance stage of its lifetime, as nearly 80% of the costs in the project come from maintenance (Telea and Voinea, 2011). The better the quality of the code, the easier it is to maintain and keep the costs lower.

Quality is further improved by the benefits that DSM brings to the workforce of the company. Experienced developers can focus on defining the modelling language and mappings to code and the solution developers can use that modelling language to define the models for the project they are working on. This kind of system can enable the company to train new developers a lot faster, bringing cost benefits as well. By doing development and orientation this way the company does not need to outsource work as it can keep core competency in house a lot easier. On the flipside of this, the company can also outsource work a lot easier as the other party only needs to understand the domain and the language of the project. (Kelly and Tolvanen, 2008) The benefits of DSM are also confirmed in practical study, where they utilized DSM in a commercial setting and found that developer productivity increased by at least 750% with the additional benefits of improved code quality and development process. (Kärnä et al., 2009)

Kelly and Tolvanen have also researched some of the downsides and challenges when it comes to DMS in their study from 2016. They state there that DSM can increase productivity by 500-1000% and that creating a DMS language is costly and difficult. They conclude that traditional UML-based MDD or MDA offers only up to 40% increase in productivity. The

upwards of 500% increases in productivity have been seen in MetaEdit+ tool but they cannot see a reason why these big increases could not be seen elsewhere. The main problem in this is that DSM language creation in other tools is difficult. They also state that in MetaEdit+ an industrial-scale DSM could be created in 5-15 days, whereas in other tools the same result could take 10-50 times longer. (Tolvanen and Kelly, 2016)

2.3.2 Code generation

Code generation at the core is just automatically creating code from some model or parameters. In today's fast-paced world of software development, any improvements to productivity are seen as a worthwhile investment. Model-based code generation is one of the main topics of this thesis and it is important to cover the basic benefits and downsides it brings. Model-based code generation means generating working application source code based on a visual or non-visual model where that model represents the systems behavior or architecture (Bell, 1998).

Using code generators contributes to the quality and efficiency gains that DSM brings and as Kelly & Tolvanen say in their book "A code generator that is defined by an expert will, no doubt, produce applications faster and with better quality than those created manually by average developers." (Kelly and Tolvanen, 2008). In addition to Kelly & Tolvanen's book, Rumpe (2017) mentions similar benefits of code generation:

1. Increased efficiency for developers,
2. Simplified maintenance, as application modelling is separate from technical code,
3. Rapid prototyping,
4. Fast feedback with demos and test runs,
5. Generation of automatic tests bring improved quality (Rumpe, 2017)

The advantages listed by Rumpe are in line with Kelly & Tolvanen, as well as the experience in SC Software.

At the base level, code generation is the result of DSM. Code generation just uses the theory of DSM to produce real usable software by transforming the model created by DSM. By being a product of DSM, code generation also has the same shortcomings. Rumpe also raises a new challenge when using a code generator. The maintainability of the generated system is dependent on the availability of the code generator. When the generator is no longer available for maintenance, the only way to maintain code is to manually alter the generated code. In addition to this, if the system requires migration to a new platform or other critical changes need implementation, the development team of the generated product needs to be prepared to do it manually. In a case where the generator is not usable or developed in an independent project, after the initial generation of the code the generator should be written in a way that allows it to be reconstructed for maintenance or evaluation purposes. This might be difficult, as modern languages and programs are getting more and more complex. (Rumpe, 2017)

In this thesis work, an API code generator will be created. The process of making a code generator is simple and described by Kelly & Tolvanen in their book “Domain-specific modeling: enabling full code generation” (2008). The process goes like this:

1. Desired output code is pasted into the generator as the content,
2. Each repeated section in the output is reduced into single occurrence in the generator loop that goes through each model structure,
3. In sections that have more than one alternative forms, each section is surrounded by code that chooses the right outcome based on the model,
4. All places in the output that contain matching property values are replaced based on the values in the model.

By the end of this process, we have a complete, executable program that does not need any manual alterations or additions to its source code. The end result is usable because it has been tightly defined to one purpose inside of one company. (Kelly and Tolvanen, 2008) This approach to creating a code generator will be utilized later in this work.

Previous studies have been done regarding code generation. As of now, the popularity of low code and no code solutions are rising, but code generators are not used that much, even today. One study investigated several popular code generation tools from big tech companies and found the reason the generation tools in them are not used that much. The reason for this is that the tools try to implement multiple different code generation ideologies at the same time without focusing on any one of them specifically. The developers of these tools also do not try to bring anything new to the generation and this has resulted in the tools not evolving any further. (Bajovs et al., 2013) This study shows that the existing tools are quite useful, but they remain unused and that they need more specific and innovative improvements. To be used effectively, they need to be defined better to suit one purpose as stated before in this chapter.

In the case of this study, model-based code generation is more of the kind that we want to look into. There is a lot of research regarding using UML as the modelling language for code generation. UML is quite close to the modelling language used in SC Software, so the findings of these studies relate to this work and the existing generator closely. UML is one possibility to enable MDD and DSM. UML can be used in multiple different kind of applications when it comes to code generation, for example one study used UML and generated code out of it for developing a System on a chip (SOC) design from UML to SystemC code (Boutekkouk, 2010). Web applications have also already been generated from UML models, like in a study from 2020, where the researchers make enterprise web applications from UML using the model-view-controller (MVC) architecture. They found that after several years of implementing their solution to model-based code generation, the way they were doing it was not what was previously reported in earlier studies, as they were successful in making a small Italian software company more competitive and successfully implementing their solution. The earlier studies were reported to be far from successful. (Paolone et al., 2020) A study from 2005 also showed positive results regarding using UML to code transformation. They successfully created a POC through pragmatic application of UML but say that the result the benefits and possible problems only show in a more complex system. (Vogel-Heuser et al., 2005) Also one study researched the realistic costs and benefits of UML in software maintenance. They studied two teams, one of which was using a UML

documentation during the maintenance and evolution of a nontrivial system, and one was not. They found that the UML team had significantly better functional correction of changes and better design quality. On the downside, UML increases the development time slightly caused by the overhead of updating the UML documentation. (Dzidek et al., 2008)

2.4 Research Methods

This work is a master's thesis, that uses Design science as the main research method. Design science is the scientific study of an artifact that is created for the study with the main goal of solving a practical problem of some interest. The artifact that is created in a design science study can be described by specifying the structure, behavior, function, environment, or effects of the artifact. This study has a practical problem, that the artifact is set to solve meaning that the stakeholders have a situation that has a current and a desired state for a problem. The outcome of design science is not only affected by the artifact but by contextual knowledge about the artifact. (Johannesson and Perjons, 2021)

This work is also a case study into SC Software. Johannesson & Perjons (2021) define case study as follows: "A case study focuses on one instance of the phenomenon to be investigated and it offers a rich in-depth description and insight into that instance". A case study is differentiated from a survey or a laboratory experiment by setting the focus on depth and context. To be more exact, this study is an action research, which is a method for case studies. Action research is a strategy that is used to address practical problems in a real-world setting. In action research the researcher generates new scientific knowledge and solves an important problem that is experienced. The action research characteristics are as follows:

- Focus on practice: the research is conducted in the real world.
- Change in practice: researcher takes action and changes the practice.
- Active practitioner participation: the practitioner is not passive subject.

