Tampere University

Antti Eskelinen

UTILIZING CODE GENERATION IN SINGLE-PAGE APPLICATIONS

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Kari Systä
Terhi Kilamo
April 2021

ABSTRACT

Antti Eskelinen: Utilizing code generation in single-page applications
Master of Science Thesis
Tampere University
Information Technology
April 2021

In Wapice, it has been noticed that setting up single-page applications takes a significant amount of time. In each project, the developers need to set up architecture, libraries, authentication and server communication. A generic React application template had been created in Wapice as a solution for the issue. However, as the template is static, it cannot contain features that vary between projects, such as authentication.

In this thesis, the possibility to utilize code generation for the issue mentioned above was researched. The goal was to determine if code generation could reduce repetitive and mechanical tasks in single-page application projects. Also, the parts of single-page applications that could benefit most from code generation were sought.

Code generation can be defined as programming on a higher abstraction level that is currently available. Code generation can improve quality and productivity. However, implementing code generators contain similar risks as any other type of software development.

The research was done as a design and implementation project. The requirements based on nine existing production-grade React projects and the existing React template. The developers of the projects and the template were also interviewed. Three different generators were decided to be implemented based on the interviews: A generator for generating back-end communication code, boilerplate generator and OpenID Connect authentication generator.

The OpenAPI Generator was selected for generating back-end communication code. For the other two generators, there were no ready-to-use tools available. The generators were initially designed to be implemented on top of Hygen. However, due to technical challenges, Hygen was replaced with Plop later in the implementation. Plop and Hygen are tools for developing small scale template-based generators.

Finally, the three generators were graded based on how easily they could be implemented and maintained, how much manual work could be saved using the generator and how easy it is to customize and extend the tool. The OpenAPI Generator got the best scores for the three first criteria. The boilerplate generator was considered easier to maintain and provide better productivity gains from the two Plop generators. The Plop generators were more straightforward to customize than the OpenAPI Generator.

The research found out that the parts that can be modelled on a higher abstraction level, such as back-end communication code, are suitable for code generation. Boilerplate code was found out to be a good target for smaller self-made generators.

Keywords: Code generation, Single-page application, OpenAPI Generator, Template-based generation, Plop, Hygen

TIIVISTELMÄ

Antti Eskelinen: Koodigeneroinnin hyödyntäminen yhden sivun sovelluksissa
Diplomityö
Tampereen yliopisto
Tietotekniikka
Huhtikuu 2021

Wapicella on huomattu, että yhden sivun sovellusten pystyttämiseen kuluu huomattava määrä aikaa. Tyypillisessä projektissa kehittäjän tulee pystyttää arkkitehtuuri, kirjastot, autentikointi ja palvelinkommunikaatio. Ratkaisuna tähän ongelmaan Wapicella on tehty yleiskäyttöinen React pohjaprojekti. Koska pohjaprojekti on yleiskäyttöinen, siihen ei voi sisällyttää projektikohtaisia ominaisuuksia, kuten autentikaatiota.

Työssä tutkittiin mahdollisuutta hyödyntää koodigenerointia edellä mainittuihin ongelmiin. Tavoitteena oli ymmärtää, voisiko koodigeneroinnilla vähentää toisteista työtä yhden sivun sovelluksien toteuttamisessa. Lisäksi haluttiin selvittää, mitkä osat yhden sivun sovelluksista soveltuvat parhaiten generoitaviksi.

Koodigenerointi voidaan määritellä tarkoittamaan ohjelmointia korkeammalla abstraktiotasolla, joka on yleisesti saatavilla kyseisellä hetkellä. Koodigeneroinnilla voidaan parantaa laatua ja tehokkuutta. Toisaalta koodigeneraattorien toteuttamiseen liittyvät samat riskit kuin minkä tahansa ohjelmiston kehittämiseen.

Tutkimus tehtiin suunnittelu- ja toteutusprojektina. Vaatimukset määriteltiin yhdeksän tuotantotasoisen React-projektin ja olemassa olevan React projektipohjan perusteella. Lisäksi projektien ja projektipohjan kehittäjiä haastateltiin. Haastattelujen perusteella päätettiin toteuttaa kolme erilaista generaattoria: Generaattori palvelinkommunikointikoodin generoimiselle, boilerplate-generaattori ja generaattori OpenID Connect autentikoinnin generoimiselle.

Palvelinkommunikaatiokoodin generoimiseen valittiin työkaluksi OpenAPI generaattori. Kahdelle muulle generaattorille ei löytynyt valmiita työkaluja, joten alunperin ne päätettiin toteuttaa itse hyödyntäen Hygen-nimistä työkalua. Kohdattujen teknisten ongelmien vuoksi Hygen jouduttiin vaihtamaan Plop-työkaluun toteutuksen aikana. Plop ja Hygen ovat työkaluja, joilla voidaan toteuttaa pieniä mallipohjaisia generaattoreita.

Toteutetut kolme generaattoria arvioitiin lopuksi. Arviointikriteereinä olivat toteuttamisen helppous, ylläpidettävyys, vähentyneen manuaalisen työn määrä ja muokattavuus. OpenAPI Generator sai täydet pisteet kolmen ensimmäisen kriteerin osalta. Boilerplate-generaattorin arvioitiin olevan kahdesta Plop-generaattorista helpommin ylläpidettävä ja tuottavan enemmän hyötyä. Plop-generaattorit olivat helpommin muokattavissa kuin OpenAPI Generator.

Tutkimuksen perusteella generoitaviksi soveltuvat hyvin osat, joita voidaan mallintaa korkeammalla abstraktiotasolla. Esimerkiksi palvelinkommunikointi voidaan mallintaa OpenAPI määrittelyn avulla. Boilerplate-koodin huomattiin soveltuvan hyvin generoitavaksi pienillä itse toteutetuilla generaattoreilla.

Avainsanat: Koodigenerointi, Yhden sivun sovellus, OpenAPI Generator, mallipohjainen generointi, Plop, Hygen

PREFACE

Due to the global Corona pandemic, this thesis was written under unusual circumstances. Lectures were held remotely, the campus was closed, and most free-time student activities were cancelled.

Despite the unusual times, this thesis could be finished with the support of various people. I want to thank my responsible supervisor, prof. Kari Systä, for the precise and supportive feedback. The thesis was done for Wapice, where I also got much support. I want to thank Jussi Haapanen for generating ideas for the thesis and encouraging me to choose the topic. I want to also thank Otto Bothas for helping me to share the findings with other Wapiceans. Besides, I am grateful to each of the interviewed Wapice experts for the insight I was given in the interviews. I am thankful for the great colleagues and the flexibility that Wapice has offered me during the last three years. Finally, I would like to thank my family and friends for their invaluable encouragements and support.

Hopefully, the pandemic is closing to an end as more and more people are getting vaccinated. As my grandmother used to say: *"Parempi hyvä päivä edessä kuin takana"*.

Tampereella, 18th April 2021

Antti Eskelinen

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
CLI	Command-line interface
EJS	Embedded JavaScript Templates
HTTP	Hypertext Transform Protocol
MVC	Model-View-Controller
NPM	Node Package Manager
OIDC	OpenID Connect
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
SPA	Single-Page Application
WSDL	Web Service Definition Language

1. INTRODUCTION

Many areas of software engineering are automated. Developers can build highly optimised binary code from source code by entering a single command into the command-line. The program can then be tested by running automatic tests. If DevOps practices are applied, building, testing and deploying the code from the developer's computer to a public server can happen automatically, even multiple times a day [1].

There also exists many tools that are used to automate writing the code. Integrated development environments often contain such features. For example, they can offer auto-complete suggestions when the developer writes code. They often also include wizards for creating template projects, which can be used as a starting point for new applications. These all are small scale examples of code generation.

This thesis focuses on automating parts of creating new single-page web applications by using code generation. The study is done as a case study for Wapice Ltd. In Wapice, it has been noticed that it takes a considerable amount of time to set up a new single-page application project. In a typical single-page application, the developer needs to create a foundation for architecture layers, install and choose libraries, configure authentication and implement communication to the server. As most single-page applications contain those elements, it is inefficient to create them from scratch for each new project.

As a solution for this problem, a generic code template has been developed in Wapice. The template is a React project combined with commonly used React components, such as the Redux library for state management and React Router for page routing. However, as the template needs to be generic, some features like authentication are challenging to include in the template. Some projects do not require authentication at all, while others use different authentication standards. In practice, templates often contain also features that all developers do not prefer. For example, some developers may prefer to use the Redux Thunk library for managing side-effects, while the template created by Wapice comes with the Redux Saga library [2, ch 6.]. Changing the library requires work, which decreases the benefits of using the template.

In this thesis, the possibility of using code generation to overcome some of the template's issues is researched. It might be possible to generate the project-specific features, such as authentication, to the template. Code generation could increase development speed and efficiency at the beginning of new projects. The research scope is limited to single-page applications built on React framework as it is widely used in Wapice.

The research seeks answers to the following research questions:

1. Could we reduce repetitive and mechanical tasks at the beginning of web software projects by utilising code generation alongside static templates?
2. What parts of single-page applications could benefit most from code generation?

The research is done as a design and implementation project. The research starts by investigating the sources of nine existing React projects and a React application template created in Wapice. The developers of the projects are interviewed to get also subjective viewpoint. Requirements for the template and generator tools are specified based on the investigation. The potential single-page application components that could be generated are identified.

After the requirements for the generator are found, the implementation phase may start. The implementation includes finding suitable tools, implementing a new React application template, and configuring and customising generators to generate the previous phase's identified components.

At the end of the research, the implementation is reviewed. Each generator tool is graded based on how easily it could be implemented, how maintainable the solution is, how much work could be saved by using the generator and how easy it is to customise and extend it.

The thesis starts with a brief literature review about code generation theory in chapter 2. In the chapter, code generation is analysed from both technical and project management perspective. In chapter 3, the single-page application development tools that are relevant in the context of this thesis are described. The requirements analysis and design for the implemented generator tools are described in chapter 4. Chapter 5 describes the implemented tools and the used implementation processes more detailedly. In chapter 6, the implemented tools are reviewed against each other. Finally, the conclusions about the review results are described in chapter 7.

2. THEORETICAL BACKGROUND

2.1 Abstraction

Abstraction is a way to manage complexity. It allows people to describe complex ideas briefly with a few words. It is easier to use the term "car" than to say "a metal chassis on four wheels and a motor in the front". Software entities are among the most complex human constructs. They have a vast number of possible states, each part is unique, and the number of interactions between the parts grows in a nonlinear fashion compared to the number of the parts [3].

Using different levels of abstractions is an integral part of software development. A software developer does not need to know about implementation details of operating systems or compilers to write code for them. To shield themselves from the enormous complexity, software developers should use the highest possible level of abstraction available [4, p. 845].

In this thesis, the term code generation is used as a synonym for automatic programming and does not refer to compilers. However, in history, the term automatic programming has first referred to assemblers, then later it has referred to generating machine code from higher-level programming languages such as Fortran. By extrapolating this development, one can see that the term automatic programming refers to programming at a higher abstraction level that is currently available. The research on automatic programming focuses on implementing higher-level languages. [5]

2.2 Model-driven development

One software engineering branch that focuses on developing in a higher abstraction level is called model-driven development. As the name suggests, in model-driven development the higher level abstractions, models, are the main artifacts of development. [6] Models can be used to describe a software system independently from technical implementation [7, p. 4]. Even though the term is widely used in literature, there is no generally accepted definition for models [7, p. 14][8]. Kühne defines that models are abstractions that allow making of predictions or inferences from systems [9]. In practice, general purpose languages like Unified Modeling Language (UML) or domain-specific languages are used to model systems [6]. Section 3.1 presents two other possible formats that are used to model web services: OpenAPI and WSDL. In literature, the terms model-driven engineering and model-driven software development are used synonymously to model-driven development [7].

Because models are independent of the technical implementation, there is a gap between a system's model and the actual implementation. The gap can be bridged with code generators similarly as compilers bridge the gap between machine code and code written in a programming language. [7, p. 4] When model is transformed into another format, for example when low-level code is generated from UML, it is called model transformation [8].

Domain-specific languages are programming languages whose expressiveness is focused and limited to a particular domain [10]. Thus, domain-specific languages operate on a higher abstraction level than general-purpose languages. Similarly to interpreted and compiled languages, code written in domain-specific language can be either executed directly or transformed to a lower-level code [10]. The generated lower-level code can be some general-purpose language such as Java or C++.

The successful applying of model-driven development is rarely done by using a top-down approach for whole systems. Instead, it is used flexibly with other methods whenever appropriate. The majority of companies using model-driven development prefer to use small domain-specific languages over general-purpose languages. [11][12]

2.3 Code generation techniques

2.3.1 Different types of code generators

In model-driven development a code generator often transforms the higher-level model to lower-level code. This is called model-to-text transformation [13]. There are multiple different ways to implement this in practice [14, ch. 2][10, ch. 8]. The size and complexity of generators vary greatly. Some generators are small coding helpers where as other generate the bulk of an application. [14, ch. 1]

Generators can be divided into two different categories by their usage: passive and active. Passive generators produce code that can be manually edited. The generator does not maintain any responsibility for the code after it is created. For example, the "wizards" for creating new projects in integrated development environments are examples of passive generators. Active generators, on the other hand, maintain responsibility for the generated code. When the generated code needs to be updated, the developer reruns the generator with different inputs. [14, ch. 2]

Maintaining synchronization between the model and the generated code is not a trivial task for active generators. If generated code is edited manually, the modifications may be lost if the model is later updated and the code is regenerated. The simplest way to solve this issue is to deny developers from editing any generated code and only allow additive changes. The generated code might be separated from other code by comments. [7, pp. 32-33] Alternatively, the only allowed way to edit the generated code might be by inheriting the generated class [10, ch. 57][7, pp. 32-33]. Automatic updating of the model based on the edits made to the object code is theoretically possible if the object language and the model are on the same abstraction level. [7, pp. 32-33] However, this contradicts one of the core benefits of automatic programming, which is to enable programmers to write code at a higher abstraction level.

