Removing Code Duplication Through Code Generation for Kotlin Web Services

Mathias Enggrob Boon

Kongens Lyngby 2020

Abstract

In web services development, it is common to maintain multiple sources of truth for similar code due to the decoupled nature of client-server systems, leading to code redundancy. A popular solution is to use a specification-based approach to generate API client libraries for requesting web services, partially solving the problem.

In this thesis, it is shown that it is possible to generate both the web API and API client library from application service definitions, maintaining a single source of truth. This goal was achieved by designing and implementing a library that generates code through annotation processing. The library was evaluated through a series of tests and interviews which indicate that it is effective at improving developer efficiency by eliminating code redundancy.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Computer Science and Engineering.

The thesis deals with the area of web services, seeking to explore options for improving on the industry standard practice of manually adapting the web API to the underlying server logic and generating the client code for requesting the service from a specification of the web API.

The thesis consists of a report, a proof-of-concept Kotlin library and a sample Kotlin project demonstrating the library in use.

Kongens Lyngby, June 17th, 2020

Mathias Enggrob Boon

Acknowledgements

The author of this thesis would like to thank his supervisors, Jakob Eyvind Bardram, Steven Jeuris and Alban Maxhuni, for their support during the thesis development and writing period.

An additional thank-you to Steven Jeuris for his extensive feedback during the writing process of the thesis.

Finally, a thank-you to the interview participants who provided feedback on the proof-of-concept implementation of the library.

Contents

Acronyms

DDD domain-driven design.

DRY Don't Repeat Yourself.

DSL domain-specific language.

HATEOAS Hypermedia as the Engine of Application State.

HTTP HyperText Transfer Protocol.

IP Internet Protocol.

JSON JavaScript Object Notation.

MDA model-driven architecture.

MDE model-driven engineering.

MSA microservice architecture.

REST Representational State Transfer.

RPC remote procedure call.

SLOC source lines of code.

SOA service-oriented architecture.

SOAP Simple Object Transfer Protocol.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

UML	Unified Modelling Language.
URL	Uniform Resource Locator.
WWW	World Wide Web.
XML	Extensible Markup Language.

Glossary

API client library library containing a collection of client methods specific to a web API or group of web APIs.

application service interface that defines the functionality a system offers to other systems.

client method method used by client applications for making a request to an endpoint.

endpoint in the context of web services, an identifier for a single service. In the context of web applications, a code block associated with a URL that is executed when a request to this URL is made.

Kodegen informal name for the proof-of-concept implementation of the library designed in the thesis.

web API interface of a web server consisting of a collection of endpoints.

web application application that provides services through a web server.

web services the services provided by a web application.

CHAPTER 1

Introduction

In this chapter, the motivation for exploring this subject is given, the problem statement and project proposal is presented, and the structure of the thesis is explained.

1.1 Background and Project Motivation

1.1.1 Development of Client-server Systems

One of the challenges when designing distributed systems is building an interface between the components of the system, allowing them to communicate. In a non-distributed application, static checking can be used to catch bugs before the application is executed, e.g. due to syntax errors or use of data types that are not applicable in the given context.

In a distributed system, however, the components may not have any knowledge of the internals of other components and therefore, static checking may not be available. Instead, each component must agree on a protocol with the other components, agreeing on how to communicate. The checks that would otherwise be made statically now depend on the developers of each component correctly implementing their protocols to interface with other components. This complicates development, as bugs that would otherwise be caught at compile-time may go unnoticed until encountered during run-time.

Architectural patterns and software engineering methodologies for maintaining compatible interfaces between components of distributed systems have been developed to help overcome these difficulties. Among these, service-oriented architecture has become highly popular in the development of distributed systems because it simplifies inter-application communication. Rather than exposing the full logic of a system, the logic is consolidated into a number of self-contained services, which are then exposed to other applications. These other applications do not need to concern themselves with the inner working of each services, only how to access it and what result to expect from the service.

In a web-based distributed system, services are commonly published using a web API. The web API is used by a server to expose a number of endpoints, each of which corresponds to a service provided by the server. Knowing the endpoints of the server, other applications can request the provided services by making a network request to the relevant endpoint and reading the response sent by the server.

While this simplifies the process of accessing the services provided by the server, several issues are still present:

- Developers of client applications must know which endpoints exist, what service they correspond to and the protocol for making requests. Moreover, developers must manually write the logic for accessing each endpoint, i.e. making the network request and interpreting the response.
- Maintainers of the server must ensure that when the server logic is updated, the web API remains consistent with the service they provide. If the client and server applications are developed together, this problem is exacerbated, as developers will need to maintain the actual server logic, the server logic for publishing it to an endpoint, and the client logic for accessing it. This also violates the principle of Don't Repeat Yourself (DRY), which states that every piece of code must originate from one source only.

A potential solution for the first issue is for the developers of the server to provide an API client library, i.e. a library that can be imported by client applications and which defines a number of methods or even a domain-specific language for accessing the web API. This approach has been employed by several organisations, e.g. Google [23], Discord [10] and Slack [37].

However, API client libraries do not solve the second issue, as they must still be maintained to remain consistent with the web API, which in turn must remain consistent with the underlying server logic.

Partial solutions currently exist, notably maintaining a specification of the web API and generating the client library from this. This approach can even be extended to generate the specification from the web API. However, this still leaves the task of maintaining the web API to remain consistent with the internal server logic.

This redundancy and violation of the DRY principle serves as part of the motivation for this thesis. Rather than using tools to assist developers in keeping the components of a distributed system consistent with each other, this thesis seeks to explore options for guaranteeing that the components remain consistent by generating them from one source only.

1.1.2 Domain-Driven Design in Distributed Systems

Domain-driven design (DDD) is a software design methodology useful for systems that operate on complex domains. When using DDD, the domain logic ideally is the domain model, serving as an abstraction of the real-life domain and allowing solving of problems.

As an example, a domain may be the area of medical studies. A domain model of this domain would explain how the different concepts relate to each other and how they interact, e.g. the attributes that constitute a study and how the study is conducted.

A domain experts will have a detailed understanding of the area. By contrast, a technical expert responsible for the development of a system relating to medical studies may only have a superficial understanding and is therefore not able unable to create an accurate domain model. Therefore, collaboration between the domain expert and technical expert is encouraged to develop a model that accurately depicts the domain and is implementable as a system.

To represent the possible interactions with a system, application services are used, serving as the interface of the system. In a distributed system, however, components are loosely coupled, making it difficult to enforce the interface. A client application may be able to make a request to a server despite not having access to the definition of the application services, allowing it to misinterpret the interface.

A potential solution to this is to make network requests available through an API client library that is guaranteed to be consistent with the application service definitions. In contrast to the API client libraries described in Section 1.1.1, these API client libraries do not just reflect the Web API, but also the underlying application services that are exposed by the web API.

By ensuring that the client uses these API client libraries which are guaranteed to be consistent with both the web API and the underlying application services, it is guaranteed that the client application interfaces correctly with the server. Moreover, using an API client library may allow static checking to be used, ensuring that the parameters used in any requests and the use of the response returned is valid.

This serves as the second motivation of the thesis, i.e. eliminating the potential for misinterpreting the services offered by a server and encouraging the use of domain-driven design in web services development.

1.1.2.1 Impact of Thesis

The goal of the thesis is to present a potential method for solving the problems presented in Section 1.1. Doing so could increase developer efficiency by reducing the amount code to developer and maintain, while also reducing the risk of bugs occurring due to developer errors.

For systems developed using domain-driven design, the solution proposed in the thesis may assist in ensuring that client applications respects the application service definitions used by the server.

1.2 Problem Statement

Based on the background presented in the Section 1.1, a problem statement for the thesis was formulated. The problem statement is as follows:

"Given a distributed system with a client-server architecture, is it feasible to automatically generate the logic for providing and requesting the server domain logic from the server domain logic alone?"

1.2.1 Project Proposal

In order to answer the problem statement proposed above, a number of project objectives were identified. These are:

- Identify existing research and technologies related to automatic generation of web APIs and API client libraries.
- Design and implement a proof-of-concept library with the purpose of automatically generating the code necessary to facilitate communication between a client and server application.
- Evaluate the effectiveness of the library by comparing how it affects the software metrics of a project using it, as well as through a qualitative study on the developer experience of using it.
- Identify the limitations of the design and proof-of-concept implementation and potential areas for further improvements.

1.2.2 Project Delimitation

The following areas of the problem domain, while relevant to the problem statement and interesting to explore, are considered outside the scope of this project, and will not be considered in this thesis:

- **API versioning**: API versioning is firstly a business issue, as the owner of the system must decide how long they wish to support a previous version of an API. While it is possible to allow multiple versions of an API to be active, organisations trend towards deprecating and eventually closing old API versions when third-party clients have been given a reasonable time to migrate to the new version [11]. Thus, automatically generated Web APIs and clients for accessing them are no different, as the problem does not lie in how quickly a new API can be generated, but rather how long previous ones should be maintained.

- **Generation of domain logic**: This project focuses on how the application services of a server are exposed, and does not concern itself with the domain logic used by the application services or whether these can be subject to automatic generation.
- **Multiplatform support and support for multiple network protocols**: While Kotlin supports compiling into multiple languages, doing so adds a significant amount of complexity that is not appropriate for a proof-of-concept. Moreover, as the main focus of this project is generating Kotlin code from Kotlin code, multiplatform compilation is not relevant for the core of the project, although it does increase the applicability of the library. Finally, Kotlin Multiplatform compilation currently only has limited support for reflection.
  Similarly, the project will not consider supporting multiple protocols, e.g. UDP, as this increases the complexity of the project greatly.

1.3 Thesis Structure

The thesis is divided into a number of chapters which will largely correspond to the project objectives presented above. The chapters of the thesis are:

- **State of the Art**: Outlines the background knowledge for the area that the thesis relates to, presents the existing methods for developing web services and the problems that are currently relevant in this area.
- **Design**: Documents the design of the proposed solution to the problem statement.
- **Implementation**: Documents the proof-of-concept implementation of the proposed design, emphasizing the decisions taken during implementation and justifying them. Also includes a guide to getting using the library with examples.
- **Evaluation**: Documents the methods used to evaluate the proof-of-concept implementation's effectiveness at solving the problem statement.
- **Discussion**: Reflects on the project, based on the design process and evaluation results, to determine what the project achieved and how it fits into the existing research.

- **Conclusion**: Summarizes the project outcome and proposes potential future work.

Chapter 2

State of the Art

The purpose of this chapter is to present the knowledge that this thesis is based on.

Firstly, a brief introduction to areas related to the thesis is given. This introduction will be "from the ground up", i.e. first explaining more general areas before introducing the areas more specific to the thesis.

Secondly, existing research and industry practices related to the problem statement of the Thesis is presented.

2.1 Distributed Systems

Unless stated otherwise, the theory presented in this section is based on the book "Distributed Systems: Concepts and Design" by Coulouris et al. [8].

According to Coulouris et al., a distributed system is a system in which the components are spread across multiple, networked devices, and where coordination occurs through message passing. Moreover, they specify three significant characteristics of distributed systems:

- **Concurrency of components**, i.e. each component is able to carry out actions concurrently with others.
- **Lack of a global clock**, i.e. components are not able to determine the order of actions across components using a shared clock.
- **Independent failures of components**, i.e. one component may fail while others continue operating.

Because of these characteristics, developing applications running in a distributed context is more complex than a single-device application, as applications in a distributed context must take them into account.

Communication between components in a distributed system is, at the most basic level, facilitated by message passing. This allows components to invoke behaviour in other components, e.g. invoking a method call.

Message passing may be synchronous, where each process waits for a message before proceeding, or asynchronous, where processes may continue working on other tasks while waiting for messages.

The asynchronous implementation useful, as it allows the components to operate concurrently on various tasks and serve several components at the same time and is thus ubiquitous in modern distributed systems.

2.1.0.1 Remote Procedure Calls

Remote procedure call (RPC) is a high-level implementation of message passing. Whereas message passing has no requirements on the ordering of messages between components, RPC is a request-response protocol. Thus, each procedure call is initiated with a request describing the procedure to call along and concludes with a response containing the result of the procedure.