- Cyclical process: The process is going through the steps of Diagnosis, Planning, Intervention, Evaluation and Reflection.
- Action outcomes and research outcomes: the research produces results that contribute to academic knowledge and are valuable to the stakeholders. (Johannesson and Perjons, 2021)

For this study, the real-world problem is the one set by SC Software. They want to know how they could implement an API generator for their existing code generator. The artifacts that are created are the baseline API and the code generator that uses said API to generate new APIs. Data is collected throughout the study by researcher observation, from developing the API and generator to using the generator and the API it generates. Efficiency is measured by making the same API by hand and with the code generator. The baseline API will be used as the comparison. The evaluation of the artifacts is done by comparing the created artifacts to their requirements and how well it fits the goals of this study.

The process for this study was the cyclical process of Diagnosis, Planning, Intervention, Evaluation and Reflection. These steps can be found in this thesis as follows: Diagnosis in chapter 2, planning for API and generator in chapters 3.1 and 3.3, intervention in 3.2 for the API and 3.4 for the generator, evaluation of the artefacts in 3.5 and 4, and reflection in chapter 5. For this study only one research cycle was needed, but another could be considered as there are still some variables. Some of these variables are covered in chapter 5.

3 Design and implementation of the API generator

For this thesis, two artifacts needed to be made: a baseline API and the generator that produces APIs. The generator uses the baseline API as a template and model of the resulted generated API. This chapter will go through the design and implementation phase of the artifacts.

3.1 Designing the baseline API

SC Software was keen to know what kind of APIs they could generate or utilize with their customers. For this study an open web API that can do simple read and write operations was selected. The target company also wanted to utilize some form of best-practice way to design the API. For this purpose, OData was selected.

3.1.1 OData as best-practice approach for the API

For the best-practice approach to APIs, there was two similar options: OData and GraphQL. On the surface both do the same thing: provide a query language and other useful features for a web API. In a nutshell, OData is a standard that defines best practices to build and consume REST APIs and GraphQL is like a programming language that is used to fulfill queries with complete understanding of data in an API.

As of now, GraphQL is quite a novel approach to developing APIs. It is not as popular as REST APIs at the moment, but it is rising rapidly in popularity because it is used by a lot of big tech companies, like Google, Microsoft and Amazon. But now it is still not suitable for everything for a few reasons: a high learning curve, possibility for queries getting too complex, difficulties in implementing a cache system and no support for file uploads. The advantages of GraphQL on the other hand are fast queries, as only specified fields are

fetches, good compatibility with complex systems, only one call needed to fetch nested data, and detailed error messages. These advantages make it great for applications that have bandwidth concerns, applications that benefit or require low latency and APIs that need to be flexible for several use cases and integrations. (Samaranayake, 2022)

OData is a standard developed on top of REST. It is an older and widely more used now than GraphQL and as it is RESTful, it is simpler to use for the customers of SC Software. OData is most useful when the API consumers are diverse and using multiple different tools, or the consumers are unknown. OData helps manage this by reducing the time needed to learn and code against different APIs. As OData is easy to implement for its RESTfulness, there is minimal effort needed to grant access to basic Create, Read, Update and Delete (CRUD) operations. OData further increases productivity by removing the need to worry about configuring response headers, status codes and HTTP methods, which leaves more time to focus on developing the business logic for the application. In addition, OData is very performant, making it competitive with GraphQL. (Samaranayake, 2022)

Advantages of OData are support for HTTP and JSON, data retrieval based on URL, support for all forms of data, including custom classes as data sources, no need for proxy service objects, full support for CRUD and good performance due to OData being lightweight. On the other hand, its disadvantages are fine grained access control that must be enforced on multiple levels, no sufficient information to consumers on how to use the services and the users of the API have to be familiar with the schema. (Samaranayake, 2022)

The decision of the best-practices guidelines for the API in this thesis is an important one, as it dictates most of the functionalities of the generated API. For this work OData was the way to go, as it provides a lot of the things needed ready and easily accessible. These features include full support for CRUD, performance, ease of implementation and configuration and it being a solid choice to the customers of SC Software because it utilizes REST and as mentioned before in the literature review the REST is a lightweight and easy to learn and use. (Halili and Ramadani, 2018) SC Software already uses REST APIs in some of its applications, so OData is easily integrated into their current technologies. Another reason is

that previous studies have already looked in to OData in similar settings, as mentioned before (Ed-douibi et al., 2018).

The code generator point of view is also something that must be considered in this as it brings its own difficulties. By using OData, we can focus more on how the code generator works rather than the API. OData provides everything we need for this study without being too complicated to implement. GraphQL could bring some extra headaches for its higher learning curve for a developer that has only used REST APIs before.

Right now, OData is the better option, but this might not be the case as we go forwards. GraphQL is rising in popularity and support and very well might be the better one in the long run. The API and code generator created for this study are going to be on POC level, so the easier solution was chosen. OData is a very good choice in the future as well, because it is an industry standard, and it is not going to be retired anytime soon because of the popularity of REST APIs. If OData is chosen for the final version of API generation, it is also not going to be a problem to use GraphQL as well because by using a REST based solution, a GraphQL layer could be added on top of that in the future (GraphQL, 2022). Another option is to migrate from REST to GraphQL, but there are some challenges to overcome in that as discovered in a study where they migrated from REST to GraphQL in an distributed systems environment (Vogel et al., 2018).

3.1.2 API requirements

As the API will be the result of the generation, the API itself is not going to be that complicated. The requirements for the API are as follows:

1. *Needs to implement basic operations, meaning read, write, update, and delete on any given resource,*
2. *API needs to be documented,*
3. *API needs to implement OData,*

4. *Support for custom endpoints other than basic CRUD,*
5. *The API must be expandable with new manually written code,*
6. *The code structure of the API needs to be easily transferrable to a template for the code generator.*

And as for limitations that this API does not need to implement, as it is not relevant for this study:

1. *API does not need to implement database access.*
2. *API does not need to have authentication.*
3. *The API will be a stand-alone project from any other SC Software resources.*

The first requirement comes from the basic needs for the customers of SC Software. They want to offer their customers the possibility to offer 3rd party clients access to the data that the customer is using. SC Software sees this as a great business opportunity as the feature has been requested before. The basic operations are a great way to start figuring out if generating the APIs is the way to implement them into the existing workflows in the company.