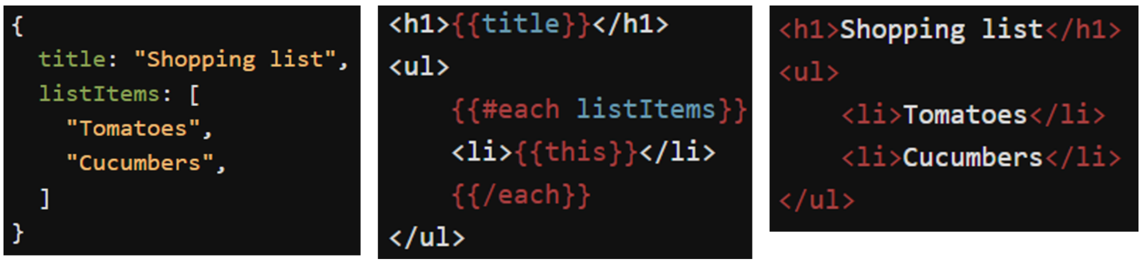
From an implementation viewpoint, the generators can be divided into homogeneous and heterogeneous generators. The language in which the code generator is written is called metalanguage. Correspondingly, the language of the generated code is called object language. For homogeneous generators, the same language is used as a metalanguage and as an object language. In heterogeneous code generation, the metalanguage and the object language are two different languages. [15, pp. 4-6]

The definition of homogeneous generating allows including some programming languages' metaprogramming features under code generation [15, pp. 4-6]. C++ templates and Java generics are examples of such features. In Java generics, the Java compiler replaces the generic parts of the code with actual types when the code is built [16]. As the generating is done by a compiler, it can take advantage of syntax and type checking. In this thesis, those metaprogramming features of compilers are considered an integral part of programming languages and therefore excluded from the scope.

2.3.2 The string concatenation approach

Many different approaches to implement code generators emerge in literature [15, p. 6][7, pp. 27-34][13]. A simple code generator can be based only on string concatenation and print statements of some general purpose language [15, p.7][13][7, p. 28]. This approach has its limitations in terms of scalability. The generator logic and output description get easily mixed up. Also special characters such as quotation marks need often be escaped explicitly. [7, p. 28] This increases complexity and reduces maintainability. Therefore, string concatenation can be used mainly to implement small and simple coding helpers. As developers are already familiar with the only tool involved – which is the programming language – there is no learning curve to utilize this method [7, p. 28].

It is possible to reduce the complexity from string concatenation by using suitable abstractions in the metacode. The same design patterns that are used in typical software development also apply to code generators. For example, the string concatenation can be hidden behind an application programming interface (API) [7, p. 28]. The interface



```
{
  title: "Shopping list",
  listItems: [
    "Tomatoes",
    "Cucumbers",
  ]
}
```

```
<h1>{{title}}</h1>
<ul>
    {{#each listItems}}
    <li>{{this}}</li>
    {{/each}}
</ul>
```

```
<h1>Shopping list</h1>
<ul>
    <li>Tomatoes</li>
    <li>Cucumbers</li>
</ul>
```

Figure 2.1. *From left to right: A JSON model, a Handlebars template and generated output based on the model and the template.*

may use concepts from the object language such as "class" or "parameter". The visitor pattern is used to map the API objects to corresponding code [13][7, p. 29].

2.3.3 Template-based generators

One method to avoid complexity caused by escaping special characters and string concatenation is to use templates. Template-based code generators can be split into three artefacts: templates, a template engine and some model. Examples of a model, a template, and the corresponding generated output are presented in figure 2.1.

The templates are static text documents, which contain dynamic parts [13][7, p. 29][15, p. 13]. In the example template of figure 2.1, static parts are written in white font. The static parts represent directly the generated object code, so they are generated directly as they are without any modification. The dynamic parts are metacode which are calculated and replaced run-time by the template engine [13]. The template engine evaluates the dynamic portions based the model [7, p. 29].

The dynamic parts of templates are often some template language, which the template engine can process. Often the dynamic parts are separated from static parts with some control characters. In figure 2.1 the dynamic parts are escaped with surrounding double curly brackets. The dynamic parts can define how data from the model is accessed. In addition to data access, control flow statements such as conditionals and loops are supported in most template languages. The template in figure 2.1 defines an example loop with `#each` keyword. The template engine might also support defining custom methods that are invoked in the templates. [7, p. 30] Thus, the templates can also define how the input data should be processed.

The benefit of using templates is that they can generate text in any language – even in natural languages. Templates are commonly used in instantiating HTML-pages in web applications [15, p. 12][7, p. 29]. Also, non-code files such as contracts and emails can be generated with templates [15, p. 13]. This flexibility is a great advantage when there are multiple different object languages. For example, a generated software project might have configuration files written in another language than the main programming language.

The template-based generator approach offers an improved separation of concerns, but it is still not perfect for every scenario. In literature, the structure of template-based code generators is compared to the web applications' structure that follows model-view-controller (MVC) architecture [15, p. 35]. In both types of programs, templates are used to implement a view to internal data, which is the model. The separation between boilerplate code and object code patterns makes template-based generators easy to understand [15, p. 14]. Similarly to MVC views, the templates should only present data and not alter the model or perform complex calculations [15, p. 35]. However, because the template languages have powerful features like invoking methods, the output description gets mixed with the generator logic in template-based generators [7, p. 30][17]

Templatization means extracting existing infrastructure as templates. It can be used as a technique to implement template-based code generators. The technique can also be applied if there is no existing infrastructure. When applying the templatization approach for such projects, a functional prototype of the generated application is first developed. The templates of the generator are then created from the prototype code. [18] When developing a generator, it must be validated that the generated code works as intended. With the templatization technique, part of the validating is already done when the infrastructure or the prototype is built. Sometimes it might be worthwhile to develop a new prototype even if an infrastructure already exists [18]. The existing infrastructure might be outdated or contain bad design solutions.

In larger generators, the template-based approach might cause extra complexity to the implementation. Usually, there is one template per generated file. If some parts of the input model are needed in multiple templates, the same data must be processed separately for each file. [7, p. 31]

The section 3.2.3 lists two tools that apply template-based code generation: Vue CLI and Angular CLI. The tools do small-scale code generation, from a few lines of code to several code files. For building larger generators, there are more advanced techniques such as abstract syntax trees or rule-based transformation.

2.3.4 Advanced techniques

The generators based on string concatenation or template-based approach do not usually support any syntax checks for the object code [7, p. 16]. Syntax safety allows finding errors earlier in the generation process before any object code is created. A generator is syntax-safe if it is guaranteed that the object code contains only valid sentences of the object language. [7, p. 3]

Two different techniques that support syntax checks appeared in the literature: abstract syntax trees and rule-based transformation. For strongly typed languages, it is possible to guarantee syntax safety by forming an abstract syntax tree representation of the object code. The transform from the abstract syntax tree to object code is done via an unparser. [15, pp. 6-7] This approach remotely resembles the way compilers are designed.

Another technique to achieve syntax-safety in code generators is called term-rewriting [15, p. 7]. This technique relies on transformation rules, which describe how each element in the model is a transformed element of the object language. Such transformations can be chained to form a code generation. The last link of the chain is a model-to-text transformation, in which the concrete object code is created. [7, p. 31] The terms in term-rewriting can be based on abstract syntax trees [15, p. 9]. The transformation rules are one-way operations, as only the transformations from the model towards the object language are allowed and not vice versa [15, p 7].

Both techniques, abstract syntax trees and term-rewriting, rely on creating an abstract representation of the object code. The benefit from instantiating an intermediate abstract model is that it is still available after the code generation for other transformations and processing steps. [7, p. 31] On the other hand, the abstract representation causes extra complexity for the generator, making maintaining the generator more challenging. It can be difficult to understand the abstract representation and to make changes to it. The grammar of the object language must be known detailedly in order to be able to create abstract syntax trees from it. [15, p. 16].

2.4 Code generation in project management perspective

2.4.1 Possibilities and risks

The software industry is globalized and highly competitive. At the same time, personal experiences with mobile devices, web applications, and other software raise people's expectations regarding software. To match the competition, companies need to improve time-to-market and reduce development costs continuously without compromising quality. [19] Code generation has much potential to increase competitiveness as it can improve both quality and productivity [20].

Code generating can improve code quality. There tend to be inconsistencies in large volumes of handwritten code, as programmers often find better approaches as they work [14, ch. 1]. If an active code generator is used, the existing generated code can be quickly replaced with improved code. Programmers also have their programming styles, which also decreases the consistency of a codebase. Similarly to automatic code formatters and style guides, code generating can be seen as one tool to improve code consistency because the output from code generators is consistent regardless of the developer running it [14, ch. 1]. The more handwritten code there is, the more probable it is that human errors are included in it. In case there is a defect in the generator, it needs to be fixed only once, and the fix will be constantly applied to all code generated in the future [14, ch. 1].

Code generators can free developers from doing repetitive tasks and therefore save time [17][14, ch. 1]. In projects applying model-driven development, productivity gains have varied from 27 % loss to 800 % gain. Most companies have reported productivity experiences between 20 to 30 %. [11] The benefits of small generators are less researched. It is unlikely that a simple coding helper could provide the aforementioned gains on the project level. However, it should be noted that also the smaller productivity gains add up. When a code generator handles the repetitive work, programmers can focus on challenging programming tasks and design. Faster development time also improves agility: issues are encountered earlier, and the software is easier to change later on. [14, ch. 1] As mentioned earlier, on the company level the productivity gains from code generation can improve time-to-market, which in turn increases competitiveness [21].

The productivity claims should be considered critically. Even if there was an eight times improvement in productivity, programming is only one part of software projects [21]. Domains, technologies, and generators' sizes vary between projects, which all may affect the experienced productivity gains. The productivity increase from code generation alone is not considered significant enough to adopt model-driven development methods in the industry. Instead, the biggest benefit from model-driven development is considered to be its support in documenting software architectures. [11]

The higher abstraction level of model-driven development is beneficial also from the project management perspective. High-level business rules can be more easily seen from a domain-specific code than from the actual implementation [10, ch. 2][14, ch. 1]. The models can function as a bridge between the development team, managers and domain experts [11][10, ch. 2]. If also domain experts can understand a model written in a domain-specific language, they can spot errors from it and communicate more efficiently with programmers [10, ch. 2]. It is also easier to translate existing logic into another programming language if a higher-level model of the logic is available [14, ch. 1].

Even though model-driven development has contributed much to improve abstraction and automation in software industry, it still faces major challenges. Research states that it is costly and difficult to define domain specific languages [12]. Model-driven development does not scale well [12][22]. There is also lack of robust tooling [22].

Generators are software, so all the risks involved in developing or purchasing software concern also code generators. There are numerous risks in a typical software project that need to be addressed. Loss of key personnel, ambiguous requirements and uncontrolled growth of scope are some of the most severe and common risks when developing new software [23]. The generator's development should go through the same processes used for any software projects to address the risks [14, ch. 1]. The level of risks depends on the complexity and size of the project [23]. Therefore, formal processes are not necessarily required with smaller generators.

2.4.2 Buying or creating a code generator

There are three different approaches to obtain a code generator: building one from scratch, using an open-source solution or paying for a commercial product [14, ch. 1].

The cost of developing software is high. The costs should be carefully weighed when deciding whether to build a custom solution. Resources are needed for both developing and maintaining generators. The costs of maintaining are often ignored [14, ch. 1][24]. However, the experience has revealed that maintenance typically consumes more than 60 % of resources in a software project [24]. With commercial and open-source generators, the community or company that has created the software often maintains it. On the other hand, it is up to the community or company how quickly defects gets fixed.

By building a custom solution, it is possible to control the evolution of the tool fully. The off-the-shelf generators might not suit well to the project context. For example, they might contain unfavored design solutions or integrate poorly into the development environment [14, ch. 1]. An open-source solution can be customized, although it can be laborious. The implementation of the generator needs to be studied in order to be able to make changes to it.

The commercial and open-source generators are often more mature than custom solutions because the user base is not limited to one company. The likelihood of new defects lessens when more users are using the generator in different use cases [25, ch. 2]. The larger the community using the tool is, the more information on using the tool is available.

3. AUTOMATING SINGLE-PAGE APPLICATION DEVELOPMENT

3.1 Web API standards

Single-page applications are often developed into a single HTML document that updates itself dynamically using JavaScript. Most of the user-interface logic is executed in the client [26]. The client communicates with the server via an application programming interface (API). Single-page applications resemble how mobile and desktop applications are designed: the user interface runs locally on the device's browser and queries only needed data from the back-end. This design allows having different kinds of applications consuming the same back-end. When developing new software products, it is often required that several different platforms are supported [27].

The clients are often developed by different teams and companies than the web services, so maintaining a consistent interface between them is challenging [27]. A change in an API must be reflected in each client using it, which causes maintenance overhead.

Different specifications and standards have been implemented to communicate and describe Web APIs in a language-agnostic way. The specifications are designed so that both humans and computers could easily understand web services without knowing about the implementation details. [28] In other words, the specifications are tools for describing web services on a higher abstraction level.