Many implementations of RPC exist, e.g. JSON-RPC or Java RMI. Each implementation defines its own protocol, e.g. how to encode content in the message or how to determine the corresponding request for a response. Most often, the protocols are not compatible with each other, and thus, for two components to communicate using RPC, they must agree on the implementation to use.

2.1.1 Distributed System Architectures

The architecture of a distributed system refers to how the components of the system are distributed and what relationship they have to each other.

The two main distributed architectures are client-server and peer-to-peer.

2.1.1.1 Client-server Architecture

In a client-server architecture, two types of nodes exist: servers and clients. A server provides services, while clients request them. This approach is analogous to a function call: a client provides a request containing the details of the service they want and how. The server parses the request, invokes the service as described and responds with a result.

As with the remote procedure call, the request-response protocol has the benefit of hiding internal server details from the client. The client does not need to know how a service result is found, or if any other clients are requesting the service, only how to request the service and how to parse the response. How to request the service and parse the result is defined using an application layer protocol, e.g. HTTP (See section 2.2.1). The rules for how to make requests and responses may be specified even further by defining an API for the server (see Section 2.3).

Client-server architectures are used ubiquitously for distributed systems. As an example, the World Wide Web (WWW) is based on the use of the client-server model: web servers accept requests for some resource and respond with the requested content. Though originally centred around sharing of documents for displaying web sites, the WWW has evolved into a system for sharing of resources in general.

2.1.1.2 Peer-to-peer Architecture

In a peer-to-peer architecture, every node has the same role. Thus, contrary to a client-server architecture, where each node either provides or consumes services, peers both provide and consume services.

Peer-to-peer architecture is especially useful for applications that can benefit from decentralization, e.g. file sharing. Rather than having a centralized network where a server provides files, files are hosted by one or multiple peers, who can provide them to other peers that request them. If one peer leaves the network, the remaining peers can still provide the file, leading to high reliability.

However, peer-to-peer networking also suffers from lower stability, as the quality of the service is based on the participants. In the file sharing example, a file may be hosted by a number of nodes. If all of these leave the network, the file will no longer be available, even though the network is still operational. By comparison, it is easier to ensure that a server remains online and available.

While the thesis project could be expanded to also consider peer-to-peer systems, the main focus will be on the field of web services, which is based on client-server architecture.

2.2 Computer Networks

A computer network is a network of connected computers, referred to as "nodes" in networking. The network allows them to share information between each other, forming the infrastructure of a distributed system.

Communication over a network is facilitated by a protocol specifying the rules for how messages are sent and received. However, it is infeasible to create a protocol that fully specifies communication over a network, as there are too many aspects to consider, e.g. converting physical signals to digital signals or how to find the recipient of a message.

Instead, a layered protocol is commonly used. Low-level protocols solve issues at a lower abstraction level, allowing high-level protocols built on top of them to make certain assumptions and solve issues at a higher abstraction layer.

2.2.1 Relevant Protocols

In two of the most common conceptual models for networking, the OSI model and TCP/IP model, layered protocols are divided into seven layers (OSI model) or five layers (TCP/IP). Explaining the details of each layer and giving examples of corresponding protocols is outside of the scope of the thesis, as most are not relevant enough to warrant a full description. Instead, the protocols that are most relevant will be described briefly.

2.2.1.1 Internet Protocol

The Internet Protocol (IP) facilitates communication between networks, allowing nodes to communicate with nodes outside of their immediate network. Each node is assigned an IP address to identify it and uses a forwarding algorithm when receiving a message to determine when and how to forward messages to other nodes.

Because the Internet is based on the use of the Internet Protocol and because of the popularity of the Internet for resource sharing, the Internet Protocol has become the standard protocol for communication between networks, and forms the communication infrastructure used by many higher-level services together with TCP.

2.2.1.2 Transport Layer Protocol

Transmission Control Protocol (TCP) is a transport layer protocol, and is therefore responsible for providing a number of guarantees related to delivery of messages. Some of the notable guarantees provided are:

- Error-checking of delivered messages.
- Reliable delivery of messages by retransmitting dropped messages.
- Ordering of messages.

Because of the guarantees provided by TCP, it is often used as the transport protocol for communication protocols and forms the communication infrastructure for many higher-level services together with the Internet protocol.

The disadvantage of using TCP is that each packet has a larger header compared to e.g. UDP and that each message requires more processing on reception.

2.2.1.3 User Datagram Packet

As TCP, User Datagram Protocol (UDP) is a transport protocol. In contrast to TCP, which provides reliable transmission of messages, UDP does not guarantee that messages are delivered. Moreover, message ordering is not implemented, and while message errors can be detected, they are solved by discarding the message, not by using error recovery.

While UDP is less reliable, it is also a more lightweight protocol, with the UDP header being 8 bytes compared to TCP header, which is between 20 and 60 bytes, depending on the header options used. Thus, it is commonly used for time-sensitive applications where a loss of messages is acceptable, e.g. voice communication or online games.

However, the guarantees provided by UDP are often insufficient for higher-level protocols unless specifically compensated for. Supporting UDP is considered less relevant to the thesis than TCP.

2.2.1.4 HyperText Transfer Protocol (HTTP)

HTTP is an application protocol for exchanging general application data, with many applications, notably the World Wide Web, being based on the use of HTTP. Because of the popularity of the World Wide Web for resource sharing, HTTP is commonly used as a general-purpose application layer protocol, with other application layer protocols instead being used for domain-specific purposes, e.g. Simple Mail Transfer Protocol (SMTP) for sending emails or Domain Name System (DNS) for resolving domain names to IP addresses.

A HTTP request always includes a URL and HTTP method. These are used to identify a resource and what action to perform on it. Additionally, one or more header fields can be added to the request to modify how the request should be processed. A body may also be included to hold data that should be included with the request. Similarly, a HTTP response always includes a status code, indicating how the request was processed.

A typical HTTP request and response will consist of a client requesting the server for some service or resource, passing any parameters for the action using either the message body or the URL itself, and using the headers of the request to pass additional information on how to parse the request, e.g. authentication information or the requested format of the output.

On receiving the request, the server will attempt to process the request. Should the request be valid and processed with no errors, it will respond with an acknowledging status code to the client, potentially with a service result or requested resource. Should the request fail to be processed, e.g. because of an invalid request or because the server cannot process it as expected, the server may provide a status code indicating what went wrong.

It is common for communication security to be implemented at the application layer. In the case of HTTP, this is done by extending the HTTP protocol with TLS encryption, denoted "HTTPS". Doing so provides encryption of all HTTP messages at the cost of increased processing time for each HTTP message due to the need for encrypting and decrypting them.

2.3 Service-Oriented Architecture and Web Applications

Service-oriented architecture (SOA) is a software design type used for structuring communication between distributed components. SOA is characterised by the encapsulation of functionality as services, which are then provided to other components through a communication protocol [30]. This provides the benefit of abstracting the implementation details from other components; other components only need to know how to request services and what response to expect, not how the service results are computed. This can be contrasted to e.g. RPC where the logic of the system may be fully exposed.

While SOA may be used to describe any network-based system that offers functionality through services, it is commonly used in the context of the World Wide Web. In this context, the term "web services" may be used to refer to the services offered by the system and "web applications" to the systems that offer them. Commonly, this will be achieved by defining an interface for the application, known as a "web API", which defines services offered and how to request them.

While the procedure for making requests to a web API is dependent on the protocol used, some features are shared across protocols, notably endpoints. Each endpoint specifies the location of a specific service on a server. For web applications, endpoints are most commonly defined using a Uniform Resource Locator (URL), in which case each service corresponds to a single URL. However, the endpoint can also be encoded in the request, which may then be made to a single URL used for any request.

Typically, web APIs are considered to follow one of two architectural types: RPC-style or RESTful [49].

2.3.1 RPC-style APIs

RPC-style web APIs are based on the concept introduced in section 2.1.0.1. Each request made by a client corresponds to a single procedure call, which ends with a response from the server.

Any server may define its own communication protocol for how services should be requested and what response should be expected. However, it is common to use an existing protocol that implements fundamentals, e.g. how a message should be formatted. Traditionally, Simple Object Transfer Protocol (SOAP) has been the standard protocol for web services [3]. The protocol defines message structure, rules for encoding of data and a convention for how to represent the actual procedure call and response.

However, SOAP has been criticised for being overly verbose and complex [44], and for requirement of using XML for messages [3]. Several alternative RPC protocols have been developed since, notably JSON-RPC, which uses a more lightweight format with fewer requirements, and gRPC, which uses a stricter specification for encoding of data and provides tools for serializing and deserializing the content of messages for various programming languages according to the rules of the protocol.

2.3.2 RESTful APIs

Representational State Transfer (REST) is a specification for the design of web APIs, with web APIs conforming to the requirements being denoted "RESTful". Developing web APIs according to the principles of REST has arguably become the industry standard. Therefore, it is also relevant to examine the importance of conforming to a web API when the goal of the thesis is to generate both the web API and API client library.

The constraints of REST are:

- **Uniform interface**: Resources must be identified using a URI, the action to perform on the resource is represented by the HTTP method, and all messages should be parseable based on the content of the message only. Finally, each response should also include the actions now available to the client, known as Hypermedia as the Engine of Application State (HATEOAS).
- **Client-server independency**: Clients and servers must be able to evolve independently of each other.
- **Statelessness**: The server holds no state information on the clients. Each request made by the client should be treated independently of previous requests.
- **Cacheability**: Resources provided by a server must be marked cacheable when possible, allowing clients to improve performance by caching responses and using these when repeating requests.
- **Layered systems**: Clients must not need to know if they are communicating directly with a server or through an intermediary.

The Richardson Maturity Model [14] can be used to determine how well a web API satisfies the constraints of REST (see Figure 2.1). According to this model, web APIs can reach four levels of RESTfulness maturity. At the first three levels, the web API is in essence still a RPC-based interface, albeit satisfying a number of RESTful constraints that simplify development and maintenance of the web API. Firstly, using multiple URLs to identify resources instead of using a single endpoint and encoding the information in the request body. Secondly, using HTTP methods to represent the action to carry out on the resource.

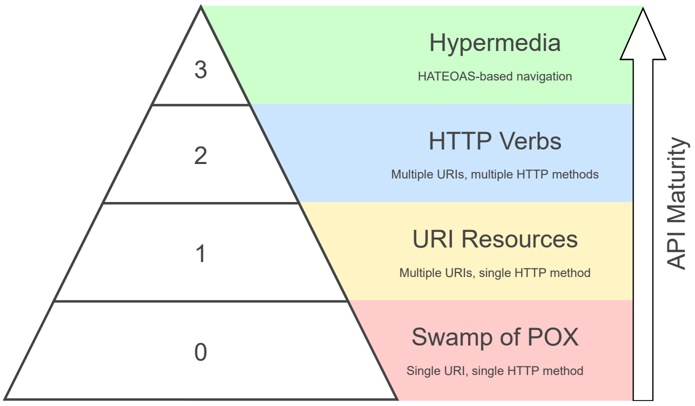
3
Hypermedia
HATEOAS-based navigation
2
HTTP Verbs
Multiple URIs, multiple HTTP methods
1
URI Resources
Multiple URIs, single HTTP method
0
Swamp of POX
Single URI, single HTTP method
API Maturity

Figure 2.1: Richardson Maturity Model, describing the recommended steps to satisfy the REST specification

However, at this level, the client still relies on knowing the web API structure beforehand, and changing the web API may break the client. Only upon introducing HATEOAS are the client and server truly decoupled, and a RESTful web API is actually achieved.

One of the major benefits of using REST for web APIs is client-server decoupling, with HATEOAS allowing servers to evolve independently of clients. A resource that is no longer exposed or changes address can simply be rediscovered by the client. By contrast, a client accessing an RPC-based API expects a service to be available at a specific endpoint, and may not function correctly if the resource has been modified, e.g. by changing the address or response format of the resource.