The second requirement for the API is also an important one. SC software wants that the documentation also is bundled into the code generation. They have specifically asked if a documentation library like Swagger could be implemented into this POC. Swagger is an open-source tool to automatically generate API documentation from the API definition. It generates a simple UI to visualize an API showing the structure of the resources, what HTTP method the endpoints use and describe the responses that the API provides. This UI also includes a way to test single endpoints in the API. (Swagger, 2023)

As previously stated in this thesis, the API will utilize OData as a best practices approach to APIs. The requirements include implementing the various query options, such as filter, count, select, orderBy etc. to be compatible with the OData standard. This allows easier

development, as most popular programming languages and frameworks, like C# and dotNet in this case, have support for OData. An important feature for the API is support for custom endpoints. In this case custom endpoint means any other endpoint other than the basic create, read, update, and delete endpoints. As OData makes it possible to execute custom functionalities of the API, these are a good feature to have in the final version of the API. For example, if the customer wants to allow any more complicated business logic to be executed with this API, these custom endpoints are the way to do that and because the basic CRUD functions are rarely the only things wanted from an API, these allow better customization for the specific needs of the customer.

To make the API as close to the current solutions of SC Software, the API needs to be easily expanded and overridden. What is meant by this is that the API code provides some simple implementation for the basic CRUD functions and the possible custom functions and then offer the possibility to override the initial implementation with any fitting custom implementation that a developer might choose. This is a key element of the final generated API and needs to be implemented in the baseline version as well.

The developed API is going to be used as a baseline template for the code generator. What this means is that the code generator will be generating APIs that resemble the baseline version. This brings its own requirements for this API, as the generator is going to use the source code and files of this API as a template. The requirement means that the code must be structured and written in a way that allows this to be as easy as possible. The baseline API will be converted into a template after the design is ready. This requirement is a challenging one, as the API will be shaped partly during the development of the generator because it might bring new insight into the APIs structure. Also, if there are any additional features, they will be first added to the baseline and then converted into the template. These new features affecting the APIs source code can be changes to the API or the generator.

As for the limitations and restrictions for this baseline API, there are a few. The API will not have any form of database access, as it is not relevant for this thesis. SC Software already has appropriate architecture for this kind of an API and so this work does not need to

implement it. The authentication side of things is also something that is not relevant for this work much for the same reasons as the database access. SC Software has working solutions for this as well and it is also just an extra feature for the API so it will not be implemented in this version of the API and generator. The last limitation for the API is that it does not need any existing resources SC Software has. As these features are easy to implement in the future and are not really needed for this thesis. However, the models and concepts used by the API will be similar to the ones used by SC Software.

3.2 API Implementation

In this chapter we will go through the implementation of the baseline API for the code generator. We will look at how the API is constructed and how the different problems defined in the previous chapter have been solved.

The implementation of the baseline API is not that complex. All the requirements set out in the earlier chapter are met as they all are relatively easy to implement. As SC Software currently defines their applications in a specific way, the API has been designed with a similar ideology. The API application has models, controllers, services, repositories, and of course other relevant infrastructure code that the application needs. The baseline API has been developed using Microsoft's dotNet 6, as SC Software uses multiple dotNet versions in their development and this version is the latest one at the time of this thesis work.

The most difficult requirements to meet are the implementation of OData and the support for custom endpoints, the expandability of the API code and the suitability to being used as a generator template. The implementation of OData was probably the most difficult out of these because of the lack of experience with web APIs by the author. DotNet 6 has a readymade OData NuGet package available, and that package has been used in the API implementation. The implementation of OData consists of Entity Data Models (EDM) and API Controllers. OData uses data models as its basis and an OData service uses an abstract data model called EDM to describe the exposed data in the service. This way an OData client

can make a GET request to the service and receive an XML representation of the data model in the API. (Microsoft, 2022) The controller is a class that implements the defined operations and acts as a bridge between the API and the OData EDMs. To make this work, we need an EDM model builder that is shown in Figure 1.

```

1 reference
public static IEdmModel GetEdmModel()
{
    ODataConventionModelBuilder builder = new();
    builder.EntitySet<UserDto>("User");

    builder.EntityType<UserDto>().Collection.Function("CustomOperation").Returns<UserDto>();
    return builder.GetEdmModel();
}

```

Figure 1 EDM model builder

In the figure we can see that a “User” entity model is created, “UserDTO” being a model and “User” being the controller the model is linked to. This binds the User controller to the OData EDM of a “User”. To further enable the features of OData, like filtering, selecting, and counting data, we need to allow that in the dependency injection of the API application. The implementation of these features is done when creating the API controllers, as depicted in Figure 2.

```

builder.Services.AddControllers()
    .AddOData(opt =>
        opt.AddRouteComponents("v1", GetEdmModel())
        .Filter()
        .Select()
        .Expand()
        .Count()
        .SkipToken())
    .AddNewtonsoftJson();

```

Figure 2 Adding the features of OData

Lastly in the implementation of OData, are the controllers. To register them as OData controllers, the controller class derives from `ApiController` that comes with the NuGet package. In this controller class are all the endpoints that are related to the EDM. The implementations of the custom endpoints are done like any other API method, with the

exception of adding them to the EDM model builder. The methods need to be added separately to be registered as endpoints. This can be seen in Figure 1.

```

0 references
public class UserController : ODataController
{
    [EnableQuery]
    0 references
    public SingleResult<UserDto> Get([FromODataUri] int key)
    {
        var result = this._userService.GetById(key);

        return SingleResult.Create(result);
    }
}

```

Figure 3 Controller and an API endpoint

The controller and a single Get method are represented in Figure 3. To implement an API method with OData, it needs [EnableQuery] annotation, that allows the different query options of OData and for it to receive parameters it needs the [FromODataURI] annotation.

The expandability of the API has been implemented with service classes and partial classes. The API controllers call service functions in similarly named files, that contain all the methods the controller needs. For example, as shown in Figure 3, User controllers method “Get” calls user services method “GetById”. To implement the code that is called by possible custom functions we have a “CustomUserService”, that derives from the base “UserService”. This custom service is a partial class, that can then override the code that is in a virtual method in the base service.

The final form of this API needed to be made so that it could be turned into a template that the generator could use. To make this happen some design choices had to be made. In the case of adding manual code to the API, the services had to be made into three separate files: “Service”, “CustomService” and “CustomerService.My”. The service itself has custom methods marked as virtual and all relevant class attributes as protected, so they can be used from deriving classes. Then the customService has a partial class called “CustomService”

that derives from the base service and a constructor method. Then the “CustomService.My” is the place that all manual code will be written. This class is a partial “CustomService” and overrides the base implementation in the base service. This kind of structure is depicted in Figure 4 and had to be done so that the code that the generator makes could be built and run as is from the generator without any modification. In addition to this, all files have been simplified by separating related segments of code into their own files. If possible, files have been formed so that they have the least number of variables to make the development of the generator easier. A good example of this is the “Program” file, where most of the code that needs parameters has been cut and made into its own class called “ProgramService”.