Simple Object Access Protocol (SOAP) and Representational State Transfer (REST) are the most dominant design models for APIs. All aspects of SOAP messages can be defined with Web Service Definition Language (WSDL). [29] Corresponding widely known standard for describing REST APIs is called OpenAPI specification [29][28].

WSDL documents are written in XML [29]. OpenAPI documents use either YAML or JSON format [28]. The WSDL and OpenAPI documents can define the endpoints of a service, their input and output data models, possible error codes and required authorization rights. [28]

In model-driven development, a higher-level abstraction – which is the model – is translated into a lower-level representation. The OpenAPI and WSDL specifications are machine-readable. Therefore they can be used as a model for generating the client and server-side code. By using an OpenAPI or WSDL based code generator, it is possible to instantly update all clients when the API of web service has changed.

Implementing back-end communication code is not the only part of single-page application development that has been automated. For example, toolchains and command-line interfaces are used to automate installing and configuring third-party tooling in single-page application projects.

3.2 Toolchains and command-line interfaces

3.2.1 Toolchains

Various third-party components and tools are often used in single-page application projects [30]. React itself is a relatively small library [31, ch. 1][30]. Third-party tools are required in order to be able to scale projects to many files and components, include third-party libraries, doing static code analysis for detecting mistakes, supporting live editing and optimizing the code for production [30].

Due to the extensive use of third party tooling, the learning curve to get into a new single-page application project can be steep. Multiple different design solutions must be made when configuring a single-page application from scratch: which tools and libraries to include, how to build the application and which language features to use. If each project uses a different set of tools and libraries, the developers cannot know what to expect when joining an ongoing project. The projects' idiosyncrasies confuse developers and decrease productivity in the beginning. [32, ch. 1]

As a solution to this, several integrated JavaScript toolchains have been developed. A JavaScript toolchain typically consists of a preconfigured package manager, bundler and compiler [30]. A popular toolchain for React applications can be used with a tool called Create React App [30]. Correspondingly, other popular single-page page applications also have their integrated toolchains. Vue CLI can create new Vue applications with standardized tooling [33]. Angular CLI is a similar program used in Angular development [34].

3.2.2 Create React App

Create React App is a tool that allows creating React application templates with standardized structure and default toolchain [30]. The tool outputs a static file structure for new React projects and installs third-party libraries. The tool supports also defining custom templates, which consist of the outputted static files and define the installed libraries. The projects initialized with Create React App use Node Package Manager (NPM) for managing the used third-party packages. NPM lists all third-dependencies with their version numbers inside a `package.json` file. [31, ch 1.]

Create React App uses a standardized toolchain in the projects it creates. Instead of listing every tool and library of the toolchain separately in the `package.json` file, the whole toolchain is encapsulated behind a single react-scripts dependency. As the toolchain is included as a third-party library, the developers cannot modify the internal configurations. The limitation ensures that all Create React App project share a similar toolchain and the developers familiar with it need less time to study the infrastructure. [32]

As the toolchain is encapsulated behind a single dependency, it is easy to update it: the whole toolchain can be updated by simply increasing the version number of the react-scripts dependency from the `package.json` file. [32] The maintainers of the react-scripts library update its internal libraries and test that there are no conflicts before releasing new versions.

In addition to the react-scripts dependency, the Create React App templates can define other dependencies installed to the created projects. The versions are defined using NPM version syntax, which allows defining version ranges in addition to exact versions [35]. For example, if a project is configured to use version `^1.0.4` of a library, any minor version newer than `1.0.4` such as `1.0.5` or `1.3.21` is allowed to be installed. Create React App will install the latest version allowed, which reduces the need to upgrade the libraries manually. On the other hand, there is a risk that newer versions of the dependencies conflict with each other.

One central feature of Create React App is called ejecting. When an application created with Create React App is ejected, the react-scripts dependency is replaced with all the actual dependencies, and the actual configuration files are added to the project [36][32]. After ejecting, the responsibility of maintaining the react-scripts dependencies moves to the developer. The action cannot be reverted [36]. Ejecting is often used when more control over either the dependencies or configurations is required [32]. It is recommended to avoid ejecting if it is possible [36]. Previously configuring proxies or using Typescript required ejecting, but the support for configuring them without ejecting was later added into Create React App [32].

Create React App does not utilize code generation. Almost all outputted files are copied directly from the template without performing any processing to them [36].

3.2.3 Vue CLI and Angular CLI

Vue CLI and Angular CLI are command-line interfaces used in Vue and Angular development. They contain advanced functionality in addition to providing a standardized toolchain. User can interactively pick included features when creating a new project with Vue CLI. The tool is developed using a plugin-based architecture. The plugins define which dependencies are installed, modify internal bundler configuration, add and edit code files and inject new Vue CLI commands. Creating new files and editing existing files can be done by defining code generators. [33]

Vue CLI plugins use template-based code generation. The templates are written in EJS templating language. The template files contain a YAML front-matter, where certain settings such as output path for the generated code can be defined. The model for the generator is created in run-time with command-line prompts that are shown to the user. Each plugin can define its prompts. Vue CLI uses the Inquirer.js library for showing the prompts. [33]

Similarly to Vue CLI also Angular CLI utilizes template-based code generation [34]. The generators are called schematics. Schematics support complex logic and can both add and modify files. [37] The Angular Schematics tooling runs all file operations inside a virtual file system before transforming the actual code files. This design allows reverting all changes in case errors occur [38].

Angular CLI comes with a basic set of schematics. For example, boilerplates for new classes, components and services can be generated with the build-in schematics [34]. Third-party schematics are also supported. Third-party library maintainers can write schematics to automate using their libraries in Angular environments. There are three types of schematics: add, generate and update. Add schematics are used for installing libraries and creating configurations: for example, command `ng add @angular/material` installs the Angular Material library and configures theming for it. The generate schematics are used to generate library artefacts. An Angular Material table with preconfigured paginating can be created by running `ng generate @angular/material:table` command. The update schematics can update the library's dependencies and adjust the project for breaking changes. [37] The add schematics are similar to Vue CLI plugins. Based on the documentation, it appears that Vue CLI does not have corresponding functionality to the generate schematics of Angular CLI.

Similarly to Vue CLI plugins, Angular schematics can query input from the user by prompts. The user does not need to have in-depth knowledge of an Angular schematic or a Vue plugin to use them, as the required data is queried interactively. [38]

The generators developed for Vue CLI and Angular CLI are platform-specific: a plugin developed for Vue CLI cannot be used in React or Angular development. This design makes it possible to have technology-specific abstractions but limits the use-cases of a single generator. For example, Vue projects typically have an entry file called `main.ts` or `main.js` depending if the project is written in JavaScript or TypeScript. The Vue CLI generator API provides an abstraction of the entry file, which the developer can use without knowing the actual name of the entry file.

Vue CLI and Angular CLI tackle similar issues with the generators developed in this thesis, although they cannot be used directly in React development. Vue CLI and Angular CLI can be seen as ecosystems for publishing and installing generators. The command-line interface platforms provide required utilities and constraints – such as utilities for showing prompts – for creating generators that offer a consistent user experience. As generators can be more easily shared, a single generator can have much more users worldwide. For example, one of the most popular Vue CLI plugins, vue-cli-plugin-vuetify, has over 150 000 downloads per week [39]. Angular CLI documentation states that schematics are typically written directly by the library authors [37]. Schematics can ease library usage in an Angular environment and, therefore, make the library more attractive among Angular developers.

4. REQUIREMENT ANALYSIS AND DESIGN

4.1 Defining scope for the generators

At the end of chapter 2, it was stated that one of the most significant risks in software projects is the uncontrolled growth of scope. Given that the generator tools are done as part of a thesis, it is reasonable to focus on smaller-scale code generation. The tools developed in the thesis are created for industrial use and are not research prototypes. Generating large, fully functional applications that meet custom requirements is complex and could not be accomplished within this thesis's scope. However, it is possible to develop new tools or improve existing tooling later if small-scale code generation provides good experiences.

It is also redundant to focus on too simple generators. Code editors and integrated development environments already support defining code snippets that can be injected into a code file. For example, in popular code editor Visual Studio Code, the autocomplete feature will suggest the user-defined snippets when writing code. The snippets can contain placeholders that are filled each time the snippet is injected into the code. [40] Repeating code structures like logging and method declarations can be quickly created with the snippets. As this feature is already well integrated into the coding environment, there is no need to implement a similar custom solution.

Given the observations above, this thesis focuses on generating components that consist of a few files. The potential candidates for single-page application components that could be generated are identified in the following section based on investigating existing React projects and interviews with their developers.

4.2 Defining the requirements for the generator

The generators' requirements were mainly based on the author's and Wapice's developers' experiences of React development. As mentioned at the end of section 2.4.1, ambiguous requirements are a significant risk in software engineering. The section also discussed that the level of the risk depends on the complexity of the developed generator. The viewpoints of different developers and projects should be considered when developing smaller generators. The more the code generator is used, the more the productivity and quality gains add up. Also, if more benefits are gained from the generator, more resources can be targeted for further maintenance and development. If the tool is not maintained, it will quickly get outdated as front-end technologies evolve rapidly.

The first requirement stated that the generator tool is used alongside a static React application template. In other words, it should be possible to use the React application template also without the need to use any code generator. Separating the React application template from the generators reduces the risks: if it turned out that code generation was not a suitable solution for the research problem described in chapter 1, the React application template could still be used. In practice, the code needs to be generated additively on top of the React application template to fulfil the requirement. From the maintenance point of view, this design might provide a better separation of concerns, as the generators are separated from the static code.

One requirement for the generator tools was that they should be easy to maintain. Web technologies are evolving rapidly. If maintenance is complex, it is avoided. It is expected that at least some maintenance work for the tool is required. Even the static React application template must be updated occasionally to keep it up-to-date with evolving front-end technologies. Maintaining the tool should not be much more complicated than maintaining the static template.

The other requirements for the generator tools and the React application template was gathered in a two-stage process. First, the source codes of existing React applications and one old React application template developed in Wapice was investigated. A total of nine different React projects were included in the analysis to consider developers' and projects' different needs more broadly. The sizes of the projects varied from 105 files to 958 files. Also, projects developed for internal use were included. As the applications' business logic is confidential information, it was excluded from the analysis.

A suitable architecture for a new React application template was derived from the existing applications. In practice, this meant seeking the most common architecture patterns used in the projects. In addition to the basic architecture, also commonly appeared components were sought. The repeatedly present components were considered good elements to be included in the static template project. The existing React application template developed a few years ago was also included in the research. However, a new template was developed in this thesis, as the existing React application template was considered outdated.

Defining the components that were good candidates for code generation was a more challenging task. The generated components should be modular and encapsulate a feature. When the user generates a specific component, it should be clear what is included in the outputted code. Otherwise, the developer experience from using the generator tool would be compromised. The developer would need to test generating the components manually and inspect what has been generated. It should also be possible to generate the components independently from each other, regardless of whether other components have already been generated. One exception to this are components that overlap entirely with each other. These kinds of components could be, for example, different types of authentication. If basic authentication has already been generated to the application, generating authentication for some other authentication standard may conflict with the generated code.

The components should be both easy to generate and useful for developers using the generators. The initial plan was to search for components that appeared in several projects but not in all. As the research material was relatively limited, there was a risk of sampling bias if only the source code findings were used. The author was unfamiliar with the project's domain areas, making it more challenging to understand the code. By only reading the code, it was unclear if the React projects' design solutions had later turned out to be undesirable. The existing React application template could have affected the projects if they were built on top of the template.

Based on the challenges faced in inspecting the source codes, it was decided that a few front-end developers should be interviewed to gain a more subjective viewpoint. The interviews formed the second stage of the requirement analysis process. Four developers were interviewed. One of them was the developer of the original React application template. The front-end developers are the end-users for the generator, so their preferences and opinions should be carefully considered when designing the tool. The goal of the interviews was to gain insight into what design solutions of the projects they have found to be functional and what has turned out to be problematic.

The following questions were asked from the developers:

1. What design solutions (used components, tool or architecture etc.) of the React projects you have been involved in have turned out to work well?
2. What design solutions have turned out to be problematic?
3. What kind of experiences you have had with the existing React application template?
4. What features you would like to have included in the new React application template?
5. What features would you like to include in the generator?

Before asking the questions from the developers, the goals of this thesis were explained. The interviews were kept short, about half an hour per developer, to avoid disturbing the developers. The interviews followed a semi-structured approach: the questions founded a baseline for the interview, but specifying questions were asked based on the developers' answers. It was beneficial to inspecting the source codes beforehand, as the used components and structure had become familiar before the discussions.

Eventually, the template's and generators' requirements were determined based on both the interviews and reading the source code. Many requirements of the template could be determined directly from the source code. The applications used similar architecture patterns.

Every project used Redux library for state management. Redux is a popular library that provides a state management layer for front-end applications. The whole application's state is stored in a centralized store. The state can be modified only by dispatching Redux actions and handling the actions in Redux reducers. Redux reducers are JavaScript functions that take the existing application state and the dispatched action as parameters and return a new application state. Redux is independent of React and can also be used with other front-end technologies such as Angular. [2, ch. 1]

Redux-Saga was used widely in the projects to implement asynchronous actions. Often, additional logic needs to be executed when updating the application state. For instance, the application might send a request to the back-end and update the state based on the response. The additional logic is called a side-effect. If a side-effect contains asynchronous operations, such as sending requests to back-end APIs, it is invoked by an asynchronous action. [2, ch. 4] Redux-Saga supports defining complex asynchronous operations and chaining them [2, ch. 6].