In practice, however, very few web APIs are actually designed around HATEOAS, and thus do not gain the benefits from using this concept [22]. This may because of the complexity in developing HATEOAS-based web APIs or because of a lack of tools.

Thus, while designing web APIs to be RESTful may be considered the industry standard, it is rarely achieved in practice, and thus conforming to the principles of REST is mostly useful in aiding client application developers in discovering and using the web API.

2.3.3 Naked Objects and Object-oriented User Interface

Naked Objects and Object-oriented User Interfaces are two related concepts, both of which revolve around applications that are modelled fully around domain objects, even in the user interface.

In object-oriented user interfaces, the user interface of the application must directly represent the domain logic objects that the user is acting on [5]. Thus, rather than designing the user interface to invoke methods which modify the object attributes, the user modifies the object attributes directly.

The naked objects pattern extends object-oriented user interfaces by requiring that the user interface should be generated automatically from the domain objects [28].

The naked objects pattern is interesting in relation to this thesis, because it encourages an approach similar to that of this thesis, i.e. focusing on domain logic and automatically generating components for exposing it. However, requiring the components to be automatically generated imposes strict constraints on the developer, and it is therefore especially interesting to investigate the effectiveness of the pattern in relation to the developers using it and how they react to it.

R. Pawson has researched the use of the pattern extensively, notably conducting a study of the use of naked patterns in the Irish Department of Social Protection [29]. In this study it was found that, despite the strict constraints imposed by the pattern on user interface design, developers reported that the pattern improved the feeling of flexibility in development. The study concludes that as developers get used to the imposed constraints and think purely in terms of domain logic, they become more efficient.

This conclusion is in support of the intended approach encouraged by the thesis project where developers are not allowed to implement the web API or API client library manually.

However, the pattern has also been faced with criticism, mainly related to the use of object-oriented user interfaces [6], which have been criticized for not being adequately usable by humans.

These issues may also be present in the similar approach of using "application generators", applications used for generating other applications, e.g. JHipster [18]. The main benefit of using these generators is that they facilitate rapid development, as a high amount of the code is automatically generated. However, the automatically generated user interface may face issues similar to those described above. Moreover, modifying the application after generation may carry a high overhead, as developers must now understand the generated code.

2.4 Domain-Driven Design (DDD)

As Domain-driven design (DDD) is central to the problem statement of the thesis, a brief introduction will be given. Unless stated otherwise, the theory presented in this section is based on the book "Patterns, Principles and Practices of Domain-Driven Design" by S. Millett and N. Tune [24], which in turn is based on the book "Domain-driven design: tackling complexity in the heart of software" by E. Evans [12].

DDD is a software development methodology which seeks to enable effective development and maintenance of software for complex business domains, i.e. the area in which the problems that the software relates to reside.

At the centre of DDD is the domain model. The domain model is an abstraction of the domain, including only the aspects that are necessary to solve problems in the domain. In DDD, a central task of development is iteratively improving the domain model, continually identifying the aspects of the domain model that must be refined.

Development of the domain model requires an understanding of the related domain, which a developer may not have. Instead, domain experts are included in the development process to give input on the domain model.

Developers and domain experts may encounter scenarios where they disagree on the meaning of a term, or where one does not understand a term the other uses. As a solution, a ubiquitous language is developed along with the domain model, serving as the "shared language" of the developer and domain experts.

Developing a single, unified domain model with an associated ubiquitous language is often infeasible. Multiple teams may work on the same domain model and want to make changes independently, leading to multiple versions of the model or slowing the process of making changes. Similarly, when the model grows, the ubiquitous language becomes difficult to maintain, e.g. because of repeated terminology with different meanings in different areas of the model. As an example, in a banking application, the term "account" may be used to refer to either a bank account owned by a customer or a user account, allowing a user to access the services of the application.

To solve this, sub-domains are identified and bounded contexts are defined along their boundary. Each bounded context uses a context map to explicitly define the relationship between bounded contexts and how they may interact. Bounded contexts may then be developed independently of each other, as long as the context map is respected.

2.4.1 Domain-Driven Design in Practice

Domain models are conceptual, identifying the domain concepts that are relevant to problem-solving and their relationships. However, to be shared, the model must be expressed in a physical format, e.g. as a visual model or description. However, the end goal is to develop a system that expresses the domain model, potentially supported by other formats as documentation, e.g. visual models or textual descriptions.

DDD identifies several artifacts used to express the domain model in code, notably entities as objects with an identity, immutable value objects as objects identified by their attributes and aggregates as collections of objects bound by one entity. Based on the artifacts of DDD, the domain layer is built, capturing the concepts, relationships and rules of the domain model.

To avoid other systems having direct dependencies on the domain logic, an application layer is defined, containing application services. This layer is used to define the interactions that other systems may have with the system. As an example, an application that wishes to login a user should not manually check if the user exists and if the password is correct. Instead, an application service should define an interaction for logging in a user, with the domain logic for checking for user existence etc. being contained in the application service.

This leads to a layered architecture, nicknamed "onion architecture" (see Figure 2.2, where the domain layer is accessed by the application layer, which in turn may then be accessed by outer layers, e.g. a web API for exposing the application services.

The assertion that application services are used to expose the functionality of a system and that a web API simply serves as a method of exposing the application services is part of the motivation for this project. Instead of requiring the developer to manually implement the web API to mirror the application services, the application services could be "translated" into the web API.

2.4.2 Related Architectural- and Design Patterns

2.4.2.1 Model-Driven Engineering (MDE)

MDE is a software development methodology in which software development is firstly considered to be the development of models. These models may be expressed in multiple forms, e.g. using visual modelling languages, e.g. UML, domain-specific languages or as executable code.

The rationale behind using this approach, rather than simply writing only the executable code, is that models expressed using third-generation languages are not as expressive as other model formats. Using domain-specific languages or

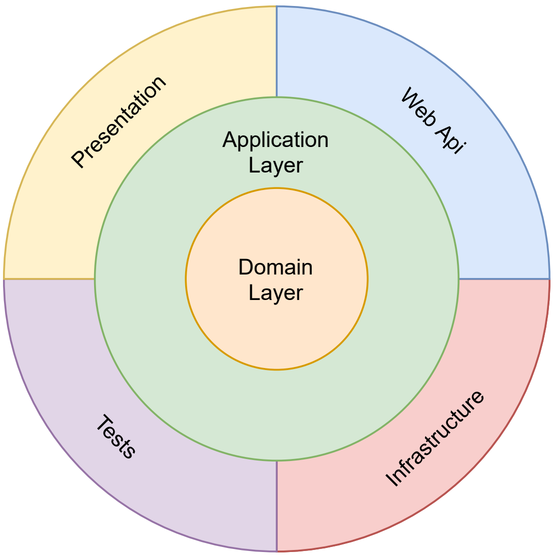

Figure 2.2: Onion architecture model.

visual model languages makes it easier to both understand and develop the models [33].

Both MDE and DDD highlights the importance of using domain models, and both considers the primary task of software development to be the development of the domain model. However, MDE concerns itself less with how to create good domain models, instead focusing on how to translate the model into other formats, including executable code (for more on translation into executeable code, see section 2.6.1.2).

2.4.2.2 Microservice Architecture (MSA)

MSA is a relatively new architectural pattern in which web applications are built from small, autonomous services that are able to work together [25]. This can be contrasted to monolithic services, where all services are provided by one program.

In DDD, bounded contexts are used to "partition" the domain model into the relevant areas with well-defined interactions between the contexts. Similarly, applications in MSA are lightweight and responsible for tasks within a smaller area and communicating with each other through interfaces without concerning themselves with the internals of other services. Thus, DDD is useful for identifying bounded contexts, which may then be translated into relevant microservices [31].

2.5 Developer Experience

Developer Experience is a relatively novel concept inspired by the field of User Experience. Fagerholm and Münch [13] proposed the definition of the field in the article "*Software Developers as Users: Developer Experience of a Cross-Platform Integrated Development Environment*" describing developer experience as "*a means for capturing how developers think and feel about their activities within their working environments*". Compared to software development methodologies, which focuses on how to efficiently develop well-functioning software, developer experience concerns itself with how to improve the experience of creating software.

The article divides the experiences of the developer into three categories, the first one of which is named "development infrastructure", referring to tools, technologies and methodologies used in software development. While existing research has largely focused on tools used in the development process, e.g. IDEs [20] and online collaboration environments, [27], the category also includes the tools used the in software itself, e.g. libraries and frameworks.

This concept is considered relevant to the thesis because the thesis seeks to develop a library that may affect how developer work with web applications. Therefore, it is relevant to consider not only how the library affects developer efficiency, but also how the library affects the developers' feelings on web services development.

2.6 Related Work

In a study by Tan et al. [43], the web service research community trends were examined. Based on the findings, the authors suggested that future research should be focused on pragmatic solutions that value practical approaches to real-life problems, criticizing the use of formal methods to solve problems that practitioners rarely encounter.

Several informal articles authored by industry practitioners share a similar opinion, criticizing the output of research in software engineering for being either too complex for common use, or solving problems that are largely irrelevant to practitioners [21, 36, 48].

According to the study by Tan et al. [43], software engineering has largely been shaped by industry practices, rather than academic research, and thus the state of the art in this field is highly practical, intended to address the issues that practitioners often face. Thus, the most commonly used solutions are pragmatic and no more complex than necessary.

As a contrast to this study, B. Selic [35] argues that limiting research to practical solution is detrimental to the field as a whole. B. Selic claims that software engineering has suffered from conservatism in research, where practical solutions for assisting in manual development of software is prioritized over automation of the development process itself.

These articles, among others, have influenced the related work that was considered most relevant to the thesis. The thesis is intended to fit into existing research, exploring the area of code generation in web services. However, the solution proposed in the thesis should also be practical and solve an actual real-life problem.

For this reason, two types of related work will be presented:

- Academic research into approaches to code generation.
- Industry standard approaches of web services development.

2.6.1 Existing Research

2.6.1.1 Automatic Programming and Code Generation

Automatic programming refers to programming at a higher abstraction level to generate code at a lower abstraction level [2]. Originally used in reference to the use of compilers to translate higher-level languages into assembly code, the term has over time come to refer to the general idea of programming at a higher level than what is otherwise the norm.

In a modern context, the term may be used to describe methods for developing software where part of or all of the code of a system is generated automatically based on some high-level input, e.g. a model expressed in a human-readable format.

While it is possible to implement a compiler that generates an executable program directly from high-level input, it is more common to instead generate source code for a high-level language and then using a compiler for this language to generate the executable program [47].

Because high-level languages already provide abstraction over lower level languages, it can be argued that generating high-level code from even higher-level input is superfluous. However, while high-level code may provide abstractions of lower-level functions and computer architecture, it still requires the developer to concern themselves with how to achieve certain functionality, rather than allowing them to focus on what functionality should be achieved. Using a domain-specific language or visual modelling language to express a model and translating it into code has been shown to provide several benefits of expressing the model directly in high-level languages [32].

Firstly, generating the code allows code template to be used. By using the same template for similar components and enforcing constraints in it, it can be ensured that every component is implemented in the same way and satisfies the same constraints, providing standardization of the source code. However, this can also be seen as a disadvantage, as requiring the same implementation to be used everywhere reduces flexibility. If a template does not support a specific case, the entire template must be modified rather than just that case.

Secondly, employing code generation may reduce the time to develop and modify systems, especially larger systems with often-repeated, similar sections of code [32], colliqually referred to as "boilerplate code".

The approaches to automatic programming that are most relevant for this project will be presented in the following sections.

2.6.1.2 Model-Driven Engineering (MDE)

As explained in section 2.4.2.1, MDE is a software development methodology that focuses on the development of models and translation into other forms, e.g. executable code.

According to B. Selic [35] in his article "The pragmatics of model-driven development", also referenced in Section 2.6, different implementations of the same model would ideally be translatable back-and-forth. As an example, a visual model should be translatable into code, which should then be translatable back into the same visual model. The translator that converts the visual model to code may then be considered both a model transformer and code generator.