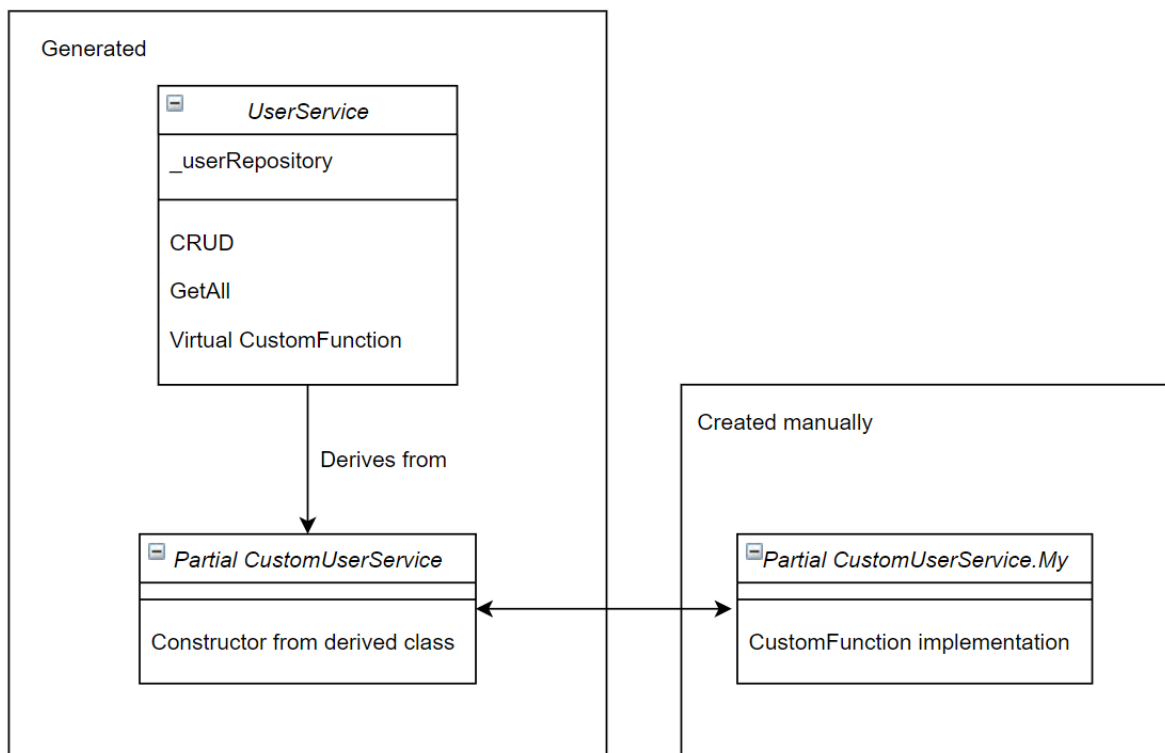


Figure 4 Structure of Service classes in the API source code

Lastly, the documentation of the API has been made with Swagger. The initial adding of swagger was straight forward. It needed to be added as a service in the initialization of the application as a service and then enabling it in the development environment. Relevant code is shown in Figures 5 and 6. The addition of comments that are shown in the Swagger UI were also implemented. This means that different things in the application code could be commented, and those comments would show in the Swagger documentation like shown in Figure 7.

```
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "SoulcoreExternalAPI", Version = "v1" });
    AddSwaggerDocumentation(c);
});
```

Figure 5 Adding Swagger to the API

```
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

Figure 6 Enabling Swagger in the development environment.

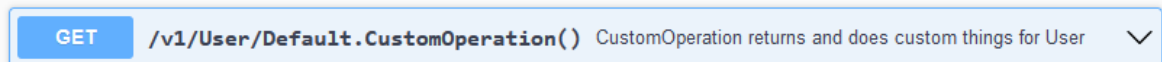


Figure 7 Comment in Swagger UI

The API does not implement authorization and does not utilize the existing resources of SC Software. The database access has not been implemented but a temporary system to get mock data for the purposes of this work has been implemented. The way data is fetched is a repository that accesses a static file with JSON formatted data that is then returned by the API. The final flow of data is then as follows: the controller calls the corresponding method in a service, and the service calls a repository method that gets the data from a static file.

3.3 Designing the generator

The generator will be of POC level, so the requirements for it are not demanding. The requirements are:

1. Generated API application can be built run as is.
2. The generator can take parameters in JSON format.

3. The baseline API is used as a template and target format.
4. The API is regeneratable.

The requirements for this work are simple because the final product of this thesis work will not be used in the final version of the API generator. The purpose of this generator is to find out what are the problems and lessons learned when creating an API generator. The first requirement is quite clear. It means that the generated result must be in a state that can be built and run without any modifications to the generated code. This makes the use of the generator easier and makes the testing process of the API a lot faster. The generator has to be able to take parameters, like models and project name. For this JSON was selected because it is easy to understand and has a lot of code libraries to help parse it. The third requirement is that the final product must resemble the baseline API that was created earlier for this work. To make it possible the API will be transformed into a template. The last requirement is probably the most important regarding this thesis. The generated API has to be regeneratable, meaning that the generated parts of the code can be generated again, without interfering with the possible manually written code that the API could have. This allows the changes in the generator to be transferred into APIs created with an older version of the generator. This also applies if the models that the API uses change. By regenerating the API, the model changes appear without manually changing anything.

In addition to the requirements there are other things to consider when designing the generator. Firstly, the parameters that the generator takes as input to create the API. The parameters could define everything, from the models the API returns to the possible custom functions it has. All of this must be defined and turned into parameters that can be used for the generator to work. Also, the abstraction level of these parameters must be considered as well. For example, when defining a model that the API uses, a single text field could be represented in a few different ways depending on the abstraction level. The definition could be the same one that it is in the code language that is used, which would be “string” in case of C#, or it could be plain language like “Text” or “Email”. In ideal conditions the abstraction level should be raised above the programming language to better accommodate MDD (Kelly and Tolvanen, 2008). The second thing to consider in the development of the generator is the syntax or symbols that it uses in the template files. The files have keywords in them that

need to be replaced by relevant terms when generating the API. For example, the project name must be changed and all the model references as well. This means that the template files will have to include different keywords that will be replaced, and those keywords have to be recognized and defined to be able to use the baseline API as a template.

The creation process will be following the code generator creation guidelines that were introduced by Kelly & Tolvanen (2008). In the case of this code generator the process will go as follows:

1. The baseline API is pasted into the output of the generator with template modifications.
2. All the different classes, such as controllers, services and models are reduced into single occurrences going through a model loop in the generator code.
3. Sections that cannot use the template code or need more modification based on the parameters are handled separately in the model loop.
4. All the different keywords are replaced with corresponding generated code, or parameters.

3.4 Implementation code generator

This chapter goes through the implementation of the code generator. The requirements set in the previous chapter and the other considerations have been addressed in the developed generator. The implementation of the code generator is a console program developed with C# dotNet 6.

The generator creates an API using OData. This API resembles the baseline version and is using dotNet 6 as well. The API program can be built and run as is without any modification, as was set in the requirements. The most challenging part of the development was defining

the parameters that the generator takes in. Those parameters are in JSON format, and define the following aspects:

1. Project name, defining the name of the API project.
2. Generation folder, defining the destination of the generated product.
3. Models, that contain the models that are the resources of the API.

The first and second parameter are easy to understand, but the model definition includes a list of model definitions. These definitions include the name, attributes, description, operations, and custom operations. All the parameters are represented in Figure 8. From the figure, the abstraction level can be seen as well. For this work, the abstraction level has been set very close to the programming language. The fields of the models are described by the actual variable types in C#, such as “int” for numbers and “string” for the text fields. The abstraction has been set on this level, because it does not affect the result of this thesis and there was no time to use on raising the level of abstraction.