The way the files were organized within the projects varied. In most projects, the application was divided into *features*. The approach is common in React development. There is no exact definition for the term feature, but it represents a major part of the application [41]. A single feature has its own folder, which contains all the related code. The other way to structure files was to do it directly by file type, which is also a popular way of organizing React projects [41]. In the latter approach, similar files – such as the files defining React components – are stored in the same folder. In all projects, the code related to back-end communication was separated into a separate folder.

The projects used many technologies that are commonly adopted in React development in the last years. All projects were written in Typescript. Build tools like Webpack that are configured to the Create React App tool appeared widely in the projects.

In some of the projects, the files related to back-end communication were generated using a tool called OpenAPI Generator. The files contained comments mentioning that the files were generated and should not be modified manually. The generated files' code consisted of Typescript type definitions and logic for sending HTTP requests to the back-end. It was decided that more information about experiences from the OpenAPI Generator should be gained from the interviews, as it was a potential tool to utilize also in this thesis.

In the interviews, more requirements for the template and the generator tools were defined. The possible candidates for the generated components were also identified mainly based on the interviews.

Two developers mentioned that they were not using the existing template when the question 3 was asked. They mentioned that the tools and the way React code is written had changed a lot since the template was implemented. However, the developers had acknowledged the considerable work of creating React projects from scratch. Instead of using the existing template, the developers used to copy parts from previous projects and refining them in subsequent projects.

One problem that also came up with the question 3 was a specific issue related to the React template's maintenance. The issue came up in two interviews. The origin of the issue was that the existing template is based on an ejected Create React App project. In section 3.2.2 it was described that in ejected Create React App projects, the single react-script dependency is replaced with the actual dependencies. In ejected projects, the different dependencies replacing react-scripts need to be manually updated to keep the project up-to-date. As there are many dependencies and the different versions of dependencies might conflict with each other, it is laborious work. It was desired that the new template was not ejected and could directly make use of Create React App dependency management. The use of react-scripts dependency was defined as a requirement for the template.

When discussing about questions 1 and 4 all developers mentioned handling the asynchronous operations with Redux Saga had worked out well. One common alternative option to handling the asynchronous operation is to use Redux Thunk [2, ch. 4]. However, as developers had more experience from Redux-Saga, it was decided that it is included in the template.

All the developers highlighted the excellent experiences they had had with the OpenAPI generator when the question 5 was discussed. One developer mentioned that by generating the code related to back-end communication, a considerable amount of time had been saved in the projects. Using the tool has also made maintaining the projects easier, as the code can be regenerated in case changes are introduced to the API of the back-end. One of the developers mentioned that he had to do some minor configuration to get the generated code to function. As commonly used back-end frameworks like Spring and ASP.NET Core both had tooling available for generating an OpenAPI specification based on the back-end implementation, the commonly used process was to generate the specification from the back-end and then generate front-end code based on the specification. As the experiences from generating API related code were positive, it was decided that generating back-end communication code is included in the thesis project.

The idea of generating authentication-related code based on some authentication standard like OpenID Connect also came up in one discussion. As authentication requirements vary between projects, authentication logic cannot be directly implemented in the template. However, it might save time if the authentication logic could be generated when required. In addition to writing the configurations and components, the developer needed to install a related client library. Because most of the logic is defined by the authentication protocol, the implementation is always almost similar. Thus, it was decided that generating of OpenID Connect authentication is implemented. The desired functionality was similar to Vue CLI plugins or add schematic of Angular CLI: the generator would install the library and create necessary configurations and components.

Other presented ideas were generating configurations for application performance monitoring and generating repetitive code such as Sagas. Also, it was hoped that the generator tool could be extended for project-specific needs. One developer mentioned that although the code related to back-end communication could be generated, the Sagas invoking the generated methods must still be written by hand. The idea to generate them was discussed.

When generating definitions for Redux-Saga was further considered, it was noticed that they did not fit well into the defined scope. By reading the source code of the existing React application, it was observed that typical Saga definitions consisted only of a few lines. Therefore they could have been an excellent candidate to be created with the code snippet feature of Visual Studio Code [40].

As mentioned, most of the existing React projects had their files organized by feature under a folder named `features/`. In addition to the Sagas, each feature typically consisted of Redux reducers, Redux actions and React components. The different file types were each in a separate folder so that, for example, files defining reducers located under `reducers/` folder and actions in `actions/` folder. The contents of similar files shared a similar structure: for example, files defining reducers contained the reducer function, its related selectors and TypeScript type definitions. The original idea of generating Sagas was extended to generating boilerplate for other files types. The boilerplate generator would decrease the required time for adding new features, as the developer could immediately start to implement the actual business logic instead of creating the folder structure and writing the boilerplate code manually.

The generating of boilerplate and file structure for new features was different from generating the authentication configuration, as new features are added throughout the project, whereas the authentication configuration is intended to be generated at most once in a project. In other words, the boilerplate generator is similar to the generate schematics of Angular CLI. On the other hand, the generated authentication configuration does not necessarily require any modification, whereas the generated boilerplate code requires implementing the actual business logic. It was expected that the different types of code generation would have provided more insight into the research question 2 about the parts of single-page applications that could benefit most from code generation.

To conclude the requirement analysis, the following requirements were set for the developed static template project and the generator tools:

1. The template should be separate from the code generation; it should be possible to use the template without using any generators.
2. The generator tools should be easy to maintain.
3. The template should use the single react-scripts dependency for easier maintainability.
4. The template should contain Redux with Redux-Saga already configured.
5. The generated components should be independent so that generating one component does not conflict with others unless the components are fully overlapping.

Based on the interviews the generating of following components were planned to be implemented:

1. Authentication configuration based on OpenID Connect specification.
2. Generating of back-end communication code based on OpenAPI specification.
3. Generating the folder structure and boilerplate code for Redux reducers, Redux actions, sagas and React components of new features.

4.3 Choosing tools

The tools for the implementation were chosen based on the scope defined in section 4.1 and the requirements defined in section 4.2. Ready-to-use tools were preferred over self-made tools, as easy maintainability was defined in requirement 2.

There were two possible ready-to-use alternatives for generating communication to the back-end: OpenAPI Generator and Swagger Codegen. The OpenAPI Generator is based on Swagger Codegen, but its development is driven by an open-source community instead of SmartBear Software company [42]. Both tools utilize template-based code generation, and their template engine is developed with Java. For the OpenAPI Generator, there is a popular NPM wrapper package, which allows invoking the tool with NPM [43]. As the NPM wrapper allows easier integration to React development environment and the OpenAPI tool was also already familiar among Wapice's developers, it was chosen to be used in this thesis project.

The OpenAPI Generator allows generating both server and client code based on OpenAPI specification. It has generators for almost 100 different languages, and technologies [42]. The typescript-fetch generator was chosen from the available options as both technologies – TypeScript and JavaScript Fetch [44] – were commonly utilized in back-end communication-related code of the investigated projects.

The OpenAPI Generator is an active generator. The files it generates have comments describing they should not be manually modified. When a change is introduced to an OpenAPI specification, the generator is rerun, and the existing files are rewritten based on the new version of the specification. In case any manual modifications have been made to the existing code, the modifications are lost.

For the other two generated components, there were no ready-to-use tools available. Instead, generator tools that could be customized were searched. As the thesis's scope was restricted, tools based on advanced techniques presented in section 2.3.4 were excluded. Maintaining these types of code generators was considered too challenging. There were two possible alternatives left: tools based on the string concatenation and the template-based approaches. The tools using a template-based approach were preferred, as the approach does not have the issue related to the complexity of escaping special characters.

The template-based approach was considered to be suitable for smaller-scale code generation. If the generated components consist of few files and one template represents one output file, only a few templates are needed per generated component. The template languages provide enough flexibility without being too complicated.

The next step was to determine whether to build the tool or use an existing open-source or commercial tool. By investigating possible open-source solutions, it seemed that there were a few options available. As discussed in chapter 2.4.2, implementing software and maintaining it is expensive and has multiple risks. Thus, it was considered reasonable to build the tool on top of an open-source solution. For small scale code generation, no suitable commercial product was found.

Although both Vue CLI and Angular CLI are open-source and utilize template-based code generation, they could not be utilized in React projects. Vue CLI supports only Vue projects, and correspondingly Angular CLI can be used only in Angular projects.

The three most potential open-source solutions that filled the requirements were compared to find the most suitable tool: Yeoman generator, Plop and Hygen. All the generators are developed to Node environment [45][46][47]. As the NPM (Node Package Manager) is used widely in React development, there is no need to install additional development tools to use the generators [31]. As the generators use template-based generating, they support generating any language.

Yeoman provides a generator ecosystem with predefined architecture and tooling. There are 9,444 different generators developed for it varying from licence generators to generators that generate whole full-stack applications. New generators are implemented with JavaScript. [45] Implementing new generators for the tool is well documented.

However, as Yeoman is designed to work as an ecosystem rather than a simple coding helper, it is more complex to implement generators for it than for the other two tools. New generators are implemented by extending `Generator` class and writing logic for reading templates, processing input data and writing output-files [45]. The class extending the `Generator` represents the template engine of the generator. As it is not limited to what kind of logic is implemented in the template engine, Yeoman is suitable for very different scenarios. On the other hand, if the generating consists mainly of reading input data to be injected into dynamic parts of the templates, the implementation may seem relatively complex.

The Plop and Hygen are closer to simpler coding helpers than a generator ecosystem. The Plop web page describes Plop as a "micro-generator framework". The Plop generators consist of two parts, prompts and actions. The prompts define input data that is asked from the user before running the generator. The actions define the used template files and the output paths. Plop generators collect user input via prompts and replace the templates' dynamic parts with inputted values. Plop's approach is more restrictive than Yeoman's one. However, the restrictions make implementing small generators simpler. The web site of Plop emphasizes that it is easy to learn and use. The templates of Plop use Handlebars templating language. New generators can be defined by creating a `plopfile.js` within the code project. As these files reside within the project, they can be shared with the developers of the project. [47]

The Hygen tool shares many of its design concepts with Plop. For instance, it uses prompts similarly to Plop. With Hygen, the generators are also implemented within the project inside a `_templates/` folder. The templates use EJS templating language. Template's actions are defined within the template's header. It is also possible to invoke scrips from the headers of the templates. This feature can be used, for example, to install libraries when the generator is run. Another attribute that the Hygen focuses on is performance. According to its web site, it is constantly benchmarked to decrease startup and generation time. [46]

Instead of a single file, each Hygen generator has its folder inside the `_templates/` folder. The folder structure directly defines the commands in which the generators are invoked. This design solution decreases the needed amount of configuration to create or edit generators. The website of Hygen states that this approach supports larger teams [46]. The Plop's way of storing all generators in a single `plopfile.js` can cause conflicts in version control when multiple developers have edited it.

From these tools, Hygen was tried first. Hygen was considered to be easy to use and maintain. Its generators can run scripts, it supports larger teams, and it provided good performance. It was stated in the requirement 2 that the tool should be easy to maintain. The approaches of Plop and Hygen were considered to fit well with the requirement. As the actions defined in those tools both restricts and steers how the tool is used, there is

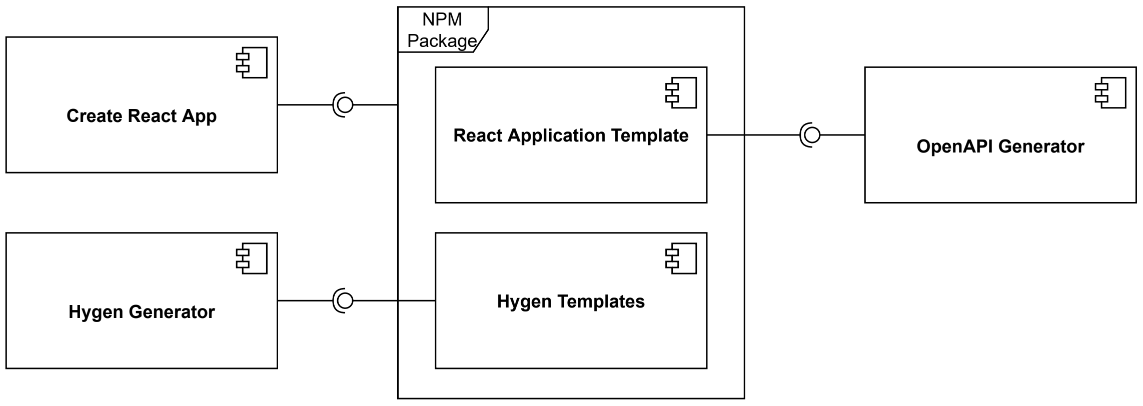


Figure 4.1. *High level architecture presentation of the tool*

less risk that the tool itself becomes a software product of its own that requires maintenance [46]. The Hygen's support for running scripts in its actions was considered advantageous. The functionality was intended to be used for installing a related client library when generating the OpenID Connect authentication. Hygen and Plop used prompts to interact with the user, which resembles the design of Angular CLI and Vue CLI generators. Vue CLI and Plop uses the same Inquirer library for showing the prompts [33][47].

However, Hygen was later replaced with Plop due to technical challenges encountered during the implementation. Hygen had a defect that prevented it from modifying files in Windows environments.

Hygen and Plop can both be used for both active and passive code generation. By default, the user is prompted before overwriting an existing file to avoid losing any modifications. This behaviour can be overwritten with local and global configuration [46]. In this thesis, Hygen was planned to be used as a passive generator because it makes it easier to modify the files afterwards. When generating features like authentication and boilerplate code for reducers, it is more important to modify the existing code than to be able to rerun the generator.