In the same article, B. Selic claims that this approach to software development has gained little popularity, mainly due to a lack of tools for generating and verifying code from models.

As this article is from 2003, it is interesting to investigate if this claim still holds true today. In a study by Sebastián et al. [34], the prevalence of academic papers investigating model-driven architecture (MDA) and code generation was investigated. While not equivalent to model-driven engineering, MDA is closely related to MDE, being based on the same concept of model transformation.

In this study, 2,145 MDA-related articles were identified, 50 of which also used code generation. The paper concludes that MDA is still highly relevant, but that most of these articles are related to transformation of models without generation of code. MDA-based code generation is, however, still a researched topic, especially in the fields of mobile- and web development.

This thesis does not concern itself with the transformation of models, instead focusing on how to generate code for exposing the domain logic that expresses the model. This domain logic, however, may be generated through transformation from a different model type. It may be interesting to investigate how to incorporate the concepts of distributed systems development, notably the exposure of domain logic through a web API, into the model transformation concept of MDE.

2.6.1.3 Generative Programming

Generative Programming is an approach to automatic generation where programs are assembled from a series of existing component and a high-level specification of the desired functionality [9]. This can be contrasted to "one-of-a-kind" systems, where a developer is responsible for the manual assembly of components to develop the system.

For generative programming to be feasible, the following requirements must be met:

- The components used must be suitable for automatic assembly by following a standard architecture for generative programming [9].
- Some mapping from high-level requirements to required components must exist [9].

Generative programming is interesting in relation to this thesis, as it could be argued that it supports the thesis motivation. Generative programming does not require that individual components are automatically generated, only that they are developed for reuse and that the assembly of them to form the final program should be automated.

It could be argued that a library for automatically generating a web API and associated client library could be considered a component for use in the "assembly" of a client-server system. Rather than manually writing the endpoints for a web application server and the methods for requesting the services, this component could be "plugged in" to automatically provide this functionality.

2.6.2 Industry and Open Source Projects

As mentioned in Section 2.6, the field of web services development may be characterised as pragmatic with a high demand for solutions that address issues commonly faced by developers. Several industry standards and open source projects for addressing these problems exist, and will therefore be presented here.

2.6.2.1 API Client Libraries

Commonly published by organisations that benefit from a rich ecosystem of third-party clients, API client libraries are libraries published by web service providers that can be imported and used in client applications. Examples of organisations that offer API client libraries include Google[23], Slack[37] and Discord[10].

By providing an API client library, the provider of the associated web services can simplify the development of client applications requesting these services. Using the libraries, developers can treat the web API endpoints as method calls, rather than having to set up the network calls.

However, providing the API client libraries requires them to be developed and maintained for each language that is to be supported. This disadvantage can be mitigated by automatically generating the API client library to mirror the Web API, requiring only a template to be developed.

Ideally, client libraries abstract the network-based nature of Web APIs, instead emulating making a method call to receive some resource. However, distributed systems are inherently different from local programs[46], and thus it is necessary to consider several potential abstraction leaks:

- **Latency**: Network requests are inherently slower than method calls due to the multiple devices involved and the greater distance travelled. As a result, method calls with network requests will most commonly be asynchronous, allowing the application to work on other tasks while waiting for the response.
- **Network Failure**: Network requests may fail, and thus method calls with network requests must be prepared to handle requests that time out.
- **Server Exceptions**: In some cases, the server may not accept the request, e.g. because it is not authenticated as required or because the parameters provided are invalid. In these cases, the client application must be prepared to handle an unexpectedly refused request.

As a result of these consequences, API client libraries are limited to providing methods that "wrap" the network request, potentially providing features such as static checking to catch illegal parameters at compile time or at run-time before the request is made. However, callers of these methods must still compensate for the potential issues mentioned above.

2.6.2.2 Specification Based API Tools

Specification-based approaches to generation of API client libraries is popular in web services development. In this approach, a specification of the web API is written and used to automatically generate related resources. These include API client libraries, documentation of the web API describing the available services, and web API code stubs. Examples of these tools are the OpenAPI Generator[26], Swagger Codegen[38] and NSwag [42].

The approach encouraged by these tools differ from that explored in this thesis in that the specification is actually independent of the actual web API, and thus the two must be maintained to remain consistent. The web API specification specifies what services should be published and how, but the task remains to ensure that the web API remains consistent with the specification. Moreover, the task of developing the web API and linking it to the underlying server logic is also left to the developer.

A variation of this approach is to generate the specification from the web API, [16]. This ensures that the specification remains consistent with the web API, but still doesn't solve the problem of linking the web API to the underlying server logic.

By comparison, this thesis seeks to explore an approach that maintains a single source of truth, i.e. the server logic. While this provides less control of how the service logic is published, it also ensures that the web API is always consistent with the underlying server logic.

For a comparison between the different approaches, see Figure 2.3.

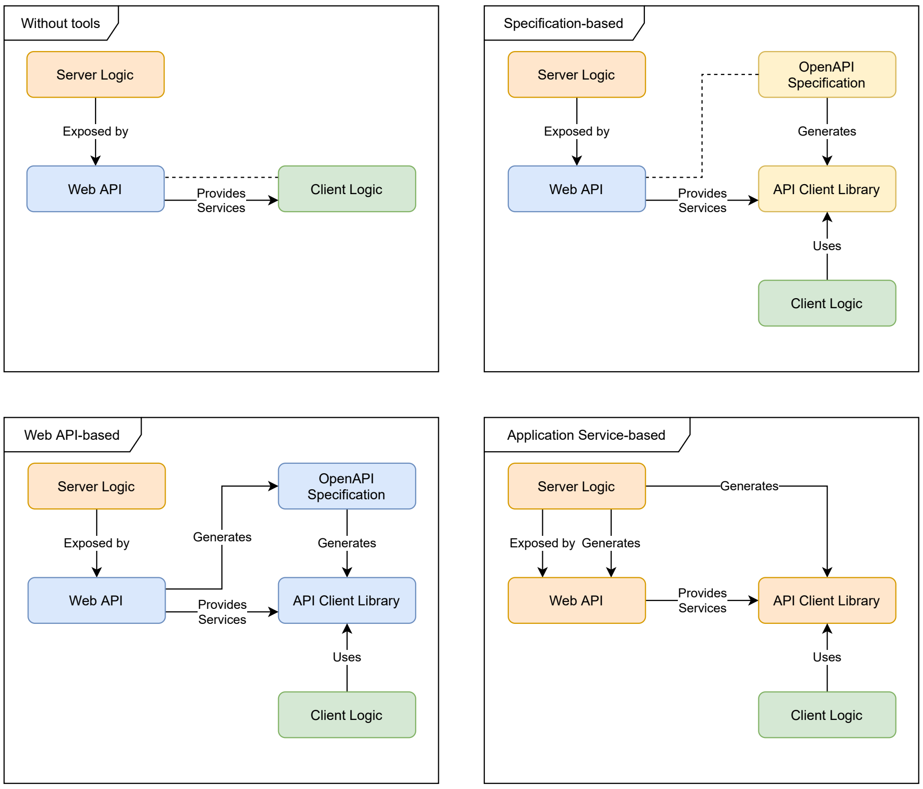

Figure 2.3: Comparison between different approaches to web services development. A dotted line indicates that the two artifacts are loosely coupled, yet must be kept consistent. Color groups indicate that all artifacts are generated from one source.

Chapter 3

System Design

In this chapter, the design for a library that solves the problem statement of the thesis is proposed. Firstly, an analysis of the requirements for the library is presented. Secondly, a design based on the identified requirements is presented.

3.1 Requirements Analysis

The purpose of this section is to identify the required functionality of the library in order to satisfy the problem statement. This will be done in three steps:

- Identify design constraints.
- Identify artifacts to generate.
- Identify the functionality that these artifacts must have.

In addition to identifying required functionality, non-essential functionality will also be discussed, with a justification for why this functionality was not included in the design.

3.1.1 Design Constraints

3.1.1.1 Assumptions

For the design, three assumptions for the systems using the library will be made.

Firstly, it is assumed that the server runs a web application using a HTTP server engine to handle incoming requests, and that these requests can then be routed to the code block or method responsible for handling it. This is the case for most web application frameworks, though the form in which it is implemented varies.

In such a web application, it is also assumed that requests may be routed through multiple code blocks in a pipeline-like manner. As an example, a request may first be intercepted by a code block for logging the request, then a code block for authenticating the request before the code block responsible for routing the request is executed.

Examples of web applications that support this behaviour are Ktor, which refers to the concept as "installing features into the application pipeline" [41], and ASP.NET, which refers to the concept as "installing middleware into the app pipeline" [1].

These assumptions are important because it allows functionality to be installed in multiple places, rather than requiring all functionality to be installed in one code block.

Secondly, it is assumed that the client application makes requests to the server using a configurable HTTP client which, in the same manner as with the server, can route outgoing requests or incoming responses through multiple code blocks. As before, this assumption allows the functionality to be installed in multiple places. Notably, the client application may want to set specific HTTP request headers which can be configured for the HTTP client itself rather than for each request.

Examples of HTTP clients that support this behaviour are the Ktor Http-Client [39] and the .NET HttpClient [7].

3.1.1.2 Configurable Functionality and Endpoint Functionality

One of the constraints of generated code is that the implementation is constrained to that defined in the template. If a template is modified to add some functionality, it will be applied to all instances of generated code. If this functionality is only required in some cases, the code generation must be configurable to allow adjustment of what code should and should not be generated.

Instead of modifying the template as necessary, it is desirable to instead let as much functionality as possible be installed through configuration of the web application or HTTP client, especially if the functionality is required for every instance of generated code in the same manner. Thus, functionality should only be present in the template if it is absolutely necessary and cannot be implemented through configuration.

3.1.2 Artifacts

3.1.2.1 Web API Endpoints

The web API endpoints are used for exposing the application services. Each endpoint, which in this context is to be understood as a single method or code block that is executed when a request to the associated URL is made, is associated with the method of an application service.

The content of the endpoints should, at a minimum, consist of a method call of the associated method. This, in turn, requires a reference to an instance of the application service containing the method. The project integrating the library is therefore responsible for ensuring that the endpoints have a reference to the application services to be used in the endpoints.

The intention of generating the endpoints is that developers should only have to concern themselves with how the domain logic is exposed through the application services, not how the application services are exposed through a web API. Once the application services have been defined and annotated, the web API should be automatically generated to always match the application service.

It could be argued that the owner of the system publishing their application services may want control over the methods that are published by the API, e.g. by allowing only certain methods to be published. However, as the intention of application services is to serve as the system interface, requiring only certain methods to be exposed indicates that the application service should instead be modified.

3.1.2.2 API Client Library

With the web API generated, client applications will be able to make requests to the server, given that they know the request format.

However, this requires developers to manually write and maintain the methods for making requests to each endpoint. Instead, these methods may be generated, too. Given that the structure of the endpoints and the application services are known beforehand, generating methods for making network requests to each endpoint is relatively simple.

Similarly to how an endpoint is generated for each method in the application service, the API client library should contain a method for each endpoint / application service method. Each method should, at a minimum, make a request to the server and parse the response.

Because the generated methods have the same function signature as the application service used for the generation, static typing will be available to ensure that the parameters passed and result returned is the same as in the web application.

It is worth noting that the URL of the server hosting the web API is not known at compile-time. Therefore, this must either be configured as the base URL in the HTTP client or passed as a parameter to be prepended to the URL at runtime.

3.1.2.3 Documentation

In a specification-based approach to web API generation, it is common to also generate documentation for the web API. This may provide a human-readable description of the web API and instructions on how to use it, assisting developers of applications that consume the Web API.

However, the intention of this library is that the web API and API client library should reflect the application services. Thus, documentation of the application services should be sufficient documentation of the web API and API client library as well.

3.1.2.4 Server Engine / Client

During the design process, it was considered to expand on the project scope from simply generating the endpoints and API client libraries to generating everything necessary to facilitate communication between the server and client.

The assumption behind this idea was that some developers would be interested in only having to write domain logic and letting the library configure everything related to networking, i.e. starting and configuring a web application engine for the server as well as a HTTP client for the client.