```
{
  "ProjectName": "TestApi",
  "GenerationFolder": "C:/gen_out",
  "Model": [
    {
      "Name": "User",
      "Fields": {
        "Id": "int",
        "Name": "string",
        "Email": "string",
        "Birthday": "string"
      },
      "Description": "User is a user using the system",
      "Operations": [ "C", "R", "U", "D" ],
      "CustomOperations": [
        {
          "Name": "CustomOperation",
          "Description": "CustomOperation returns and does custom things for User",
          "Parameters": {
            "id": "int"
          },
          "ParameterComments": {},
          "ReturnType": "User",
          "HttpMethod": "Get"
        }
      ]
    }
  ]
}
```

Figure 8 Generator parameters

As was defined by Kelly & Tolvanen (2008), all files in the baseline API were inspected and analysed for patterns that can be looped through by model in the parameters and static content that stays the same with every generated API. Every model in the parameters results in a model, controller, service, service interface, custom service, and repository classes. This list does not include the files that are present regardless of the number of models in the parameters. After the different sections were defined, a set of keywords and markup language had to be defined. These keywords are used as placeholders in the template files to represent different parts of the code or some important names etc. For example, project name is a variable that can be given in the generator parameters and is put into places that are marked “&<<<projectName>>>&”. All the keywords are shown in Figure 9. There was also a need to for a keyword that was meant for more complex replacement than just single words. This code is something that is constructed in the generator and then placed into the template. The keyword for this is “&<<<@@@>>>&”

```
{
  "&<<<>>>&": "Placeholder",
  "&<<<@@@>>>&": "Code slot",
  "projectName": "API name, project name",
  "modelName": "Name of a model",
  "_modelName": "ModelName without capital letter",
  "diCode": "ProgramService dependency injections",
  "edmCode": "ProgramService EdmModel code",
  "description": "Comment"
}
```

Figure 9 Generator keywords as JSON

The process of implementing the generator went surprisingly easily as the process was described previously in this work. The most frustrating part was transforming the code of the baseline API to generator code so that the generated code works as it was very error prone because it was just text strings in the generator.

3.5 Evaluation of the implementation

The implementations were evaluated based on the requirements set for them in the previous chapters. For the API, the requirements can be found in chapter 3.1.2 and the generator

requirements can be found in chapter 3.3. This chapter will describe on how the requirements have been met, how it is to use the artifacts and how it addresses the needs of SC Software regarding API generation.

The generator developed for this thesis work is just a console application, that reads the parameter JSON file from the application files and writes strings into text files. The actual program is run from Visual studio, as it is only meant for research purposes, and does not need an executable version at the moment. The generator does not check the validity of the parameters, so for the generator to work as intended, the user has to manually make sure the parameters are correct. If the parameters are not as they should be the generator will give an exception and interrupts the generation. This exception can be anything, as the execution of the generation will stop at the point where the parameter is not correct, and this can result in multiple types of exceptions.

When the parameters are correct and the generation process is finished, the final API application can be found in the directory that is given in the parameters. The generator generates dotNet 6 solution file that can be opened with Visual Studio. The API is now ready to be built and run in Visual Studio. As defined before, the API is in a state that can be run, but it lacks viable data and the proper implementations of the custom API endpoints. The end result is a Swagger UI of the API, shown in Figure 10.

The screenshot displays the Swagger UI for the TestApi v1 API. At the top, the Swagger logo is visible, along with the text 'Supported by SMARTBEAR'. A dropdown menu shows 'Select a definition' with 'TestApi v1' selected. Below this, the API title 'TestApi v1 OAS3' is shown, followed by the URL 'https://localhost:7125/swagger/v1/swagger.json'. The main content area is titled 'Customer' and lists 13 endpoints, each with a colored button indicating the HTTP method (POST, GET, PATCH, DELETE) and a dropdown arrow. The endpoints are:

- POST /v1/Customer
- GET /v1/Customer
- GET /v1/Customer/\$count
- GET /v1/Customer({key})
- PATCH /v1/Customer({key})
- DELETE /v1/Customer({key})
- GET /v1/Customer/{key}
- PATCH /v1/Customer/{key}
- DELETE /v1/Customer/{key}
- GET /v1/Customer/Default.DoCustomThing() CustomOperation returns and does custom things
- GET /v1/Customer/DoCustomThing() CustomOperation returns and does custom things
- POST /v1/Customer/Default.UpdateCustomerStatus Sets customers status to given value
- POST /v1/Customer/UpdateCustomerStatus Sets customers status to given value

Figure 10 Swagger UI for the API

The Swagger shows all available endpoints in the API and even though only 6 are implemented in the code, them being the basic create, read, update, delete actions; get all action and the custom endpoint; there are a lot more shown in the Swagger UI. This is because of the implementation of OData, as Swagger shows all the different ways to call these endpoints. The same endpoint can be called with the URI “https://example.org/v1/Customer(1)” and “https://example.org/v1/Customer/1”. The custom endpoints won’t work unless they have been implemented properly. In the generated version the implementation only throws a “Not Implemented” type of exception.

In the current implementation, the documentation is handled by Swagger, as was shown on Figure 10. In addition to swagger, there was a need for more detailed comments in the API. These comments have been implemented for the models, custom endpoints and the parameters of the custom endpoints and they are all defined in the generator parameters. There is no particular reason why these were chosen to be implemented, as they are quite easy to add later if needed and are only here to demonstrate that there is a possibility to comment the API further than just what swagger generates.

Modification of the generated code in the API is as easy as just keeping the destination folder of the generated product the same. If there is a need to add a new custom operation or a new field to a model, it can just be added to the parameters of the generator. The generator deletes all of the generated files before it generates them again. This way we make sure we don't add complexity to the generator and the manual modification of the generated files is made pointless. As only the files are deleted, the folders stay in place and the manually added files stay in their places. The manually added files might need some fixes after generation in some cases but they are to be expected. These fixes are usually simple and are fast to fix.

The point of this thesis is to get a working API that is compliant with the OData standard, so the API needs to work as expected. To test the features, an API with customer and user mock data was created. By sending the following request *"https://localhost:7125/v1/Customer(12)"* we get the following response:

```
{
  "@odata.context": https://localhost:7125/v1/\$metadata#Customer/\$entity,
  "Id": 12,
  "Name": "Firstname12 Lastname12",
  "Email": firstname12.lastname12@example.fi,
  "Status": "12no-status"
}
```

The request gets a customer resource with the id "12", and this can be seen from the response. The response is recognizable as OData from the "@odata.context" parameter in the beginning of the response. We can make a filtering query, where we look for Users that have the character "1" in their name and select only the "Id" and "Email" values of those Users.

The request for this is `“https://localhost:7125/v1/User?$filter=contains(Name, '1')&$select=Id, Email”` and the response is the following:

```
{
  "@odata.context": https://localhost:7125/v1/\$metadata#User\(Id,Email\),
  "value": [
    {
      "Id": 10,
      "Email": firstname10.lastname10@example.fi
    },
    {
      "Id": 11,
      "Email": firstname11.lastname11@example.fi
    },
    {
      "Id": 12,
      "Email": firstname12.lastname12@example.fi
    }
  ]
}
```

As we can see, there are only Users with the character “1” in the name and only the Email and Id fields are in the response.

OData has many powerful features like filtering, selecting, and search that are easy to implement and use. For the API in this thesis, the search function, and other features, that require the data to have complex values as attributes in the model, are not functional at the moment. In the case of filtering on complex type, the feature would work if a complex type was added to the model. The search function is something that would have to be implemented separately, as it does not know what it would compare the value it is given. For example, if we were to make the following request `“https://localhost:7125/v1/User?$search=Firstname”` the API would return all users, as there is no definition for a custom search that would match the parameter to anything.

A simple implementation of the custom endpoints has also been developed for the API. They work like any other endpoint listed in the Swagger UI. For example if we were to call a custom endpoint for the Customer we can make a call `“https://localhost:7125/v1/Customer/DoCustomThing()”` and we get the following response:

```

{
  "@odata.context": https://localhost:7125/v1/\$metadata#Collection\(TestApi.Models.CustomerDto\),
  "value": [
    {
      "Id": 0,
      "Name": "Custom name",
      "Email": custom.name@email,
      "Status": "alive"
    }
  ]
}

```

This is as expected, as the function was defined to return a single customer. The implementation of the function is shown in Figure 11.