4.4 Architecture

The original high-level architecture of the template and used tools is presented in the figure 4.1. The React application template was designed to be implemented as a custom template for Create React App to fill template requirement 3. This way, Create React App's dependency management was utilized, thus making maintaining the template easier. As Create React App templates are NPM packages, storing the templates in a private package registry is possible. The template's users could create new projects based on the latest version by simply invoking `npx create-react-app` command.

Two different generator tools – Hygen and OpenAPI Generator – are used alongside the template. The usage of the tools differs. Hygen is used to passively generate boilerplate code that is extended and modified later, whereas OpenAPI Generator is used actively to keep up with external API's changes. During the implementation, Hygen was replaced with Plop due to technical challenges encountered with Hygen.

It was planned that the Hygen files should reside inside the React application template to allow quickly generating components on top of the template. The requirement 1 stated that it should be possible to use the template without any code generation. Hygen supported well this design. When the base for a new project was created with the Create React App tool, the developers could choose whether to use the code generation features or not.

The use of OpenAPI Generator is considered in the design of the React application template. It should be straightforward to generate the back-end communication code to the template. Therefore a separate folder is reserved for the generated files. This design makes it easier to later distinguish between handwritten code and generated code. As OpenAPI Generator is an active generator, having its output in a separate folder lessens the likelihood of accidentally overwriting handwritten code. A script that invokes OpenAPI Generator with parameters is included in the React application template.

5. IMPLEMENTING CODE GENERATORS

5.1 Implementing the template

Both the generators and the usage of Create React App affected the file structure of the template. The template's core files and folders are presented in figure 5.1.

The folder structure has been derived from two projects: from the official Create React App TypeScript template and the existing React application template made by Wapice. The TypeScript template contains required configurations and tools to support the TypeScript language [36]. The implementation started by cloning the official TypeScript template source and copying the custom logic from the Wapice React application template to the copied TypeScript template.

The custom templates of Create React App must be named so that the name starts with `cra-template-` following by the template name [36]. The name appears in two locations: in the root file name and in `package.json`. When `npx create-react-app` command is invoked for example with `--template wapice` argument, Create React App looks for templates named as `cra-template-wapice`.

The files located on the root level are for the Create React App and not inserted directly into the created project. Instead, they are read by Create React App when creating new projects. As the template itself is an NPM package, there is a `package.json` file on the root level, which contains required metadata about the package. Instead, the dependencies in the `package.json` file are not included in projects created from the template. The dependencies of the created projects are defined in `template.json` file [36]. As mentioned in section 4.4, by defining the dependencies versions using version ranges, Create React App automatically installs the latest compatible versions of the libraries to the created projects.

The files from `template/` are outputted when the Create React App is run. The core structure of the application is visible under the `src/` folder. The `api/` folder contains the code related to back-end communication. Its contents can be entirely generated with OpenAPI Generator. In case OpenAPI Generator has been used, any handwritten code should not be added to the `api/` folder to avoid overwriting handwritten code.

The `features/` folder contains the project-specific code divided by features. The boil-

```
cra-template-wapice/
    |-- README.md
    |-- package.json
    |-- template/
    |   |-- README.md
    |   |-- gitignore
    |   |-- _templates/
    |   |-- public/
    |   '-- src/
    |       |-- api/
    |       |-- features/
    |       |-- framework/
    |       '-- index.tsx
    '-- template.json
```

Figure 5.1. *File structure of React application template*

erplate generator can generate folder structures under the `features/` folder. As the boilerplate generator is used as a passive generator, there can also be handwritten code among the generated boilerplate code.

The documentation was written in two `README.md` files. In the root level `README.md` it was documented how to start using and maintain the template. As the file is not inside the `template/` folder, it is not included in the projects created with Create React App. Therefore it was a natural place to write the instructions related to maintaining the template and the generators. In the `README.md` inside the `template/` folder it is instructed how to run and – if required – how to eject the project. Also, the suggested usages of the OpenAPI Generator and Hygen generator are instructed there.

The project-specific views, reducers, actions and sagas are intended to be implemented inside the `features/` folder. An example implementation is added to the folder to demonstrate the usage of the template and the generators. However, as the project-specific code is separated from the shared application code, the developers can delete the contents of the `features/` when they start to develop the actual application.

As there is no `package.json` file in the `template/` folder, the code cannot be directly run from there. In order to be able to test the code, it was written into a separate React application. The code was regularly copied from the React application to the Create React App template. The integration was tested by running Create React App with the template and running the created React application.

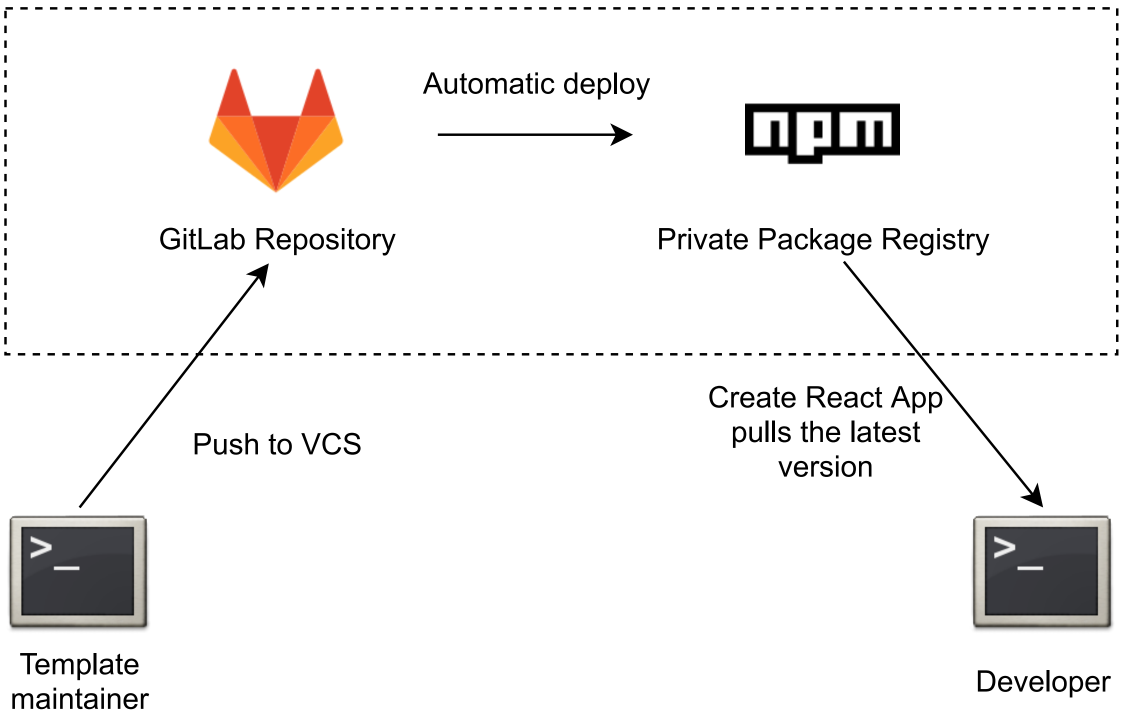


Figure 5.2. *A CI/CD pipeline designed for the template.*

5.2 Developing a CI/CD for the template

To reduce the work needed in maintaining the template, a CI/CD pipeline that automatically creates a new NPM package in a private registry when code is checked in the version control was also designed. GitLab was used as a platform for building and deploying an NPM package from the code.

The process for publishing newer versions of the template is present in the figure 5.2. When the maintainers of the template check their code to the version control system, the CI/CD pipeline is automatically triggered. The pipeline builds an NPM package from the source and deploys it to a private registry. If any errors occur during the build, the deployment is cancelled. Create React App uses the latest version of the package automatically when creating new projects.

GitLab NPM packages' names consist of a scope and a name of the package. The former is the group or user where the GitLab project belongs, starting with @-symbol. As the GitLab group was named `cra`, the name of the package hosted in GitLab NPM registry was `@cra/cra-template-wapice`. [48] The GitLab's package naming convention conflicted with the way Create React App templates were intended to be named. However, the Create React App had lately started to support also scoped packages [36]. For scoped packages, the full name consisting of the scope and the package name needed to be added as value to `--template` flag instead of the end of the name.

Scoped packages suit well private NPM registries. NPM supports pointing private registries to scopes [48]. In the thesis project, the `@cra` scope was pointed to the private GitLab registry. In practice, all packages that share the same scope are searched first from the pointed private registry.

5.3 Implementing code generation to the template

5.3.1 Using OpenAPI Generator

Adding code generation to the template started with setting up the OpenAPI Generator. A script invoking the OpenAPI Generator was included in the `template.json` file. The script ran the TypeScript-Fetch generator and defined `api/` as output directory. The only parameter not included in the script was the path to an OpenAPI document, as it was intended to be passed by the user.

After implementing the script, it was tested. The initial plan was to implement an example back-end service and to generate an OpenAPI document from it. However, with the approach, the generated example code could not be used if the users did not have access to the API. One option was to include the API implementation in the template, but alternatives approaches were also sought. Having back-end code amongst the front-end code might have confused developers. Instead, the back-end functionality was implemented using a mock-server library called json-server [49]. Another library called json-server-auth was used to mock authentication [50]. The OpenAPI Generator was tested by writing an OpenAPI document describing the mock server and running the generator against it. The generated code was left to the template as an example.

The TypeScript interfaces and Fetch request logic was generated within seconds on the first run. In the first run, the output used the syntax of TypeScript 2. As the template used TypeScript 4, the project could not be built successfully. By appending an additional parameter describing the preferred TypeScript version to the script invoking the generator, the version conflicts were solved, and the project built successfully.

The next step was to test the generated code. The mock API had an endpoint for reading and writing to-do notes. Based on a model defined in the OpenAPI specification, a `Todo.ts` file had been generated into the template. The generated `TodoApi.ts` file contained logic and interfaces for invoking the Todo endpoints.

The generated code was tested by calling a generated function `getTodos()` from the React application. The Todo model had a boolean attribute `done`, which represented whether a single to-do task was already done or not. The `getTodos()` endpoint supported filtering the returned to-do notes based on the attribute. This functionality was reflected also in the generated code: The generated function had a single parameter of type `GetTodosRequest` which contained the `done` attribute. It was possible to see how the API should be called by simply looking at the generated code.

The request did not succeed on the first try. By debugging the code, it was found out that a simple utility function handling binding of JavaScript's and TypeScript's `this` keyword had to be implemented in order to use the generated code. The typescript-fetch generator documentation was somewhat limited and did not provide any help in solving the problem. The documentation had no information about the intended use of the generated code, so instead, the usage had to be studied by directly reading the code. Fortunately, the generated code was clear and readable. As the generated code was left to the template, it served as an example of using the implemented utility function.

Although some issues were encountered, solving them did not require editing the generated code. Thus, it is possible to use the OpenAPI Generator as an active generator and rerun the generator when the generated code needs to be updated. The active code generation supports iterative development processes: when the OpenAPI model is extended or changed, the changes can be applied immediately to the front-end code. The typical use-case in Wapice was to generate the OpenAPI specification from back-end implementation. If the back-end is also under development, it is possible to immediately generate code to the front-end when the back-end has some available endpoints.

As the OpenAPI Generator itself did not require any custom code or modification, it does not have any maintenance overhead. The open-source community has updated the tool when for instance, newer versions of TypeScript have been published.

5.3.2 Implementing generating of authentication

Hygen generator was planned to be used for generating the other components. Unlike OpenAPI Generator, Hygen did not provide ready-made generators apart from a generator that outputs new Hygen generators.

To start using Hygen, it must be first globally installed with `npm i -g hygen` command. The `_templates/` folder can be created by running `hygen init self` command. [46] The created folder contains templates for the generator that can be used to generate other generators.

Hygen uses the concepts of generators and actions. A generator combines one or more actions. The actions, in turn, define the actual behaviour of the generator. A single action can consist of multiple template files. The hierarchical model is visible under the `_templates/` folder: the root level folders represent the generators, and the folders under the generators define the actions. The folder structure of `_templates/` also corresponds to the commands used to invoke the actions [46]. For instance, when invoking command `hygen oidc configuration` an action called *configuration* belonging to *oidc* generator is invoked. When an action is invoked, Hygen processes all template files located under

```
1 module.exports = [
2   {
3     type: 'input',
4     name: 'client_id',
5     message: 'Please enter a client id'
6   }
7 ]
```

Figure 5.3. *Hygen prompt.js file defining a single prompt.*

the action folder.

The only way Hygen can get any input data is by prompting it from the user. The prompting is done by adding a `prompt.js` file inside the action folder. There is no other way to pass input data on run time to the generator. Therefore, it is impossible to implement generators based on some complex model such as OpenAPI documentation with Hygen. The dynamic parts of Hygen templates are replaced with the model formed from the user input, similarly as in figure 2.1 represented in section 2.3.3.

The figure 5.3 shows an example `prompt.js` file which defines a single prompt. When the generator is run, the message defined on the fifth line is printed to the console. The user can then enter a value and proceed by pressing the enter key. The name of the variable where the value is stored is defined on the fourth line. When Hygen processes the templates, it looks for dynamic partitions that contain references to the `client_id` variable and replaces them with the value.

As the Hygen templates are written as EJS template language, the files must be generated in order to be able to test them. Prototypes of the generated components were implemented directly to a React project created from the React application template to make testing easier. The OpenID Connection prototype implementation was tested against two different identity providers: ASP.NET Core IdentityServer 4 and Auth0. The IdentityServer 4 is an open-source project built on ASP.NET Core framework [51]. The Auth0, on the other hand, is a commercial service [52]. Although both providers supported the OpenID Connect protocol, there were minor differences in the client configuration required by default. By testing against two different identity providers, it was possible to identify the configurations' differences and add them as dynamic parts in the generator templates.