However, this was ultimately excluded from the project based on the following arguments:

- Modifying the web application configuration is an infrequent task as it is less susceptible to changes.
- The application server configuration is defined once only, making it ill-suited for code generation, which is instead more suited for similar, repetitious code.
- The configuration of the server may vary greatly, making it an ill-suited task for a library that does not know of the individual project needs.

3.1.3 Generated Code Functionality

As mentioned in section 3.1.1.2, the functionality in the generated code should be as minimal as possible to avoid cluttering the template with functionality which is only occasionally required.

The functionality discussed in this section was identified as either being required for the web API / API client library to function or as a notable optional feature that cannot be installed through configuration of the web application or HTTP client.

3.1.3.1 Routing

In this context, routing will be used to refer to the actions to take when a request to a specific URL is received by the server.

This functionality is the core functionality that the library seeks to offer, as it provides a mapping between the URL of requests to the methods of the application services of the system.

This functionality may be implemented by generating a method or code block to execute when a request to a given URL is received. Inside the method, the relevant underlying server methods are called and a response is returned.

Similarly, a corresponding client method should be created for each endpoint, containing the code for sending a request to the corresponding endpoint and parsing the response.

This functionality must be generated as the content depends on the method to call.

3.1.3.2 Serialization

In order to send parameters with the request, it is necessary to serialize them. Similarly, when receiving the request, it is necessary to deserialize the request parameters. Thus, some serialization logic must be in place in the generated code, instructing what object to serialize to and from.

The serializer in the endpoints and client methods does not necessarily need to contain the logic on how to serialize the object. Instead, this can be defined in a type-specific, format-specific serializer. Thus, the serialization logic in the generated code can be minimal, simply containing instructions on what objects to serialize from and to, not how to do so.

3.1.4 Configurable Functionality

The functionality in this section, while potentially important for the system, was considered unsuitable to be included in the generated code, and should instead configured in the web application.

3.1.4.1 Encryption / HTTPS

For HTTP, encryption may be implemented by using the HTTPS protocol. Doing so requires having a SSL certificate, which identifies the server, and for the application knowing the location of it. Furthermore, it can be argued that HTTPS will typically be enabled for the entire application, not only for parts of it, supporting the argument that including encryption functionality is the endpoints is unsuitable.

3.1.4.2 Exception Handling

In a local system, exceptions may be handled by propagating them through the call stack until it reaches a block of code suitable for handling it. As an example, an exception thrown due to invalid user input may be propagated until it reaches the presentation layer, where the user may be warned of the error.

Distributed applications provide a challenge in exception handling as the exceptions cannot inherently be propagated between components. It could also be argued that doing so is not suitable, as the execution flow of the components should be independent. Instead, the exception must be translated into a response that informs the client of the error that occurred and how to act on it. In web services, the web application can be configured to catch exceptions and use a mapping to determine the status code to return.

For this library, two options for handling exceptions were considered. The first is to simply allow the developer to configure exception handling as described above. The second option was to serialize the exception thrown by the server, sending it in the response, deserializing it again in the client and throwing it. If the client and server are implemented in the same language, this would allow the exception to essentially be propagated from the server to the client.

However, it could be argued that in most cases, the exception is not relevant to the client. The client does not need to know how the server failed, only how it should act next. Moreover, providing exceptions to the client may not be desired, e.g. if the exception contains details that should not be leaked to the client. Finally, the client may not be able to parse the exception, e.g. if it runs on a different platform. Based on these arguments, the first approach was chosen instead.

3.1.4.3 Caching

Caching allows responses to be stored in a cache, allowing them to be reused. This may be implemented in the server, e.g. by storing the response for certain requests in a cache, avoiding having to recompute them, or in the client, e.g. by storing the received response for a request, avoiding having to resend it.

Caching is commonly implemented using a dictionary-like cache structure with additional rules to limit memory use and ensure freshness of data, e.g. expiration or prioritization based on the frequency of requests. Because caching can be implemented in this manner, it is not necessary, nor suitable, to implement caching in each endpoint or client method.

3.1.4.4 Logging

Logging allows monitoring of the operation of the system. This is most commonly implemented in the web application configuration, and is therefore not considered relevant for the library.

3.2 Proposed Design

From the requirements identified in section 3.1, a design was made for a library for automatically generating the endpoints and API client libraries. The library has the following functionality:

- Generation of server endpoints for exposing application services.
- Generation of an API client library for making requests to the aforementioned endpoints.

The library is used by annotating application services to indicate that they should have artifacts generated for them and including the library in the compilation process.

During building of the web application, an annotation processor is used to find all classes "tagged" for artifact generation. Introspection of the classes is used to generate type specifications of the application services. These type specifications are in turn used for generating the artifacts, which are finally compiled. For an overview of this process, see Figure 3.1.

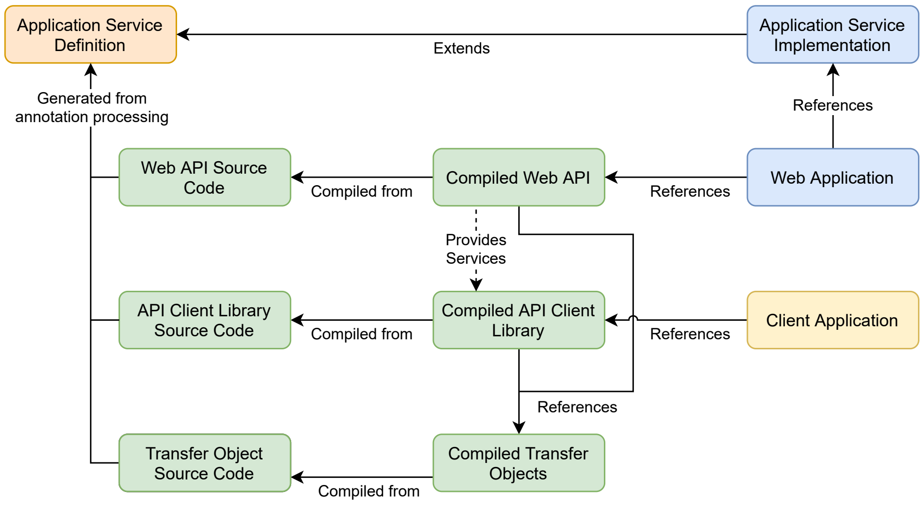
Application Service Definition
Extends
Application Service Implementation
Generated from annotation processing
References
Web API Source Code
Compiled from
Compiled Web API
References
Web Application
Provides Services
API Client Library Source Code
Compiled from
Compiled API Client Library
References
Client Application
References
Transfer Object Source Code
Compiled from
Compiled Transfer Objects

Figure 3.1: Graph showing the generated artifacts and their use in a project.

The generated server endpoint classes are built with the following structure:

- For each application service, build a class for hosting the endpoints, the constructor of which takes the application service as a parameter.

- For each method:
  - Await request at addresses specific to the corresponding methods of the endpoint.
  - If the method has been annotated to require authentication, attempt to authenticate the request.
  - Deserialize the function parameters in the request, if any.
  - Call the method corresponding to the endpoint using the provided application service.
  - Serialize the method return value, if any, and send a response to the client.

The API client library is constructed in the following:

- For each application service, build a class for hosting the client methods, taking a base URL as a parameter.
- For each method:
  - Create a method with the same signature as the application service method.
  - In the method, serialize the parameters, if any, and make a request at the address specific to that method appended to the base URL.
  - Await the response.
  - Deserialize the response.
  - Return the result, if any.

Any additional functionality required on the server's end, e.g. caching or logging, must be implemented by installing it in the web application pipeline. Similarly, additional functionality required by the client must be implemented by configuring the HTTP client.

Chapter 4

Implementation

In this chapter, a proof-of-concept implementation of the proposed code generation library, nicknamed "Kodegen", will be presented.

Firstly, an overview of the implementation is given. Because the implementation will mostly reflect the proposed design, the overall implementation will not be explained in-depth; instead, the overall structure of the project is presented.

Secondly, the third-party frameworks and libraries that were used for the project will be presented and their inclusion justified.

Thirdly, an introduction on how to use the library, including examples of use, is given.

The source code for the proof-of-concept implementation is publicly available on Github [4].

4.1 Overview

The proof-of-concept implementation functions through annotation processing. Annotations, which serve as a way of adding metadata to elements in Java and Kotlin, may be added to application service types, indicating that they should have code generated for them (See Figure 4.1 for an example). An annotation processing tool may then be used to scan for any type with specific annotations and use the type signature to generate code.

The logic for code generation is contained in the `KodegenProcessor` class. This processor extends `AbstractProcessor`, a base Java class with methods that facilitate annotation processing. Thus, the `Kodegen`

`Processor` serves as the entry point for code generation. Moreover, two annotation classes are defined: `ApplicationService`, to mark types which should have code generated for it, and `RequireAuthentication`, to mark methods that should require authentication.

On compilation, an annotation processing tool is used to provide the `KodegenProcessor` information on all types annotated with `ApplicationService` and all methods annotated with the `RequiresAuthentication` annotations.

For each `ApplicationService` type, a function responsible for generating the relevant artifact is called: `generateServerEndpoints`, `generateClientMethods` and `generateTransferObjects`.

Each of these functions uses a DSL-based template to generate a file specification (see Section 4.2.4 for details), combined with inspection of type information to fill out the specification. This results in a file being generated for each artifact per annotated application service. These artifacts can then be used directly in compilation of the project or distributed and imported in other projects.

The library uses third-party libraries for the generated artifacts, which must therefore also be available in any other project using the library. For this reason, the library uses transitive dependencies for the relevant third-party libraries. While it is preferable to avoid doing so, not making the dependencies transitive will cause issues while generating code, as the library will then attempt to use libraries that are not available.

While the library can be used in any project setup that supports annotation processing, it is highly recommended to use it in a Maven or Gradle project. Firstly, this allows the annotation processing task to be declared a dependency of a project, ensuring that code can be generated before any other code with dependencies on the generated code is compiled. Secondly, it allows for automatic resolving of the dependencies used by the library.

4.2 Technologies Used

In this section, the technologies used in the project, e.g. programming languages and third-party libraries, will be presented and their use justified. This will be done by explaining what feature is provided by the library and comparing it to potential alternatives.

4.2.1 Programming Language - Kotlin

For this project, Kotlin was decided as the programming language for the implementation of the project. This decision was made based on the following points:

- Kotlin offers experimental multiplatform compilation, which allows Kotlin code to compile into Java bytecode, JavaScript or native code. This makes Kotlin highly relevant for projects that are expected to run on several platforms, e.g. client-server applications where the client may be a website (JavaScript), a mobile application (JVM / native) or a desktop application (native). As explained in Section 1.1, one of the main problems associated with API client libraries is the need to maintain them for multiple languages. However, with multiplatform support, the API client library could be developed just once in Kotlin and then used in other projects, regardless of the platform.

- Kotlin is interoperable with Java code, allowing Java libraries to be used for Kotlin projects. Java, in turn, has rich support for annotation processing, which is therefore also available to Kotlin.

4.2.2 Annotation Processing - Kapt, Java Annotation Processing & Java Language Modelling

Annotation processing in Kotlin is supported using the "Kapt" plugin. Kapt generates the environment for annotation processing, essentially serving as an entry point for annotation processors by granting them access to the source code.

Kapt supports annotation processors implemented using Java Annotation Processing, located in `javax.annotation.processing`. By implementing a class inheriting from `AbstractProcessor`, the processing environment is made available, allowing inspection of files by scanning for annotations.

Because the annotation processor inherits from `AbstractProcessor`, the type introspection provided by the processing environment is also provided using the interfaces of the `javax.lang.model` package, which is used for modelling the Java language.

However, this is a source for problems in Kotlin projects. Prior to annotation processing, Kapt will generate code stubs to use for introspection. However, these code stubs are generated as Java files, and thus Kotlin types are also converted to Java types, e.g. `kotlin.String` to `java.lang.String`. This will cause the annotation processor to generate code using Java types instead of Kotlin types, leading to incorrect code.