```

4 references
public partial class CustomCustomerService
{
  3 references
  public override IQueryable<CustomerDto> DoCustomThing(int id)
  {
    return new List<CustomerDto>()
    {
      new CustomerDto()
      {
        Id = id,
        Name = "Custom name",
        Email = "custom.name@email",
        Status = "alive"
      }
    }
    .AsQueryable();
  }
}

```

Figure 11 Implementation of the DoCustomThing endpoint

Figure 11 shows the implementation that has been implemented manually in the generated API and returns a single User back to the requesting client. In this scale, implementing the custom functions is not a difficult task, as the structure of the application is clear and simple. The implementing only needed a new file called "CustomCustomerService.My.cs" and defining the partial class "CustomCustomerService" in it. Because the "CustomCustomerService" class is the only Service registered in the start of the application for the Customer resource, the manually written implementation is automatically used when calling the custom endpoint. If the custom functions needed more complex logic, there is nothing preventing creating more structure for the code. As the generator only cares for the

files it creates, the developer of the API could add anything to it, as long as the generated files or their folders are not modified. The folders can have manually created files that won't be affected.

The requirements set for the API and generator have all been met in the implementations of the artifacts and SC Software can use these artifacts as a good starting point when they decide to implement a production version of the generator. As one of the most important goals for this thesis was to research how APIs could be generated, the artifacts show exactly that. As the ideology behind the artifact implementations resembles the development ideology in SC Software, they show even more promise that API generation is possible and that the work in this thesis can solve their need for an API implementation method.

4 Findings

This thesis was set out to do research into API development with a code generator. The focus of the thesis was to find out if API development was plausible with a generator, implement a POC of a generator that creates APIs and to see what kind of difficulties and considerations there are in a project like this. This chapter goes through the findings from the creation of the two artifacts and answers the research questions of the thesis.

4.1 RQ1: Is it viable to use a code generator for customer specific API creation?

This thesis was set out to find out if it was viable to generate APIs with the assistance of a code generator. This feature has been frequently asked by the target companies customer base and SC Software is keen to know what the implementation of this feature could look like in their current code generator. In this work, a generator that can generate basic web APIs was created following the requirements that were set in chapters 3.1.2 and 3.3. The result is a simple code generator, that was created by following the steps defined by Kelly & Tolvanen (2008). By using JSON formatted input parameters, it can create an API application with basic CRUD functions and custom functions. The implementation still has some open questions, but it is viable to create APIs with a code generator. The APIs created in this thesis work are not useable in production and the generator cannot be integrated into the existing SC Software generator as is, but this study has solved a lot of question when making an API generator for them.

4.2 RQ2: Does code generated APIs bring work efficiency benefits compared to coding them manually?

The second question is closely related to the first one and tries to enlighten if generating the APIs is an efficient way to create APIs. As the needs of SC Software's customers' needs

have not been defined properly yet, it is possible that generating won't be the most efficient way to do APIs for them. For this work, the baseline API is used as an example solution, that represents the basic needs a customer might have for a web API, meaning simple CRUD operations and additionally some other more complicated or custom functions including all the functionality that comes with complying with OData.

The process of developing the API started with planning and coming up with everything that the API needs using a similar ideology that SC Software uses in their current development. This included the research for OData and other similar solutions, like GraphQL. After the research was the development of the API and all its features using OData. The process is similar for all possible APIs that SC Software could implement in their existing solutions. It is important to note that most of the solutions provided by SC Software are generated and the general architecture is usually similar between the generated applications. This means that if they wanted to add a web APIs to their applications, they would have to write a lot of the code multiple times. The development of the baseline API took one developer 10 workdays. This included mostly only the development and fixing technical problems, as OData was previously unfamiliar technology for the developer and some research. This time can be improved significantly, if the technologies used are familiar and there have been other similar APIs developed previously by the developer.

When using the generator, the same API can be developed in a fraction of the time compared to the manually written one. To develop the API, we only need to define the parameters used by the generator and implement the code required by the possible custom functions in the generated product. This can be done in minutes, or faster by one developer. The process is longer, if the API has not yet been defined, meaning that we don't know what kind of models we want to include. The result is the exact same as the one in the baseline API. Already from this we can see that the generation is more efficient, but the efficiency is further demonstrated when we want to develop the API further. To add a new model and controller manually for the API, basically the same code has to be written again. When doing this manually, the process includes developing and testing the result but by generating, it we can just write a few more lines of parameters and get the same result, but it won't have to be tested as

thoroughly as the manual one. This study further proves the advantages of code generation, that were introduced in chapter 2.3. Additionally, the efficiency comes in when we want to get into creating the custom parts of the API. The API needs the basic functionality first before we can develop anything more complicated. If the API is generated, all necessary infrastructure and basic CRUD features are ready, and the developer can concentrate on the custom functions and business logic of the API.

5 Discussion and future research

In this chapter we discuss the results of the study and cover possible future research that could be made based on this subject and case. Discussion also goes through things that affected the results and what could have been done differently regarding the baseline API and the generator.

5.1 Discussion

The results of this thesis are in line with the results of previous studies, as can be seen from comparing the studies introduced in chapter 2 to this work's findings. This holds true for both the viability of generating APIs, as well as the efficiency benefits of code generation and MDD. The results also confirm the results of the previous studies. Overall, the results are positive, and SC Software can add an OData API generator to their existing generator in a way that serves their customers.

The result shows that it is viable to develop APIs with code generation and it is in general a very efficient way to develop them if the generator is defined in a way that suits the developed application. As was noted previously in the study, the generator is suitable only if the project scope has been strictly defined (Kelly and Tolvanen, 2008). The generator that SC Software uses at the moment could be made to include an API generator as well. This is because the ideology behind the current generator also suits the needs of an API very well. The result for viability is also in line with the study by Ed-douibi et. al. (2018) as they also managed to make a similar generator. In general, full generation of APIs from database to usable endpoints has not been researched that much. This study shows it is possible and that there are a lot of different ways to practically make them and that there is room for future research as well.

The efficiency results found in this study are very similar to the previous code generation and MDD efficiency studies, as the results show a reduction in time when creating APIs by generation. This is supported by multiple studies (Kärnä et al., 2009; Tolvanen, 2004; Tolvanen and Kelly, 2016)

Although the result of this study is positive, the generated artifacts could be made in multiple other ways as well. Because this version of the generator works as intended, it would be possible to make even quite large modifications or changes to the generator, structure of the generator and API and the technologies used in the API. Even with these changes, the result of the study would not be affected. The biggest changes to the implementation would probably be the change of OData and the abstraction level of the DSM.

One major change would be the change of the abstraction level of the DSM and generator as it is on very low level now. The level of abstraction is one thing that most probably should be raised, as the current production generator has a higher level of abstraction already. By raising the abstraction level, the usability of the API generator will increase and raise the efficiency of the developers as well (Tolvanen and Kelly, 2016). Another possibility is also to make the DSM graphical, as is done in SC Software already, or use another tool, like UML, as that has also been proven usable by multiple studies. (Bajovs et al., 2013; Boutekkouk, 2010; Ed-douibi et al., 2018; Vogel-Heuser et al., 2005)

As this is just a study for viability and general mapping of how web APIs could be implemented in SC Software's existing code generator, the generator is just a POC. Thus, it won't be integrated into the existing generator as is and is not ready for production use by any means. There are a few considerations that must be addressed in this generator before it can be integrated like the lack of database access, authentication, and real requirements of customers regarding this topic. For these reasons the integration of an API generator could take a while and the best way to make APIs could change in a major way during this time. This is why it would be good to consider if OData is the right choice and possibly replace it by something else, like GraphQL. As it is a growing novel technology used by big companies in their products, the usage of it will probably grow in the future. As it is very similar to

OData, this study could have used GraphQL already. Although generating GraphQL APIs has not been researched very much, a GraphQL wrapper could be generated for REST services. (Wittern et al., 2018)