A higher abstraction level from the configuration details was applied in the generator's prompts for easier usability. Instead of asking the user directly for such OpenID Connect attributes as `response_type` or `scope`, the users were asked which identity provider they were using, IdentityServer or Auth0.

Some of the configurations were application-specific rather than specific to the identity platform. For example, the client id attribute of the OpenID Connect protocol is a string used to identify the client in the authentication process. These kinds of configurations were also added as dynamic values to the generator templates. The values for those dynamic parts were asked directly from the user.

After the prototype implementation was functional, the Hygen templates were implemented based on it. The made changes were inspected with diff tool of Visual Studio Code and turned into EJS templates. Thus, the templates were created using the templatization technique described at the end of subsection 2.3.3. A total of nine template files were created in the process.

Two kinds of Hygen operations were used for the generator templates: add and inject. The add operations are used to generate new files. The added files contained a configuration file for oidc-client, a login button component and components for handling login and logout redirects from the identity provider. The inject operations are used to modify existing code, as the templates are injected within existing files. In the thesis project, the inject operations were used to include generated UI components into the existing code and configure callback routes for the authentication. One Hygen template can modify only one part of a file. Therefore, multiple templates were often required for modifying a single file.

When testing the Hygen generator, a significant issue was encountered. The injected templates were always inserted at the start or the end of the code files. Further research revealed that the community of Hygen tool had encountered the same problem when using Hygen in Windows environments. A related issue had been submitted to Hygen's GitHub issue tracker after the last release of Hygen. The last change to Hygen had been made a half year ago, so it seemed that the tool was no longer in active development. As Windows is commonly used as a development environment in Wapice, this was a blocking problem.

It was possible to continue the implementation by changing the used tool from Hygen to Plop. As mentioned earlier, these tools share many similar concepts such as the prompts, actions and templates, so there was no need to alter the higher-level design because of the change. Plop uses Inquirer.js instead of Enquirer.js for the prompts and Handlebars templating language instead of EJS. The prompts and the templates were rewritten with different syntax.

Hygen documentation stated that Hygen provided better support for larger teams than Plop because Hygen generators consist of multiple files. However, it was noticed that it is possible to use a similar design with Plop by splitting the `plopfile.js` into several JavaScript modules. As Plop did not provide direct support for running scripts, NPM modules could not be automatically installed when the generator was run. As a workaround, a message which instructed which package to install was printed. It was not a major shortcoming, but it still weakened the user experience.

```
features/
    '-- car/
        |-- sagas/
        |   '-- carSagas.ts
        |-- components/
        |   '-- carComponent.tsx
        |-- reducers/
        |   '-- carReducer.ts
        '-- actions/
            '-- carActions.ts
```

Figure 5.4. *An example of generated file structure.*

The implementation of OpenID Connect authentication was finished with Plop. The generated part was responsible for authenticating the client against an identity provider but did not include logic for using the retrieved token in further requests. The responsibility of using the token was left to the developer because he or she could either use OpenAPI generated code or manually written code for invoking the requests. The requirement 5 stated that the generated components should be independent so that the generated components do not depend on each other. In other words, the developers should be able to use the OpenId Connect authentication generator whether they have used the OpenAPI Generator or not.

Modifying existing files had a few challenges with both Hygen and Plop. Although the changes were often small, a separate action and template had to be defined for each modification. Plop used regular expressions for specifying the part which was modified. From the maintainability viewpoint, this was found out to be problematic. A modest change in the React application template could have broken the generator if the regular expression pattern did not match the code anymore. It is not visible from the template code which lines are used by the generator. A viable alternative was using specific comments to identify the parts where code was to be generated. The approach slightly conflicted with the requirement 1, which stated that the template should be entirely separated from the generators. However, as otherwise the requirement 2 of easy maintenance would have been compromised, the comments were used to solve the problem.

5.3.3 Implementing generating of boilerplate code

The generating of boilerplate code for new features under `features/` folder were also implemented with Plop by using the templatization method. A sample file structure of generated feature folder is presented in figure 5.4. In the sample, the feature was named "car". When the user enters any other name as input to the generator, the generated files are named correspondingly.

```
{{ #if component.reducers }}
const dispatch = useDispatch();
const {{ camelCase name }} = useSelector(get{{ pascalCase name }});
{{ /if }}
```

Figure 5.5. *Conditional block in a Boilerplate generator template.*

The prompt of the generator consisted of two parts. First, the name of the feature was asked from the user. The second prompt was a multiple-choice question about what types of files are generated from sagas, components, reducers and Redux actions.

The name of the feature was the most crucial part of the model formed from the prompts. All generated files contained the name multiple times with different casing. Plop has predefined helper methods for formatting user inputs in camel case and pascal case [47].

The second prompt also affected the contents of the files and defined which files and folders are being generated. For example, code importing Redux actions was generated to the reducer boilerplate code only if the user had selected to generate Redux actions. Similarly, the reducer file was used in the generated React component boilerplate only if the reducer file was also generated. The functionality was achieved using conditional blocks of the Handlebars templating language. One of the conditional blocks is presented in figure 5.5. The figure's middle two lines are outputted if the user has chosen to generate also reducer boilerplate. The figure also shows how the user input can be modified to use different casing with the `camelCase` and `pascalCase` helpers.

Another option would have been to inject code to other files after they had been generated. However, this approach would have resulted in more Plop generator actions that modified existing code. While implementing the authentication generator, the Plop actions modifying existing files were considered problematic from the maintenance perspective. Therefore, the injecting approach was avoided.

The generated code can be divided into two categories: boilerplate code and example code built on top of the boilerplate code. The boilerplate code is code that appeared, for example, almost in each reducer regardless of the project. The example code built on top of the boilerplate code shows how the boilerplate code is intended to be used. It is removed when the actual project-specific code is implemented on top of the boilerplate code.

Determining the required amount of the example code was considered to be challenging. If there were too much example code, it would require too much effort to remove it each time the generator is run. If too little code was generated, two problems are encountered: it might be impossible to implement the templates so that the project builds without errors, and the intended usage of the templates would be unclear.

It was decided that there would be as little example code as possible to build the project successfully after running the generator. For example, the Redux actions vary a lot between reducers, so they could not be included in the generator template. If the Redux action boilerplate did not have any actual actions defined, the generated code could not build without errors. Therefore, a single example action was included in the generated code. The example action could be easily renamed and modified later with the refactoring tools of integrated development environments or code editors like Visual Studio Code. The finished boilerplate templates contained mostly empty functions and TypeScript type definitions, which had comments stating that they are only generated stubs and should be implemented manually.

The Plop tool combines two unrelated JavaScript libraries, Inquirer.js and Handlebars template engine [47]. There was a slight mismatch between the model structure formed by Inquirer.js and the model structure the Handlebars template engine expected to receive as input. For example, when using a multiple-choice prompt, the user's choices were stored into a JavaScript array. The Handlebars template language did not provide any utilities for checking if a specific string was present in an array. An additional model transformation had to be implemented to bridge the gap between Inquirer.js prompts and the Handlebars template engine. Inquirer.js supports defining filter functions that can alter the output from prompts [53]. The transformation was implemented as a filter function.

Although implementing generating of Sagas, reducers, React components and Redux actions into a single generator reduced the number of Plop actions that modified existing files, there was still five of those Plop actions in the finished implementation. The modifying actions configured the generated reducer and sagas into the Redux store. Special comments such as "Plop reducers" were added to the locations where the templates were being injected. The two files containing the Redux store definition did not have any other logic, so the risk of breaking the generator by manually modifying the files was low. The comments used by Plop also resembled the maintainers that they should be careful when modifying the files.

Plop and Hygen were both relatively simple tools which made learning them easy. More experience was gained from Plop as an issue that prevented modifying existing files in Windows environments was encountered when testing Hygen. However, in addition to being easy to learn, Plop offered comprehensive customizability and extensibility. The libraries Plop was based on, Inquirer.js and Handlebars template engine were both popular and mature libraries and have millions of weekly downloads in NPM registry [54][55]. No defects were found from Plop during the research. Both the prompts and the template engine can be customised and extended in numerous ways [47]. Also, JavaScript suited well as a language for implementing simple code generators. The flexibility of dynamically typed interpreted language made implementing the transformation between the Inquirer.js and the Handlebars effortless.

5.4 Using the generator tools

Although the code generators are used mostly by software developers, they should still provide a friendly user experience. Otherwise, they will not be used [14]. There are often plenty of other tools that the developer must learn in a software project, so the threshold to learn any additional tools is high. The generator tools should imply what they do, what files or folders they modify and inform about errors in a suitable amount of decorum [14]. In the implemented generators, most of the user interaction was handled by the used tools.

Before the generators can be used, the user must create a new project with the Create React App tool. By entering `npx create-react-app my-application --template @cra/cra-template-wapice` into the command-line, the Create React App pulls the latest version of the template from the private NPM registry and outputs a new project. The outputted project has the generators available.

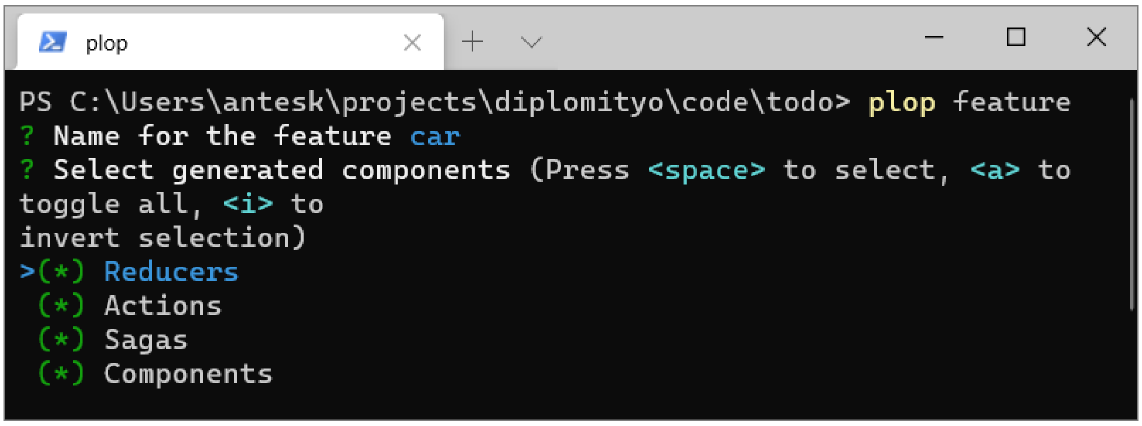
An NPM script invoking the OpenAPI Generator was added to the React application template. The back-end communication code can be generated by invoking `npm run generateApi` command and passing a path to the OpenAPI document as a parameter. This command invokes the actual command of OpenAPI Generator with the correct configuration. In case the OpenAPI specification is later modified or extended, the command can be rerun, and the code under `api/` is updated accordingly.

The two other generators implemented on Plop require more user interaction. The user can only run `plop` command in the project folder, and the two available generators are listed. The user can either select one of the available generators by using arrow keys. Alternatively, the user can directly run the chosen generator by adding the generator's name directly to the command. The user can, for example, run the OpenID Connect authentication generator by invoking `plop oidc-auth` command.

When the OpenID Connect authentication generator is run, the user is asked to enter the OpenID Connect authority URL and a client id. The third prompt asks the user which identity provider is used: Auth0 or IdentityServer. After the last prompt, a list of created and modified files is printed to the terminal. In case some generated file already exists, the user is prompted whether the file should be overwritten or skipped.

By running `plop feature`, the boilerplate code for new features is generated. The shown prompts are presented in figure 5.6. The user is first asked for the name of the feature. The next prompt is a multiple-choice question about which files from the Redux actions, reducers, components and Sagas are generated.

All files are selected by default, but the user can select and deselect the different options using arrows, spaces, and enter keys. Similarly to the OpenID Connect authentication



```
PS C:\Users\antesk\projects\diplomityo\code\todo> plop feature
? Name for the feature car
? Select generated components (Press <space> to select, <a> to
toggle all, <i> to
invert selection)
>(*) Reducers
 (*) Actions
 (*) Sagas
 (*) Components
```

Figure 5.6. *Boilerplate generator prompts.*

generator, a list of modified and added files is printed.

For both Plop generators, the first prompt can also be skipped by adding the value as the third parameter to the command. It is also possible to add answers to other prompts directly in the command as command-line flags [47]. As the user interaction can be skipped, the Plop generators can be invoked from a script.

For example, the template and the generators could be used in a full-stack project consisting of a React front-end, an ASP.NET Core back-end and an IdentityServer identity platform. The developer initializes the front-end project with the customized template by passing `--template @cra/cra-template-wapice` argument into Create React App. Create React App creates a new folder with the template files. Besides, it initializes Git version control for the created folder and installs the latest versions of third-party dependencies.

The developer can proceed by removing the example code under `api/` and `features/` folders. If the back-end implementation has any available endpoints, the developer can generate the code required for communicating with them to the front-end. Swashbuckle is a library that can be used in ASP.NET Core environments to generate an OpenAPI specification based on ASP.NET Core back-end implementation [56]. The developer generates an OpenAPI specification from the back-end using Swashbuckle and then uses the OpenAPI Generator to generate the front-end code based on the specification.