To circumvent this problem, the annotation processor also uses the KotlinPoet Metadata API [17] to inspect the metadata provided for each class along with the code stub. Kotlin types are preserved in this metadata, allowing for the generated code to use the correct types.

4.2.3 Web Application Framework - Ktor

The web application framework is responsible for providing essential web application functionality, notably routing based on the URL of the request. While this is achievable without a framework, doing so would require manually implementing this functionality, which is not suitable for a proof-of-concept project. Thus, an existing framework was used. Two application frameworks were considered:

- Spring Framework [45], a mature Java framework.
- Ktor [40], a more recent Kotlin framework.

For this project, Ktor was chosen. This choice was made with the following supporting reasons:

- Ktor is less complex, making it more suited for a proof-of-concept.
- Ktor is a Kotlin framework, supporting several Kotlin features such as multiplatform compilation and Kotlin coroutines.
- The Ktor lifecycle is built on the concept of pipelines, and therefore supports the assumptions of the library design (see Section 3.1.1.1.

The cost of using a web application framework is that the generated endpoints are specific to that framework only, and thus the proof-of-concept library essentially serves as a plugin for Ktor, providing additional functionality.

4.2.4 Code Generation - KotlinPoet

For code generation, two options were considered:

- Use a control flow-based to generate a string with the content of the code to generate.
- Use a DSL-based template for writing files.

The approach chosen is mostly relevant for further development and maintenance of the library, as the generated output will be the same regardless of the approach chosen. For this reason, a library offering a domain-specific language for code generation was chosen, KotlinPoet. Compared to a template engine, this allows fast changes to the code generator, while still being more structured than using a string-based approach, increasing maintainability.

4.3 Integration with Other Projects

The library is intended to be used with Gradle, allowing reflection and code generation during compilation of the project using the library. Using the library in this manner requires four actions:

- Add the Kodegen library as a dependency using the `implementation` configuration. This makes the contents of the Kodegen library available to the project, including the `ApplicationService` and `RequireAuthentication` annotations and the dependencies used in the generated code.
- Add the Kodegen library as a dependency using the `kapt` configuration. This causes Kapt to be executed using the `KodegenProcessor` on project build. Note that the Kapt plugin must also be enabled for the project.
- Annotate the services and methods to generate code for using `ApplicationService` and `RequireAuthentication`. This is used by the `KodegenProcessor` to find the types to generate code for.
- Install the `ContentNegotiation` feature in the Ktor web application and a corresponding serialization feature in the Ktor HTTP client, e.g. `JsonFeature` if using JSON. This is used to ensure that transfer objects can be serialized and deserialized.

Thus, a project using the library will add the following to their `build.gradle` file:

```
//..
dependencies {
    //..
    implementation "dk.cachet:kodegen:1.0.0"
    kapt "dk.cachet:kodegen:1.0.0"
    //..
}
//..
```

And annotate the service to generate the endpoints in the following manner:

```
@ApplicationService
interface SampleService() {
    @RequireAuthentication
    fun authenticatedMethod() {
        // ...
    }
}
```

This results in the following code being generated:

- A Ktor module with endpoints for each method.
- An API client library class containing client methods for each endpoint.
- Transport objects wrapping method parameters and return values. These objects are simply used for convenience as they simplify the template of the endpoints.

Serialization is currently restricted to the formats available in the Kotlin serialization library, i.e. JSON, CBOR and Protobuf. However, Ktor only provides built-in support for JSON serialization in both the web application and HTTP client. Therefore, using other formats also requires a related handler for that format to be implemented for both the web application and HTTP client.

To use the built-in JSON serialization in the server web application, `ContentNegotiation` should be installed into the application pipeline with a JSON serializer:

```
fun Application.configurationModule() {
  install(ContentNegotiation) {
    json(anyJsonConfiguration, anySerialModule)
  }
}
```

Similarly, in the HTTP client, `JsonFeature` should be installed, providing a JSON handler as a parameter:

```
HttpClient() {
  install(JsonFeature) {
    serializer = KotlinxSerializer(json)
  }
}
```

4.3.1 Usage of Generated Code

4.3.1.1 Ktor Modules

To use the Ktor modules, they must be imported into a Ktor application engine environment. This is possible either by adding their qualified names to a Ktor configuration file or by importing them in an application environment (for an example, see Figure 4.3).

As each module corresponds to an application service, the constructor of which may take one or multiple parameters, it is recommended to use the second approach. This allows the application services to be instantiated elsewhere and passed into the modules.

Once the modules have been imported into the application engine environment, a server engine must be started using the environment. An example of this is available in Figure 4.2.

4.3.1.2 API Client Library

For each application service, a class with client methods corresponding to each method is constructed. The class takes the following parameters:

- A Ktor HttpClient. While a default HttpClient can be used, it is recommended that the client is configured to add relevant headers to each request, e.g. authentication or caching.
- A string with the base URL of the server to make the requests to, e.g. `"http://localhost:80"`.

Once the client has been created, it can be used as if accessing the application service directly by calling the functions of the service. However, exception handling and concurrency must still be handled outside the function call. An example is available in Figure 4.4.

4.4 Example of Generated Code

The following section contains an example of a simple application service, `DateService`, and the code generated when annotated with `ApplicationService`.

```
@ApplicationService
interface DateService {
  suspend fun getDate(): Date

  suspend fun getOffsetDate(offset: Int): Date

  @RequireAuthentication
  suspend fun setDate(newDate: Date)
}
```

Figure 4.1: The "DateService" interface used for the examples in Figures 4.2, 4.3, 4.4 and 4.5

```
Application.DateServiceModule(
  service: DateService,
  varargs authSchemes: String) {
  routing {
    post("/kodegenApi/dateService/getDate") {
      val result = service.getDate()
      call.respond(DateServiceGetDateResponse(result = result))
    }
    post("/kodegenApi/dateService/getOffsetDate") {
      val request = call.receive<DateServiceGetOffsetDateRequest>()
      val offset = request.offset
      val result = service.getOffsetDate(offset)
      call.respond(DateServiceGetOffsetDateResponse(result = result))
    }
    authenticate(*authSchemes) {
      post("kodegenApi/dateService/setDate") {
        val request = call.receive<DateServiceSetDateRequest>()
        val date = request.date
        val result = service.setDate(date)
        call.respond(HttpStatusCode.OK)
      }
    }
  }
}
```

Figure 4.2: Generated Ktor module for the "DateService" interface

```
//..
val dateService: DateService = DateServiceImplementation()
val environment = applicationEngineEnvironment {
    module {
        DateServiceModule(dateService, "basicAuthentication")
        // ..other modules
    }
    // ..other environment variables
}

val server = embeddedEngine(engine, environment)
server.start(wait = true)
```

Figure 4.3: Example of a Ktor project that uses a (here unspecified) HTTP engine and imports the module shown in Figure 4.2

```
class Date  ServiceClient(
  val client: HttpClient,
  val baseUrl: String) {

  suspend fun getDate(): Date {
    val response = client.post<Date> {
      url("$baseUrl/kodegenApi/dateService/getDate")
    }
    return response.result
  }

  suspend fun getOffsetDate(offset: Int): Date {
    val messageBody = DateServiceGetOffsetDateRequest(offset = offset)
    val response = client.post<Date> {
      url("$baseUrl/kodegenApi/dateService/getOffsetDate")
      contentType(ContentType.Application.Json)
      body = messageBody
    }
    return response.result
  }

  suspend fun getOffsetDate(newDate: Date) {
    val messageBody = DateServiceSetDateRequest(newDate = newDate)
    val response = client.post<Unit> {
      url("$baseUrl/kodegenApi/dateService/setDate")
      contentType(ContentType.Application.Json)
      body = messageBody
    }
    return
  }

}
```

Figure 4.4: Generated client service invoker for the "DateService" interface

```
@Serializable
data class DateServiceGetDateResponse(
  val result: Date
)

@Serializable
data class DateServiceGetOffsetDateRequest(
  val offset: Int
)

@Serializable
data class DateServiceGetOffsetDateResponse(
  val result: Date
)

@Serializable
data class DateServiceSetDateRequest(
  val newDate: Date
)
```

Figure 4.5: Generated transfer objects for the "DateService" interface

Chapter 5

Evaluation

The purpose of this chapter is to present the methods used to measure the effectiveness of the library.

Firstly, the success metrics for the evaluation are identified. Secondly, the evaluation methods used and their results are presented. Finally, the evaluation results are presented.

The evaluation methodology used in this thesis is largely based on the evaluation framework presented by Gediga et al. [15] in "Evaluation of Software Systems". Moreover, the interview questions were partially based on the concept of "Developer Experience" as introduced by Fagerholm and Münch [13].

5.1 Evaluation Metrics

As mentioned in Section 1.1, the goals for the thesis were to eliminate redundant code, reduce complexity and ensure consistent use of application services. This, in turn, is expected to increase developer efficiency.

The metrics used to evaluate the library should reflect these goals as well as possible. However, it is difficult to measure developer efficiency and experience, and doing so generally requires empirical data from development teams.

As an example, a commonly used metric in iterative development is "lead time", indicating the time from a product request to the delivery. While it would be interesting to compare how the lead time of a team is affected when using the library compared to without it, doing so is outside of the scope of this thesis.

Instead, metrics for which data can be made immediately available must be used.

Therefore, the following metrics were considered the most relevant:

- Reduction in Source Lines of Code as an indicator of reduced complexity
- Build time as a secondary metric to evaluate if the library negatively developer efficiency by excessively increasing build times.

Furthermore, it was decided that while conducting an experiment to determine developer efficiency was not feasible, a viable alternative would be to conduct interviews with developers to receive feedback on the experience of using the library.

5.1.1 Source Lines of Code (SLOC)

SLOC is intended to be a measurement of the size of an application, carried out by counting the total number of lines of code in the application source code. The determined size can then be used in deriving related metrics, e.g. the work required to maintain or extend the application.

Using SLOC to derive other metrics is controversial for multiple reason:

- The lines of code does not necessarily correlate with the amount of functionality offered by an application. This is especially true when comparing software written using different programming paradigms, e.g. imperative and declarative programming, or programming languages.
- The amount of lines of code varies greatly depending on the code layout, e.g. by using a ternary conditional expression on a single line instead of a traditional "if/else" conditional expression over multiple lines.

However, SLOC is somewhat effective in indicating the change in size or complexity from increasing or decreasing the number of lines of code. Thus, while not as useful in comparing different projects, it can be used in comparing the size of project to itself.

For this project, SLOC is most relevant in relation to the effort required to develop and maintain an application. For the evaluation, it will be assumed that a higher count of source lines of code also results in a higher effort to develop and maintain software, and thus reducing the amount of lines of code by automatically generating them instead decreases this required effort.

5.1.2 Build Time

While build time does not relate directly to the problem statement of the thesis, it is still relevant to measure, as excessive build times may decrease developer efficiency, which is to be avoided.

Because the library adds additional steps to the compilation process, notably annotation processing and writing of new files, the additional build time may be significant, and thus it was deemed worthwhile to investigate how big of an impact the library has on build times.

It should be noted that the library is based on the use of Gradle, which allows incremental builds. Ideally, the build time should be unaffected when the code has already been generated and the source from which it is derived has not been modified.

5.1.3 Developer Experience

As mentioned in Section 1.1, part of the motivation of the thesis was to eliminate certain unnecessary developer tasks in development of web applications, which may affect both developer efficiency and the developer experience. Thus, it is relevant to evaluate the library not only on objective software metrics, but also by making qualitative measurements of the actual developer experience when using the library.

The goal of using this metric is firstly to determine if the library actually helps developers by reducing the amount of work necessary, and secondly to determine if developers are accepting of how this is achieved.

5.2 Test Environment

For all of the tests, a sample project was used. This sample project consist of a simple Ktor web application and an associated client application for requesting the services of the web application. The intention of this project structure is to reproduce a client-server web application environment that developers of this type of software would normally use.