Also, the entire structure of the baseline API should be reviewed closer and see if it needs more development for the production version. For example, the implementation of the manual code that the custom endpoints need can be changed. The current implementation is also probably not suitable for the possible usage of GraphQL. It is also important to evaluate, if it is necessary to include the custom endpoints in the generator parameters and what is necessary to include in the generation overall. It could be possible that it is easier to implement the custom endpoints manually all together. This would mean that the generator just needs to create the CRUD functionality and everything else is handled by a developer. The advantages of this lie in defining the API and it would mean that when defining the generator parameters, we do not need to think about the implementation of the API at all. The only things to include would be the models. Now the parameters include the custom endpoints, and it is necessary to understand the source code of the API. If we also raise the abstraction level of the MSD and generator, the custom functions might complicate the generation.

In a case where an API needs to be added to an existing solution, the API could be very easy to implement, as some of the features customers want, could already have been developed into the application and they just need another way to access it. For the API this just means calling the existing functionality and returning the result and that the API itself does not need to implement anything complicated. There is also another thing to be considered as well if we want to add an API to an existing solution: what do we want to expose of the application? This is a question that has to be asked in any case, because of the nature of a web API. For example, the models that the API returns to the client could be entirely different from the existing model that SC Software's current generator creates and uses.

5.2 Future research

There are a few points of interest in the future research of this study. The viability of generating APIs with code generation does not need that much looking into, but it would be good to look better into the efficiency benefits of it. The results of this study are not very comprehensive and need more data to support the findings.

Even though OData is a standard, it could be that it won't be a relevant way of implementing a web API in the future, as technologies like GraphQL are rising in popularity. This would be a good place to further expand this study and compare OData and GraphQL with code generation. It would be beneficial for SC Software to know if there are, for example performance differences, especially in production environments, where there might be unforeseen problems.

It also could be beneficial to look better into the benefits and drawbacks of API generation versus coding them manually in SC Software. As the company grows larger, and the number of customers gets higher, their requirements could also get more complicated. If they for example had a requirement too complicated for our API generator, that it would be too hard or impossible to implement, it could be easier to implement the API without the generator.

6 Conclusion

This master's thesis was set out to find answers for SC Software regarding generating web APIs with a code generator. The research questions were, "Is it viable to use a code generator for customer specific API creation?" and "Does code generated APIs bring work efficiency benefits compared to coding them manually?" The results of this study show that it is viable to generate customer specific APIs and that they do bring efficiency benefits for the company.

This study shows that generating APIs is something that SC Software can and should do as a next big step in their generator development. The artifacts created in this study are POC-level implementations, but they contain a lot of valuable information regarding some design decisions and possible problems. OData, that was used as a best-practices-solution for the generated API is a solid choice as the defining technology, but GraphQL is worth looking into when developing the production version of the generator.

The results of this study are in line with previous studies and literature regarding this topic. This study verifies the results that have previously been stated. APIs have been generated with a code generator before and usage of code generators does bring efficiency benefits.

This thesis generates a few points that could be looked at in future work. For example, the results regarding this work's efficiency benefits are based on only a few data points and should be investigated more in the future. Another thing is changing OData to GraphQL as the best-practices solution in the API, as it is a good alternative. The last thing to look at in future research is the drawbacks of code generation as SC Software grows and their customer base gets more diverse.

References

- Amazon, 2023. What is an API? - Application Programming Interfaces Explained - AWS [WWW Document]. Amaz. Web Serv. Inc. URL <https://aws.amazon.com/what-is/api/> (accessed 1.24.23).
- Bajovs, A., Nikiforova, O., Sejans, J., 2013. Code Generation from UML Model: State of the Art and Practical Implications. *Appl. Comput. Syst.* 14, 9–18. <https://doi.org/10.2478/acss-2013-0002>
- Bell, R., 1998. Code generation from object models. *Embed. Syst. Program.* 11, 74–88.
- Bhattacharyya, S.S., Kumar, S., 2021. Study of deployment of “low code no code” applications toward improving digitization of supply chain management. *J. Sci. Technol. Policy Manag.* 14, 271–287. <https://doi.org/10.1108/jstpm-06-2021-0084>
- Boutekkouk, F., 2010. Automatic SystemC Code Generation from UML Models at Early Stages of Systems on Chip Design. *Int. J. Comput. Appl.* 8, 10–17. <https://doi.org/10.5120/1215-1744>
- Brito, G., Mombach, T., Valente, M.T., 2019. Migrating to GraphQL: A Practical Assessment, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE. <https://doi.org/10.1109/saner.2019.8667986>
- Dzidek, W.J., Arisholm, E., Briand, L.C., 2008. A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. *IEEE Trans. Softw. Eng.* 34, 407–432. <https://doi.org/10.1109/TSE.2008.15>
- Ed-douibi, H., Izquierdo, J.L.C., Cabot, J., 2018. Model-driven development of OData services: An application to relational databases, in: 2018 12th International Conference on

Research Challenges in Information Science (RCIS). IEEE.

<https://doi.org/10.1109/rcis.2018.8406667>

Fadjukoff, L., 2021. Toimialakohtainen mallinnus sovelluskehityksessä. Tampereen yliopisto.

GraphQL, 2022. GraphQL | A query language for your API [WWW Document]. URL

<https://graphql.org/> (accessed 2.19.23).

Halili, F., Ramadani, E., 2018. Web Services: A Comparison of Soap and Rest Services.

Mod. Appl. Sci. 12, 175. <https://doi.org/10.5539/mas.v12n3p175>

Hurlburt, G., 2021. Low-Code, No-Code, What's Under the Hood? IT Prof. 23, 4–7.

<https://doi.org/10.1109/mitp.2021.3123415>

Johannesson, P., Perjons, E., 2021. An Introduction to Design Science. Springer

International Publishing. <https://doi.org/10.1007/978-3-030-78132-3>

Kärnä, J., Tolvanen, J.-P., Kelly, S., 2009. Evaluating the use of domain-specific modeling in practice. 9th OOPSLA Workshop Domain-Specific Model.

Kelly, S., Tolvanen, J.-P., 2008. Domain-Specific Modeling : Enabling Full Code

Generation, 1st Edition. ed. IEEE Computer Society Press.

OData, 2022. OData - the Best Way to REST [WWW Document]. URL

<https://www.odata.org/> (accessed 1.29.23).

Paolone, G., Marinelli, M., Paesani, R., Felice, P.D., 2020. Automatic Code Generation of

MVC Web Applications. Computers 9, 56. <https://doi.org/10.3390/computers9030056>

- Red Hat, 2020. What is a REST API? [WWW Document]. URL <https://www.redhat.com/en/topics/api/what-is-a-rest-api> (accessed 1.24.23).
- Rumpe, B., 2017. Agile and UML-Based Methodology, in: Agile Modeling with UML. Springer International Publishing, pp. 9–31. https://doi.org/10.1007/978-3-319-58862-9_2
- Samaranayake, M., 2022. Choosing between GraphQL vs Odata API [WWW Document]. Medium. URL <https://blog.bitsrc.io/choosing-between-graphql-vs-odata-api-823856bce8c2> (accessed 3.15.23).
- Sanchis, R., García-Perales, Ó., Fraile, F., Poler, R., 2019. Low-Code as Enabler of Digital Transformation in Manufacturing Industry. Appl. Sci. 10, 12. <https://doi.org/10.3390/app10010012>
- SC Software Oy, 2023. Our method [WWW Document]. SC Softw. Oy. URL <https://www.scsoftware.fi/en/about/our-method/> (accessed 3.25.23).
- Sebastián, G., Gallud, J.A., Tesoriero, R., 2020. Code generation using model driven architecture: A systematic mapping study. J. Comput. Lang. 56, 100935. <https://doi.org/10.1016/j.cola.2019.100935>
- Setälä, M., Abrahamsson, P., Mikkonen, T., 2021. Elements of Sustainability for Public Sector Software – Mosaic Enterprise Architecture, Macroservices, and Low-Code, in: Wang, X., Martini, A., Nguyen-Duc, A., Stray, V. (Eds.), Software Business, Lecture Notes in Business Information Processing. Springer International Publishing, Cham, pp. 3–9. https://doi.org/10.1007/978-3-030-91983-2_1
- Swagger, 2023. API Documentation & Design Tools for Teams | Swagger [WWW Document]. URL <https://swagger.io/> (accessed 3.12.23).

Telea, A., Voinea, L., 2011. Visual software analytics for the build optimization of large-scale software systems. *Comput. Stat.* 26, 635–654. <https://doi.org/10.1007/s00180-011-0248-2>

Tolvanen, J.-P., 2004. MetaEdit+: domain-specific modeling for full code generation demonstrated [GPCE], in: *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. Presented at the OOPSLA04: ACM SIGPLAN Object Oriented Programming Systems and Applications Conference, ACM, Vancouver BC CANADA, pp. 39–40. <https://doi.org/10.1145/1028664.1028686>

Tolvanen, J.-P., Kelly, S., 2016. Model-Driven Development challenges and solutions: Experiences with domain-specific modelling in industry, in: *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. pp. 711–719.

Tolvanen, J.-P., Rossi, M., 2003. MetaEdit+: Defining and using domain-specific modeling languages and code generators, in: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*. pp. 92–93. <https://doi.org/10.1145/949344.949365>

Vogel, M., Weber, S., Zirpins, C., 2018. Experiences on Migrating RESTful Web Services to GraphQL, in: Braubach, L., Murillo, J.M., Kaviani, N., Lama, M., Burgueño, L., Moha, N., Oriol, M. (Eds.), *Service-Oriented Computing – ICSOC 2017 Workshops*. Springer International Publishing, Cham, pp. 283–295.

Vogel-Heuser, B., Witsch, D., Katzke, U., 2005. Automatic Code Generation from a UML model to IEC 61131-3 and system configuration tools, in: *2005 International Conference on Control and Automation*. IEEE. <https://doi.org/10.1109/icca.2005.1528274>

Wittern, E., Cha, A., Laredo, J.A., 2018. Generating GraphQL-Wrappers for REST(-like) APIs, in: Mikkonen, T., Klamma, R., Hernández, J. (Eds.), *Web Engineering, Lecture*

Notes in Computer Science. Springer International Publishing, Cham, pp. 65–83.

https://doi.org/10.1007/978-3-319-91662-0_5