The developer can use the OpenID Connect authentication generator to implement the authentication against IdentityServer. After answering the shown prompts, the required configuration, user-interface components and callback routes are generated to the project. Inserting the authorisation token into the back-end requests must be implemented manually. However, the required steps are documented in the template's documentation.

During the project, the developer can use the OpenAPI Generator and the boilerplate generator to speed up development. When more back-end endpoints become available, the developer can rerun the OpenAPI Generator to update the front-end implementation. When adding new features, the boilerplate for them can be created with the boilerplate Plop generator. For instance, a simple feature can consist of three parts: it makes back-end requests with Sagas, updates the state of the application based on the response in a reducer, and presents the current state in a user interface component. When developing such a feature, the developer can pick reducer, React component, and Sagas from the available boilerplate options. The generated Saga and reducer boilerplate code is automatically added to the application configuration, so the developer can proceed to writing the actual implementation on top of the boilerplate code.

6. RESULTS

6.1 Defining scoring criteria

After the generators had been implemented, they were reviewed. It was discussed in section 2.4.1 that code generating could improve quality and productivity. There was also some risks involved in developing code generators, such as too high maintenance efforts. These observations formed a basis for the scoring criteria used in the review. The review's goal was to highlight the differences between the generators and find answers for the second research question about the parts that benefit most from code generation. Therefore, the scoring is mainly relative and cannot be applied when comparing the implemented generators to other generators. If all generators were considered to perform poorly or exceptionally well, the scores were given accordingly.

A code generator has added value in software projects only if the gained benefits from it are more significant than the resources spent in developing it. The implementation effort was used as one review criterion. Only the effort required to customize the generators for company-specific needs was included in the review. The work needed for developing an open-source or commercial generator is excluded, as the work has been done elsewhere. In case commercial products were used in the research, their costs would have been included in this criterion. However, only open-source products were used in the research.

The second review criterion considered the maintenance aspect of the tool. In section 2.4.2 it was discovered that maintenance might account for over 60 % of the total costs of software development. Therefore the maintenance was included as one review criterion.

As a counterweight to the development and maintenance efforts, the generators' potential productivity gains were also analysed as the third review criterion. The more manual work could be avoided with the generator, the more time and resources are saved.

The fourth review criterion was the customizability of the generators. The customizability came up in the interviews as a preferred feature. This criterion also included the effort required to extend the tool with new features. If it is effortless to adopt the tool into new contexts, there are more potential use cases for the tool, and thus more productivity gains can be achieved by using the tool.

As a summary the defined criteria is listed below:

1. Required effort to implement the generator
2. Required effort to maintain the generator
3. Potential productivity gains
4. Customizability of the generator

The scoring scale was defined to be from zero to five. The bigger the score, the better the generator was from a project management perspective: a higher score for the maintenance attribute implied that maintaining the tool did not require much effort. The score of zero was reserved for situations where the performance of the generator was flawed. For example, if no productivity gains could have been achieved using a generator, it would have gotten the score zero for the criterion. Objective measures such as lines of codes were used whether it was possible.

6.2 Reviewing the generators

6.2.1 Implementation effort

As a single software developer implemented the generators, there were no additional costs to direct personnel costs from the spent time. However, the generators' development time cannot be directly compared when deducing the score for the criterion 1. For example, when developing the first generator for Plop, plenty of time was spent learning the tool. The gained knowledge could be directly applied when developing the second generator, and therefore the time used to develop the second Plop generator decreased. To get more objective results, a divide and conquer approach was used. The required work was divided into smaller units, and time spent on every single unit was analysed. As the development of both Plop generator consisted of similar units such as the development of actions and prompts, the effort can be reliably compared between generators.

The OpenAPI Generator was the only ready-to-use generator, so mostly configuration and testing were required to be done to use the tool. As mentioned in section 5.3.1, a few minor issues were encountered in the configuration. However, the time to get the generator function was only a fraction of the time used to develop the other generators. Therefore, the OpenAPI Generator got a score of five for the criterion 1. The good experiences of the interviewed experts also support the high score.

The other two generators developed on top of Plop also required implementation work. Developing generating of the OpenID Connect authentication took about one third longer than developing the boilerplate generator. However, they both have similar structures: they consist of eight actions and two to three prompts. Both generators had four actions that added new files and four actions that edited existing files. The difference in the implementation time was considered to be caused by the time spend learning Plop. Thus, the difference was excluded from the review. As the logic for handling prompt data was handled by Inquirer.js, most of the time was spent implementing a prototype implementation and turning it into Handlebars templates. Based on the analysis, there were no significant differences in the implementation effort of the Plop generators. Although the implementation was straightforward with both Plop generators, significantly more work was required when compared to the ready-to-use OpenAPI Generator. Therefore the Plop generators are given a score of three for their implementation effort.

6.2.2 Maintenance effort

As the generators had just been implemented, their maintenance efforts had to be approximated by inspecting the implementation. Multiple different quantitative metrics have been developed for analysing code maintainability [57]. It has been observed that the number of code lines correlates highly with the other more sophisticated measures on the method level [58]. The maintenance effort was analyzed by comparing the number of written code lines for each generator. In addition to the quantitative measure, also qualitative analysis was done. For example, the sensitivity to changes was considered in the score.

The OpenAPI Generator is a large product maintained by the OpenAPI community, making analysing its maintenance challenging. The required effort depended on the activity level of the community. If the user had to fix any defects or update the generator by themselves, the maintenance could be cumbersome. On the other hand, if the community handled the maintenance well, there was not much effort left for the user. Similar issues also considered the Plop tool that was used in the two other generators. The benefits of using ready-to-use tools compared to implementing them were listed in section 2.4.2. To emphasize them and to simplify the review process, it was decided that the maintenance effort of code written by an open-source community is excluded from the score. As the OpenAPI Generator was a ready-to-use tool that required only a little configuring, it was rated the score of five also for its maintenance.

The Plop generators' templates will probably require maintaining at some point when technologies such as TypeScript and React evolve. Therefore more work is required than with the OpenAPI Generator. However, as there is no complicated handwritten logic, it is relatively easy to modify the templates. Although the OpenID Connect authentication generator and the boilerplate generator required writing an equal amount of code and share a similar structure, they differ from the maintainability viewpoint. The actions that modify existing code were considered to weaken the maintainability of the template. Both had four of those actions implemented. The OpenID Connect authentication generator modified React components, whereas the boilerplate generator modified Redux and Saga configurations. By investigating the existing React projects, it was noticed that the former files were more prone to change than the latter. The Redux and Saga configurations were present in every project, whereas the React components had often been modified to project-specific needs. To highlight the minor difference in maintainability, the OpenID Connect generator was given a score of two and the boilerplate generator score of three.

6.2.3 Potential productivity gains

The number of generated lines was considered an adequate objective measure for the potential productivity gain. However, there was no straightforward method to calculate the number of generated lines for the OpenAPI Generator and the boilerplate generator. With the OpenAPI Generator, the number of generated lines varied based on the OpenAPI document's size, whereas the size of the output from the boilerplate generator depended on how many times the generator was run. Approximations of the number of generated code lines were made based on investigating the nine existing React projects.

The amount of back-end communication code varied in the projects from about 300 lines of code to almost 10 000 lines. On average, the investigated projects had about 3 000 lines of back-end communication code.

For the OpenID Connect authentication generator, the amount of generated code was approximated based on the implemented templates. The generator was not intended to be run more than once per project, so the amount of generated code is almost static. The templates had about 80 lines of code.

The boilerplate generator could be run multiple times in a project, so the amount of generated code could not be approximated solely by inspecting the templates. The times the generator was typically run in a project was approximated by counting the number of features implemented in the existing React projects. Three projects were excluded from the investigation as their files were not divided by different features. The number of features varied in the rest six projects from 3 to 17. On average, each project had eight features. The boilerplate generator outputted about 60 lines of code in each run. The approximation of the amount of generated code was made by multiplying the average number of features by the number of outputted code lines in a single run. Based on the calculation, the approximated number of generated code lines in a single project was 480.

The OpenAPI Generator was approximated to output remarkably more code than the Plop generators. Hence, its potential productivity gain criterion was rated with a score of five. As the approximation for the amount of generated code by the boilerplate generator was six times the amount of the OpenID Connect authentication generator, the former generator was given a score of two and the latter score one.

Table 6.1. *Review results. Scores are between 0-5.*

Review criterion	OpenAPI Generator	OpenID Connect Authentication Generator	Boilerplate Generator
1. low implementation effort	5	3	3
2. low maintenance effort	5	2	3
3. productivity gains	5	1	2
4. customizability	2	5	5

6.2.4 Customizability

There were few objective measures for the customizability. Instead, the criterion was scored by investigating the generator tools' sources and documentation and determining the required work based on it. The OpenAPI Generator supports implementing new generators and modifying the existing ones [42]. The generators are implemented inside the source of OpenAPI Generator. The OpenAPI Generator source code must be cloned and built in a local environment to modify or add new generators [43]. The OpenAPI community's further changes to the OpenAPI Generator need to be manually merged into the customized version, making maintaining more difficult. Also, there is no straightforward way to apply the customizations to the NPM wrapper used in this thesis [43]. Based on the factors above, the OpenAPI generator's customizability gets a score of two, although extending and modifying the generator is supported and documented [42].

In the Plop generators, the templates are directly included in the project, making implementing new generators and modifying the existing generators easy. The modifications are available for all the other developers in the project team. There are no predefined generators, so implementing new generators is also thoroughly documented [47]. Both Plop generators are given a score of five.

6.2.5 Review results

The review results are summarized in the table 6.1. As mentioned earlier, the higher the score is, the better the generator is from the project management perspective. For example, a high score for the maintenance effort criterion implies that the generator is easy to maintain.

As shown from the results, the OpenAPI Generator was given the best score in almost all categories. A significant factor in the good scores was that OpenAPI Generator is a ready-to-use open-source product. It has comprehensive support for generating client and server-side code based on OpenAPI documents [42]. Developing a corresponding generator from scratch could not have been possible within the scope of the thesis.

From the two Plop generators, the boilerplate generator was given better scores for easier maintainability and better potential productivity gains. The files the boilerplate generator modified were less prone to change. The boilerplate generator could be run multiple times in a single project which increased the available productivity gains. As Plop had the templates included in the projects it was used, customizing them was straightforward. Therefore, the Plop generators excelled when reviewing the criterion 4.

6.3 Risks for validity

Although objective metrics such as implementation time and the number of code lines were used in the review, the research also included a subjective element. The scores reflected the experiences received when implementing and testing the generators. The results may be different if, for example, different tools were used or the generators were designed differently.

The scoring's main goal was to summarize the differences of the implemented generators in a clear way rather than provide universally utilizable results. Therefore, the results are mainly relative and cannot be directly used in other reviews. For example, defining requirements for getting the best score for the productivity gains criterion would have been challenging, as there is no sensible theoretical limit on how much code a single generator can output.

There was not yet experimental data available for the generators. Therefore analysing the maintenance efforts could be done only on a rough level by comparing the number of written code lines.

As mentioned in the section 4.2, the used research material was relatively small, consisting of nine projects developed within a single company. Applying the results of the case study to other contexts should be done carefully. The same projects were used in forming the requirements and in analysing the finished generators. The approximated number of generated code lines, for instance, depends highly on the projects. If more projects or projects from different contexts were analysed, the approximated number of code lines could differ. On the other hand, all of the projects were production-grade software developed to solve real-world problems, which improved the study's reliability.

As the scope of a thesis is limited, the number of implemented generators was also small. Different kinds of generators and targets to be generated were chosen in order to get more comprehensive insight. However, all the tools applied template-based code generation, leaving the other techniques untouched. Many areas of single-page applications that could be generated were also left uncovered by this research.

7. CONCLUSIONS

The research was to investigate if code generation could be applied to single-page applications. The research was done as a planning and implementation project by developing and using three different code generators. The generators' requirements were defined based on investigating source code and interviewing the developers of nine existing production-grade React applications.

In the first research question of the thesis, it was asked if code generation could reduce mechanical work at the beginning of single-page application projects. Based on the research, it seems that code generation can be successfully applied to single-page projects. It was approximated that by average 3 000 lines of code could be generated in a single project with the OpenAPI Generator. With the custom made boilerplate generator, the corresponding amount was almost 500 lines of code per project.

The initial focus was on the beginning of software projects, as it was hypothesised that it would be a fertile place to apply code generation. The hypothesis was directly included in the first research question. However, based on the study, it was found out that focusing only on the beginning of software projects is an unnecessary constraint. Two of the three used generators – the OpenAPI Generator and the boilerplate generator – were such that they could also be used later in the software projects.

Insight into the second research question was also gained in the research. The second research question asked what parts of single-page applications could benefit the most from code generation. One such part was found out to be back-end communication code. The services of the back-end can be modelled based on OpenAPI specification. As the specification is designed to be also machine-readable, it is possible to transform it to working code with OpenAPI Generator. As various back-end frameworks such as ASP.NET Core and Spring have tools for generating the documentation based on the actual back-end implementation, generating back-end communication code for the single-page application can be fully automated.

The parts that can be modelled in higher abstraction seem to be well suited for code generation. However, depending on the model's size and complexity, these kinds of generators tend to be large and therefore implementing these from scratch can be challenging. Using existing generators should be preferred in order to reduce the risks.

Generating boilerplate files for new Redux actions, Redux reducers, sagas and React components was also found out to be a suitable place to use generation. The boilerplate code repeatedly appeared in the investigated React projects. As the same generator can be run multiple times, more code can be created with the same generator, which leads to improved productivity gains without affecting development or maintenance costs.