The project structure is as follows:

- **kodegen-study-client**: Client application with a dependency on *kodegen-study-core.*
  - **ClientApplication**: Simple application that uses the automatically generated library from *kodegen-study-core* and a pre-configured URL to make requests to an application running *kodegen-study-server.*
- **kodegen-study-core**: Library containing code shared by the client and server.
  - **application**: Contains a singe application service definition.
  - **domain**: Contains domain logic used by the application service.

  - **infrastructure**: Contains serialization modules and other code necessary to facilitate correct operation.

- **kodegen-study-server**: Server application with a dependency on *kodegen-study-core*.
  - **ServiceImplementation**: Implements the application service definition found in *kodegen-study-core*.
  - **ServerApplication**: Ktor web application that uses the generated endpoints from *kodegen-study-core* to expose *ServiceImplementation*.

5.3 Test Methodology

5.3.1 Source Lines of Code

To determine the reduction in SLOC, it is assumed that the generated code uses the same number of lines as would otherwise be written manually. Thus, the number of lines generated can be used to determine the reduction in SLOC.

The lines of code are counted as the number of logical lines for each client method, starting at the function declaration and ending at return statement, and for each API endpoint, starting at the initial `routing` or `authentication` statement and ending at the response statement.

Because the lines of code in both the client methods and endpoint is dependent on the number of parameters of the function and whether it returns a value, an assumption will be made that a function takes no more than five parameters. The reduction in SLOC will then be given as a range, rather than a single value, with the lower bound representing a function with no parameters or return value, and the upper bound representing a function with five parameters and a return value.

5.3.2 Build Time

To determine the change in build time when using the library, the study project is built in several configurations. During each build, a timer is used to measure the time taken for the build tasks, which are then added to get the total build time. The builds are made using Gradle with the `-no-daemon` argument to ensure that each build time is independent of previous builds.

The study project is built with the following configurations:

- Clean build with manually written code.
- Clean build with generated code.
- Rebuild with manually written code.

- Rebuild with generated code and incremental processing disabled.
- Rebuild with generated code and incremental processing enabled.
- Rebuild but with generated code deleted.
- Rebuild after modifying the manually implemented API
- Rebuild after modifying the interface from which code is generated.

The intention of these build configurations is to give an overview of the increase in compilation time that can be expected from generating code.

These tests are run again for a larger project where code must be generated for 20 application services. The intention of doing so is to see how the compilation time increases with project size when the library is used.

Note that this version of the project is not included in the source code, but was instead constructed by copying and renaming the existing application service and its implementation until 20 copies existed.

5.4 Results

5.4.1 Source Lines of Code

For each client method, the following lines are generated:

- Method signature.
- (If the function takes at least one parameter) A declaration of the message body.
- Network request
  - URL assignment.
  - (If the function takes at least one parameter) Content type assignment.
  - (If the function takes at least one parameter) Request body assignment.
- Return statement.

Thus, when the method takes no parameters and has no return value, 4 lines of code are generated. When the method takes at least one parameter and returns a value, 7 lines of code are generated.

For each endpoint, the following lines are generated:

- (If the function is annotated with `RequireAuthentication`) Authentication configuration.

- Routing configuration.
- Request reception method call.
- (For each method parameter) Parameter assignment.
- Service method call.
- Response method call.

Thus, when the method takes no parameters and has no return value, 4 lines of code are generated. When the method takes five parameters and requires authentication, 10 lines of code are generated.

Therefore, for each method declared in an application service, between 8 and 17 logical lines of code are generated. Therefore, the reduction in SLOC is 8 to 17 logical lines times the number of methods across all application services annotated with `ApplicationService`.

5.4.2 Build Time

The results of the tests on build time can be seen in Table 5.1 and 5.2. Each result is the average value of 3 tests.

Test	**Build time in milliseconds**	**Difference from Test #1**
#1: Clean build without code generation	10935	0%
#2: Clean build with code generation	15219	39.2%
#3: Rebuild without code generation	2001	-81.7%
#4: Rebuild with code generation and incremental processing disabled	7753	-29%
#5: Rebuild with code generation and incremental processing enabled	2144	-80.3%
#6: Rebuild with generated code deleted	6741	-38.4%
#7: Rebuild after modifying the web API, API client library, the associated application service and its implementation	8207	-24.9%
#8: Rebuild after modifying the interface from which code is generated and implementations of the interface	12443	13.8%

Table 5.1: Results from the build time tests with one application service containing 4 methods.

Test	**Build time in milliseconds**	**Difference from Test #1**
#1: Clean build without code generation	25519	0%
#2: Clean build with code generation	38112	49.35%
#3: Rebuild without code generation	1974	-92.25%
#4: Rebuild with code generation and incremental processing disabled	5686	-77.72%
#5: Rebuild with code generation and incremental processing enabled	2036	-92.02%
#6: Rebuild with generated code deleted	7133	-72.05%

Table 5.2: Results from the build time tests with 20 application services containing a total of 80 methods.

5.5 Developer Interviews

For evaluating the developer experience of using the library, semi-structured interviews were deemed the most viable method.

While surveys and questionnaires allow a higher amount of participants, the information they offer is less suited for this type of evaluation. Firstly, the answers may lean more towards a quantitative evaluation, e.g. by asking participants to rate the experience of using the library and then making a statistical analysis on the results.

Secondly, the participants are not expected to be familiar with the technologies used in the project, notably Kotlin and Ktor, nor the design paradigms encouraged, notably domain-driven design. Thus, the participants may require some introduction to the project context before the evaluation, which makes one-on-one interviews a better option, as they allow for explanation of the project as the participants are interviewed.

5.5.1 Interview Methodology

The interviews were intended to have the following format:

- **Interviewee Introduction**: The interviewee introduces themselves and presents their background in software development.
- **Introduction to the thesis context**: The context of the project and the functionality of the library is presented.
- **Introduction to study project**: A sample Kotlin project is presented to the interviewee. The interviewee is allowed to examine the project to become familiar with the project structure. Afterwards, the interviewee is asked to make changes to the project to get a small amount of "hands-on" experience using the library.
- **Evaluation questions**: The participants are asked open-ended questions on the experience of using the library and their thoughts on the library functionality. Participants are encouraged to share thoughts on both issues they encountered or expect to encounter and the positive experiences of using the library.

A series of questions were formulated to be asked at the beginning and end of the interviews. The intention of these questions is to determine if the developers felt that the library improved efficiency, improved understanding of the project and how it affected the developer experience of developing. Because these metrics are inherently qualitative, the questions are also open-ended.

The planned interview process, including the questions to be asked, can be seen in Tables 5.3, 5.4 and 5.5.

Topic	**Questions**
Experience in Software Engineering	What is your background in software development? Do you have any experience with full-stack development of web applications? Have you ever worked with projects where the client and server applications were built from a shared codebase?
Knowledge of Thesis context	Have you worked with projects using domain-driven design? Have you worked with reflection / code generation?

Table 5.3: The questions used for assessing the background of the participants

Section	**Content**
Introduction to the thesis	The motivation and goal of the thesis is presented and an introduction to the "Kodegen" library is given.
Introduction to the sample project	A sample project integrating the library is presented. The interviewee is given a quick overview of the project structure to provide some familiarity with the environment.
Project experimentation	The interviewee is asked to make a series of changes to the project, notably to the application service to show how library generates code.

Table 5.4: The scheduled project presentation for presenting the library to the interviewee

5.5.1.1 Recruitment and Results

Recruitment for the interviews were made through personal inquiry in several channels accessible to the author of this thesis. In total, approximately 60 persons were asked to participate in the interview. Of these, 3 agreed to do so.

It should be noted that the recruitment process took place during the lockdown in Denmark due to the COVID-19 pandemic taking place. As a result, the interviews took place using video conferencing rather than being conducted in person. This, in turn, may have affected the ratio of people willing to participate in an interview.

Interview notes containing paraphrasings of the answers given by the respondents during the interviews are available in Appendix A.

Topic	**Questions**
First impression of the library	What do you see as the advantages / disadvantages of using a library as this?
Project Complexity and Understanding	Do you consider the abstraction of the API to be beneficial or harmful to the understanding of a project? Can you imagine scenarios where you would prefer one approach over the other?
Limitations	Can you imagine scenarios where this approach would not work?
Interest in use	When would / wouldn't you use the library? What would need to be different for you to want to use the library and what additional features would you want from a library as this? How would you compare this approach to existing approaches, e.g. specification-based API client library and documentation generation?

Table 5.5: The questions used for evaluation of the library in the semi-structured interviews

Chapter 6

Discussion

In this chapter, the effectiveness of the library will be discussed and compared to existing approaches based on the results found in the evaluation and an analysis of the proof-of-concept implementation of the design.

6.1 Analysis of Evaluation Results

6.1.1 Source Lines of Code

As shown in section 5.4.1, the reduction in source lines of code when generating code is between 8 and 17 logical lines of code per method.

As explained in Section 5.1.1, it is difficult to compare this to the overall size of a project, the size of which may vary by orders of magnitude depending on type and scope. Instead, it may be more worthwhile to consider the reduction in SLOC as a contribution not to the overall size of the project, but to the area of the project concerning the web application.

In this case, the reduction in SLOC may be significant. As mentioned in 3.1.2.4, the web application must only be configured once, whereas each exposed service requires a corresponding method. Thus, endpoints and client methods are responsible for the most additional required code as the project grows in size and complexity, and eliminating it may effectively reduce the task of developing and maintaining the web application to just the configuration of the application.

6.1.2 Build Times

As shown in section 5.4.2, the build times were longer when code had to be generated. A clean build with code generation incurred a slowdown of 39.2% compared to a clean build without code generation, while a rebuild after modifying the interface and implementations took 13.8% longer than a clean build. By comparison, making a similar change without code generation allowed the build to complete in 24.9% less time.

In the tests with 20 application services, similar results were seen: a clean build with code generation took 49.35% longer than a clean build without code generation.

While this is a significant increase, it should be noted that the build times remained relatively low despite the increase, with the smaller and larger project compiling in approximately 15 and 38 seconds, respectively.

Rebuilding with code generation and incremental annotation processing provided results similar to rebuilding without code generation.

Depending on the use of the library, these build times may be inconsequential. If the generated artifacts are only generated once and then distributed, the increase in build time has no relevance for projects using the generated artifacts. If, however, the library is used to generate artifacts which are used immediately after, the build time may be source to development inefficiency.

6.1.3 Developer Experience

According to Fagerholm and Münch [13], the developer experience is in part impacted by "how developers perceive the development infrastructure". This can be influenced positively by ensuring that the development infrastructure gives a feeling of empowering the developer rather than restricting them.

For this project, the analysis of developer experience will not be applied to using the library itself, but rather how using the library impacts the development infrastructure by eliminating the task of developing and maintaining the web API and API client library. This analysis will mainly be based on the interviews presented in Section 5.5.

6.1.3.1 Usability and Impact on Feeling of Understanding

In the interviews, the participants were asked how they believe the library would affect the understanding of a client-server project.

Two of the interviewees answered that the library did not negatively impact the understanding, because the generated code simply served as a way to link two individual components, the client and server, together. By automatically generating this code, the developer is instead allowed to focus on the underlying services and domain logic.

Only one of the interviewees had worked with domain-driven design, and thus little feedback was given on how well the library would support this. However, given the answers on how the library allows the developer to focus on the domain logic, rather than how it is published, it could be argued that the library aligns with the goals of DDD.

One interviewee responded that while the library was intuitive to use, they would imagine that it would not fare as well in edge cases, notably when the template of the library does not support functionality which also cannot be configured in the web application or HTTP client. Similarly, a second interviewee mentioned that they would want a statement in each endpoint that uses the authentication to authorise the user, which was not present in the template.

In this case, the library may negatively impact the developer experience. A developer may first try to use the default approach but encounter problems because the edge case is not supported. Then, they may attempt to investigate if their desired functionality can be aligned with the functionality offered by the library. Finally, on realizing that this is not the case, they will resort to manually implementing the change without the library.

However, all interviewees responded that they felt the library was an attractive option for development of web applications. Furthermore, during the project demonstration, each interviewee quickly grasped the functionality of the library and how to apply it, indicating that the library itself is also easy to use.