Based on the research, boilerplate code seems to be a good target for smaller self-made generators. The boilerplate code often follows a specific structure, so no complex logic is required for the generator. The templates for the generator can be created from existing code using the templatization technique. Boilerplate code also appears repeatedly in a project so that the same generator can be run multiple times. In case the boilerplate repeatedly appears inside a file, the functionality of an integrated development environment or code editor such as code snippets of Visual Studio Code can be used to generate it [40]. Instead, if the generator creates or modifies multiple files, a micro-generator framework such as Plop may be a suitable tool.

The least successful generator was the OpenID Connect authentication generator. The initial idea was to try to automate a complex task such as configuring authentication. However, it was possible to automate only part of the work to avoid conflicts with the OpenAPI Generator. The developer could not use the generator without knowing the generated code's implementation details; the complex task was not entirely abstracted away. Plop did not directly support installing NPM packages, so the OpenID Connect client library had to be manually installed. As the generated amount of code was also relatively small, the potential productivity gains were considered too small compared to maintenance efforts.

The code generation of Vue CLI and Angular CLI was found at a late stage of the thesis project. This thesis's tools were already implemented when it was found out that these command-line interfaces automate similar tasks such as configuring libraries or creating boilerplate. Also, their design was very similar to the Plop generators: both Vue CLI plugins and Angular CLI schematics used prompts to interact with the user. The finding highlighted that also other developers had encountered similar issues when developing single-page applications.

Further research could be done in order to gain more comprehensive information about generating parts of single-page applications. This research relied on approximations when reviewing the generators. Experimental research should be done to get more precise information about the generators. An exciting research area would be to compare the actual costs of developing a code generator to the saved costs to find out when code generating is profitable.

As mentioned in section 6.3, only a few tools were covered by this research. In addition to generating only parts of single-page applications, it is also possible to generate whole applications. For example, Divjoy generates whole React applications [59]. JHipster goes a step further, as it generates whole full-stack applications consisting of a front-end and back-end [60]. Researching using those tools could provide more insight into what issues and benefits the generator's larger scope has.

In addition to investigating different tools, the tools used in this research could also be utilized in other software development areas than single-page applications. A natural area to extend the research under web development would be researching generating back-end code. OpenAPI Generator also has generators for generating back-end stubs based on OpenAPI documents [42]. Layered architecture is commonly used in web back-ends [61, ch. 1]. Plop could be used to generate the boilerplates for the components of the different layers. Further research is required in order to see how well the tools suit back-end development.

Although code generation is often linked to model-driven development and generating whole systems, it can also be inspected from a different perspective. Most developers are used to renaming variables, extracting methods, and importing external files with integrated development environments. Code generating can be seen as a tool that helps developers in their daily work. The generators implemented on top of Plop were simple but yet powerful. They offer similar functionality as the wizards of integrated development environments, but due to their open nature are easier to customize and extend.

However, there is also room for larger and more complex generators. Using such generators can provide good results, as found in this research with OpenAPI Generator. With the extending selection of open-source and commercial tooling available, there are few reasons why code generating should not be applied in daily software development.

REFERENCES

[1] Forsgren, N., Smith, D., Humble, J. and Frazelle, J. *2019 Accelerate State of DevOps Report*. Tech. rep. 2019. URL: https://services.google.com/fh/files/misc/state-of-devops-2019.pdf.

[2] Garreau, M. *Redux in action*. eng. Shelter Island, NY, 2018.

[3] Brooks. No Silver Bullet Essence and Accidents of Software Engineering. eng. *Computer (Long Beach, Calif.)* 20.4 (1987), pp. 10–19. ISSN: 0018-9162.

[4] McConnell, S. *Code complete*. eng. 2nd ed. Redmond, WA: Microsoft Press, 2004, 914 sivua. ISBN: 0-7356-1967-0. URL: https://tuni.finna.fi/Record/oma.111462.

[5] Parnas, D. L. Software aspects of strategic defense systems. eng. *Software engineering notes* 10.5 (1985), pp. 15–23. ISSN: 0163-5948.

[6] Falzone, E. and Bernaschina, C. Intelligent Code Generation for Model Driven Web Development. *Current Trends in Web Engineering*. Ed. by C. Pautasso, F. Sánchez-Figueroa, K. Systä and J. M. Murillo Rodríguez. Cham: Springer International Publishing, 2018, pp. 5–13. ISBN: 978-3-030-03056-8.

[7] Jörges, S. and Kittler, J. *Construction and Evolution of Code Generators: A Model-Driven and Service-Oriented Approach*. eng. Vol. 7747. LNCS sublibrary. SL 2, Programming and software engineering. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2013. ISBN: 9783642361265.

[8] K, C. and S, H. Feature-based survey of model transformation approaches. eng. *IBM systems journal* 45.3 (2006), pp. 621–645. ISSN: 0018-8670.

[9] Kühne, T. Matters of (Meta-) Modeling. eng. *Software and systems modeling* 5.4 (2006), pp. 369–385. ISSN: 1619-1366.

[10] Fowler, M. *Domain-specific languages*. eng. 1st edition. The Addison-Wesley signature series. Boston, MA: Addison Wesley. ISBN: 1-282-79730-1.

[11] Whittle, J., Hutchinson, J. and Rouncefield, M. The State of Practice in Model-Driven Engineering. eng. *IEEE software* 31.3 (2014), pp. 79–85. ISSN: 0740-7459.

[12] Tolvanen, J.-P. and Kelly, S. Model-Driven Development challenges and solutions: Experiences with domain-specific modelling in industry. eng. *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SCITEPRESS, 2016, pp. 711–719. ISBN: 9897582320.

[13] Syriani, E., Luhunu, L. and Sahraoui, H. Systematic mapping study of template-based code generation. eng. *Computer languages, systems & structures* 52 (2018), pp. 43–62. ISSN: 1477-8424.

[14] Herrington, J. *Code generation in action*. eng. 1st edition. Greenwich, CT: Manning. ISBN: 1-932394-07-9.

[15] Arnoldus, J. *Code Generation with Templates*. eng. 1st ed. 2012. Atlantis Studies in Computing, 1. Paris: Atlantis Press. ISBN: 1-283-63445-7.

[16] *Type Erasure. The Java Tutorials*. 2020. URL: `https://docs.oracle.com/javase/tutorial/java/generics/erasure.html` (visited on 03/19/2021).

[17] Possatto, M. A. and Lucrédio, D. Automatically propagating changes from reference implementations to code generation templates. eng. *Information and software technology* 67 (2015), pp. 65–78. ISSN: 0950-5849.

[18] Völter, M. and Bettin, J. Patterns for Model-Driven Software-Development. Jan. 2004, pp. 525–560.

[19] Maedche, A., Botzenhardt, A. and Neer, L. Software for People: A Paradigm Change in the Software Industry. *Software for People: Fundamentals, Trends and Best Practices*. Ed. by A. Maedche, A. Botzenhardt and L. Neer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–8. ISBN: 978-3-642-31371-4. DOI: `10.1007/978-3-642-31371-4_1`. URL: `https://doi.org/10.1007/978-3-642-31371-4_1`.

[20] Elec, T., Sci, C., Uyanik, B. and Şahin, V. A template-based code generator for web applications. *Turkish Journal of Electrical Engineering and Computer Sciences* 28 (May 2020), pp. 1747–1762. DOI: `10.3906/elk-1910-44`.

[21] Blackburn, M. *Risks and Mitigation Strategies for Using Automatic Code Generation Tools*. Tech. rep. Apr. 2004. DOI: `10.13140/RG.2.1.3688.1524`.

[22] Bucchiarone, A., Cabot, J., Paige, R. F. and Pierantonio, A. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling* 19.1 (Jan. 2020), pp. 5–13. ISSN: 1619-1374. DOI: `10.1007/s10270-019-00773-6`. URL: `https://doi.org/10.1007/s10270-019-00773-6`.

[23] Dey, P., Kinch, J. and Ogunlana, S. Managing risk in software development projects: A case study. *Industrial Management and Data Systems* 107 (Mar. 2007), pp. 284–303. DOI: `10.1108/02635570710723859`.

[24] Syavasya, C. Estimating software maintenance cost by varying maintainability metric. eng. *International journal of advanced research in computer science* 3.5 (2012). ISSN: 0976-5697.

[25] Raymond, E. S. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. eng. Sebastopol: O'Reilly Media, Incorporated, 2001. ISBN: 0596001088.

[26] *Choose Between Traditional Web Apps and Single Page Apps (SPAs)*. Jan. 12, 2020. URL: `https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps` (visited on 02/13/2021).

[27] Sferruzza, D. Top-down model-driven engineering of web services from extended OpenAPI models. eng. *Proceedings of the 33rd ACM/IEEE International Conference on automated software engineering*. ASE 2018. ACM, 2018, pp. 940–943. ISBN: 145035937X.

[28] *OpenAPI Specification. Version 3.0.3*. URL: https://swagger.io/specification/ (visited on 02/25/2021).

[29] *SOAP vs REST 101: Understand The Differences*. 2020. URL: https://www.soapui.org/learn/api/soap-vs-rest-api/ (visited on 03/19/2021).

[30] *React. A JavaScript library for building user interfaces*. 2021. URL: https://reactjs.org/ (visited on 02/13/2021).

[31] Banks, A. *Learning React : modern patterns for developing React apps*. eng. Second edition. Beijing: O'Reilly, 2020. ISBN: 1-4920-5169-1.

[32] Griffiths, D. *React Cookbook*. eng. 1st edition. O'Reilly Media, Inc., 2021. ISBN: 1-4920-8583-9.

[33] *Vue CLI. Standard Tooling for Vue.js Development*. URL: https://cli.vuejs.org/ (visited on 03/19/2021).

[34] *CLI Overview and Command Reference*. URL: https://angular.io/cli (visited on 03/19/2021).

[35] *About semantic versioning*. URL: https://docs.npmjs.com/about-semantic-versioning (visited on 03/24/2021).

[36] *Create React App. Set up a modern web app by running one command.* 2021. URL: https://create-react-app.dev/ (visited on 02/13/2021).

[37] *Generating code using schematics*. URL: https://angular.io/guide/schematics (visited on 03/20/2021).

[38] *Authoring schematics*. URL: https://angular.io/guide/schematics-authoring (visited on 03/20/2021).

[39] *vue-cli-plugin-vuetify. A Vue CLI 3 Plugin for installing Vuetify*. URL: https://www.npmjs.com/package/vue-cli-plugin-vuetify (visited on 03/20/2021).

[40] *Snippets in Visual Studio Code*. Apr. 2, 2021. URL: https://code.visualstudio.com/docs/editor/userdefinedsnippets (visited on 02/11/2021).

[41] *File Structure. Is there a recommended way to structure React projects?* URL: https://reactjs.org/docs/faq-structure.html (visited on 04/11/2021).

[42] *Create React App. Generate clients, servers, and documentation from OpenAPI 2.0/3.x documents*. Feb. 14, 2021. URL: https://openapi-generator.tech/ (visited on 02/24/2021).

[43] *OpenAPI generator NPM command line interface*. URL: https://www.npmjs.com/package/@openapitools/openapi-generator-cli (visited on 02/24/2021).

[44] *Fetch API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API (visited on 04/11/2021).

[45] *Yeoman*. URL: https://yeoman.io/ (visited on 02/12/2021).

[46] *Hygen*. URL: https://www.hygen.io/ (visited on 02/12/2021).

[47] *Plop*. URL: https://yeoman.io/ (visited on 02/12/2021).

[48] *NPM packages in the Package Registry. GitLab Docs*. URL: https://docs.gitlab.com/ee/user/packages/npm_registry/ (visited on 02/25/2021).

[49] *JSON Server. Github*. URL: https://github.com/typicode/json-server (visited on 03/24/2021).

[50] *JSON Server Auth. Github*. URL: https://github.com/jeremyben/json-server-auth (visited on 03/24/2021).

[51] *IdentityServer. The Identity and Access Control solution that works for you*. 2018. URL: https://identityserver.io/ (visited on 04/03/2021).

[52] *Auth0. Secure access for everyone. But not just anyone*. 2018. URL: https://auth0.com/ (visited on 04/03/2021).

[53] *Inquirer.js. A collection of common interactive command line user interfaces.* Feb. 25, 2021. URL: https://github.com/SBoudrias/Inquirer.js/blob/master/README.md (visited on 03/07/2021).

[54] *Inquirer.js*. URL: https://www.npmjs.com/package/inquirer (visited on 03/07/2021).

[55] *Handlebars.js*. URL: https://www.npmjs.com/package/handlebars (visited on 03/07/2021).

[56] *Swashbuckle.AspNetCore*. URL: https://github.com/domaindrivendev/Swashbuckle.AspNetCore (visited on 04/16/2021).

[57] Ardito, L., Coppola, R., Barbato, L. and Verga, D. A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review. eng. *Scientific programming* 2020 (2020), pp. 1–26. ISSN: 1058-9244.

[58] Eskelinen, A. Ohjelmakoodin kompleksisuuden mittaaminen metoditasolla. Finnish. Bachelor's Thesis. Tampere, Finland: Tampere University, 2019.

[59] *Divjoy. The React Codebase Generator*. 2021. URL: https://divjoy.com/ (visited on 03/18/2021).

[60] *JHipster. Full Stack Platform for the Modern Developer!* 2021. URL: https://www.jhipster.tech/ (visited on 03/18/2021).

[61] Richards, M. *Software architecture patterns : understanding common architecture patterns and when to use them*. eng. First edition. Sebastopol, CA: O'Reilly Media, 2015. ISBN: 1-4919-7143-6.