6.1.3.2 Impact on Feeling of Efficiency

In all interviews, the participants responded that they thought the library would improve efficiency in development because the code generation eliminates the task of writing and maintaining repetitious code. However, this benefit is further amplified by two additional benefits: firstly, the generated artifacts are loosely coupled but must remain consistent, which is ensured by generating the code. Secondly, because the generated code is used for distributed systems, it allows the code to remain consistent despite the applications using them potentially being developed independently of the others.

Thus, the library provides a feeling of efficiency not only because it automates a common task, but because that task is one that requires the developer to remain vigilant of changes.

6.2 Comparison to Existing Solutions

6.2.1 Specification-based Approach

The most relevant comparison to draw is to the specification-based approach which was explained in Section 2.6.2.2. Furthermore, in two of the interviews, the library was also compared to specification-based approaches.

In the specification-based approach, the web API may either be maintained manually along with a human-readable specification of it or the specification may be generated from the web API. In both approaches, it is left to the developer to ensure that the web API is consistent with the underlying server logic. In the first approach, the developer must also ensure that the specification matches the web API.

By contrast, the library developed in this project only requires the developer to maintain the application service definitions. From these, both the web API and API client library can be developed.

The desired goals of each approach are similar, though not identical. In two of the interviews, the interviewees cited development time and reduction in developer errors as the reason that they use specification-based generation. Moreover, these were also two of the benefits they identified for the library.

However, the project library has the additional benefit of preserving information from the application service. In a specification-based approach is used, the web API must be translated into a specification and from the specification back into code. In the approach, the generated artifacts may preserve the original application service information.

A benefit of the specification-based approach is that it is easily adaptable to any platform, given that tools for that platform exists, because the specification is technology agnostic. By contrast, a Kotlin library will only support the platforms supported by Kotlin, and will otherwise require a similar library to be developed for other technologies. Notably, interpreted languages, e.g. Python, must use libraries that allow them to execute native code for this library to support them.

6.2.2 Hybrid Approach

An alternative approach that has not been attempted in this project is to combine the application service-based approach with the specification-based approach. This could be implemented by only generating the web API from the application services, then using the specification-based approach to generate the remaining artifacts.

One of the benefits of this approach is that it allows the integration of the existing specification-based code generation ecosystem into the library. Given that a specification can be generated from a Ktor web API and API client libraries from the specification, this would allow the generation of API client libraries for all platforms support by the specification-based code generation tools. This may negate the need for multiplatform compilation, assuming that the server web application only needs to be compiled to JVM.

6.3 Issues and Limitations

This section explains the most relevant limitations of the proof-of-concept implementation of the library. These limitations are identified based on the evaluation results and an analysis of the implemented library compared to the requirement specification described in Section 3.1.

6.3.1 Flexibility

The artifacts generated by the library use a DSL-based template. This template contains some decision-making based on the input provided, e.g. whether to include a serialization of parameters or not based on whether a method takes parameters. However, the only allowed adjustment of the template is through the use of the `RequireAuthentication` annotation which adds an authentication to the route in the endpoint.

The intention, as described in Section 3.1.1.2, was that most functionality should be installed through configuration of the web application or HTTP client, with the endpoint only containing essential functionality. As an example, using encrypted communication can be enabled by installing it in the web application without modifying the endpoints.

However, it is possible a developer will want specific functionality which must be present in the endpoints which is not available to the library. As an example, a developer may want an authorisation check using a private security library before specific services are invoked.

In this case, it should be decided if the library should allow the template to be modified, or if the library should only support the functionality it is built with.

While the first option greatly increases the complexity of the library and opens up the potential for errors occurring due to invalid templates, the second option also limits the utility of the library.

If it was decided that the template should be allowed to be modified, one method of doing so would be to "break down" the structure of each generated artifact into sections and allowing injection of new code between sections. As an example, the endpoint could be broken into the following sections:

- Authentication

- Routing
- Request reception and deserialization of parameters
- Service invocation
- Response

Implementing a customizable template that guarantees valid code could prove an interesting task, but was considered outside the scope of this project.

6.3.2 Security and Authorization

In its current form, the proof-of-concept implementation supports authentication but not authorisation based on the principal that the authentication provides. For projects that require authorisation, this makes the library an unviable option.

The motivation behind not adding an option for authorisation in the same manner as with authentication was that Ktor does not support authorisation. Moreover, Ktor authentication returns a Ktor-specific principal type, which therefore cannot be used for authorisation libraries outside of Ktor.

A second security-related issue is ensuring that serialization occurs as expected, which was also brought up by an interview participant. On generation of the transfer objects, the library checks the type of all parameters and return types to see if they have a built-in serializer, as is the case with primitives, or if they are annotated with `Serializable`. If neither are present, the parameter or return type is annotated with `ContextualSerialization`, indicating that either a serializer must be declared at runtime or the `ContextualSerializer` must be used.

In cases where the `ContextualSerialization` annotation is applied, it should ensured that a serializer specific to that object is defined and used by the web application / HTTP client. Not doing so will result in the `ContextualSerial-izer` being used for serialization, which may unintentionally expose fields that should not be serialized.

6.3.3 Threats to Validity

In addition to the limitations of the library, the following could be considered threats to the validity of the thesis:

- The library was designed specifically with an implementation in Kotlin in mind because Kotlin supports multiplatform compilation. The premise was that libraries would only have to be defined once and could be deployed on multiple platforms. However, the Kotlin annotation processor, Kapt, currently only works for projects that only target JVM. This could either be considered a limitation of the current implementation or a threat to the

premise of the thesis.
JetBrains s.r.o., the main developer of Kotlin, has not announced any plans to support multiplatform annotation processing. An open-source third-party annotation processor for multiplatform projects does exist, MpApt [19], though it has not been investigated if it offers the same functionality as Kapt.

- The number of participants that could be recruited for the interviews was limited to just three. Having more participants would allow for a wider array of opinions on the library and the approach it encourages. Moreover, having more participants express the same opinion or thought increases the trust that it is representative for software developers in general. In addition to the limited number of participants, all participants had limited professional experience with software development. Having interviews with developers with more professional experience may have exposed concerns that newer developers would not identify.

Chapter 7

Conclusion

7.1 Project Findings

From the library design presented in Chapter 3 and the proof-of-concept implementation presented in Chapter 4, it was shown that it is possible to generate a web API and API client library from application service definitions that allows client applications to request web services provided by a web application.

Conceptually, the approach used by this library is considered an improvement over the specification-based approach because it allows a single source of truth to be maintained. This, in turn, eliminates the task of maintaining the web API and either the API client library or the specification from which it is generated.

The proof-of-concept implementation of the library was evaluated using a series of tests and interviews. The results from these indicate that the library is effective in increasing developer efficiency by eliminating boilerplate code and eliminating multiple code sources that must be kept consistent. Moreover, the evaluations indicated that the library is effective in encouraging a domain-driven design approach to web services development by allowing developers to focus on domain logic rather than how to expose it.

The current design has a number of limitations. Most notably, the generated code is restricted to a template defined in the library, limiting the flexibility of use and available functionality in the generated code.

7.2 Future Work

The following areas would be the most relevant to further explore:

- Adapting the library to support Kotlin projects that target multiple platforms.
- Configurable templates, allowing for more flexible code generation, notably by allowing the use of third-party libraries in the template and generation of templates for other web application frameworks than Ktor.
- Integration with specification-based approaches, e.g. by generating a web API from application service definitions, a specification from the web API and finally an API client library from the specification. This would also allow implementations in languages that do not support multiplatform compilation.

Bibliography

[1] Anderson, R., and Smith, S. ASP.NET Core Middleware | Microsoft Docs. https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware.

[2] Balzer, R. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering SE-11*, 11 (1985), 1257–1268.

[3] Belqasmi, F., Singh, J., Bani Melhem, S. Y., and Glitho, R. H. Soap-based vs. restful web services: A case study for multimedia conferencing. *IEEE Internet Computing 16*, 4 (2012), 54–63.

[4] Boon, M. E. Boonoboo/kotlin: Kotlin library for automatic generation of Ktor Application modules and Ktor HttpClient methods. https://github.com/Boonoboo/rad.

[5] Collins, D., and Collins, D. *Designing object-oriented user interfaces.* Benjamin Cummings Redwood City, CA, 1995.

[6] Constantine, L. The Emperor Has No Clothes: Naked Objects Meet the Interface, 2002.

[7] Corporation, M. HttpClient Class (System.Net.Http). https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient.

[8] Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. *Distributed Systems: Concepts and Design*, 5th ed. Addison-Wesley Publishing Company, USA, 2011.

[9] Czarnecki, K., and Eisenecker, U. W. Components and generative programming. *ACM SIGSOFT Software Engineering Notes 24*, 6 (Jan 1999), 2–19.

[10] discordjs. discordjs/discord.js: A powerful JavaScript library for interacting with the Discord API. https://github.com/discordjs/discord.js.

[11] Espinha, T., Zaidman, A., and Gross, H.-G. Web API growing pains: Loosely coupled yet strongly tied. *Journal of Systems and Software 100* (2015), 27–43.

[12] Evans, E. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[13] Fagerholm, F., and Münch, J. Developer experience: Concept and definition. In *2012 International Conference on Software and System Process (ICSSP)* (2012), pp. 73–77.

[14] Fowler, M. Richardson maturity model: steps toward the glory of rest. *Online at* ***http://martinfowler.com/articles/richardsonMaturityModel.html*** (2010), 24–65.

[15] Gediga, G., Hamborg, K.-C., and Düntsch, I. *Evaluation of Software Systems*, vol. 72. CRC Press, 11 2002, pp. 127–153.

[16] Huang, Z. zijianhuang/webapiclientgen. https://github.com/zijianhuang/webapiclientgen.

[17] Inc., S. KotlinPoet Metadata - KotlinPoet. https://square.github.io/kotlinpoet/kotlinpoet_metadata/.

[18] JHipster. JHipster. https://www.jhipster.tech/.

[19] Klingenberg, J. Foso/MpApt: Kotlin Native/JS/JVM Annotation Processor library for Kotlin compiler plugins. https://github.com/Foso/MpApt.

[20] Kuusinen, K. Software developers as users: Developer experience of a cross-platform integrated development environment. In *Product-Focused Software Process Improvement* (Cham, 2015), P. Abrahamsson, L. Corral, M. Oivo, and B. Russo, Eds., Springer International Publishing, pp. 546–552.

[21] Larus, J. What happened to the promise of software tools?

[22] Liskin, O., Singer, L., and Schneider, K. Teaching old services new tricks. *Proceedings of the Second International Workshop on RESTful Design - WS-REST 11* (2011).

[23] LLC, G. API Client Libraries | Google Developers. https://developers.google.com/api-client-library.

[24] Millett, S., and Tune, N. *Patterns, principles, and practices of domain-driven design*. John Wiley & Sons, 2015.

[25] Newman, S. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[26] OpenAPITools. OpenAPI Generator. https://github.com/OpenAPITools/openapi-generator.

[27] Palviainen, J., Kilamo, T., Koskinen, J., Lautamäki, J., Mikkonen, T., and Nieminen, A. Design framework enhancing developer experience in collaborative coding environment. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (New York, NY, USA, 2015), SAC '15, Association for Computing Machinery, p. 149–156.

[28] Pawson, R. Naked Objects. *IEEE Software 19*, 4 (2002), 81–83.

[29] Pawson, R., and Matthews, R. Naked objects: A technique for designing more expressive systems. *SIGPLAN Not. 36*, 12 (Dec. 2001), 61–67.

[30] Perrey, R., and Lycett, M. Service-oriented architecture. In *2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings.* (2003), pp. 116–119.

[31] Rademacher, F., Sorgalla, J., and Sachweh, S. Challenges of domain-driven microservice design: A model-driven perspective. *IEEE Software 35*, 3 (May 2018), 36–43.

[32] Ray, W. J., and Farrar, A. Object model driven code generation for the enterprise. In *Proceedings 12th International Workshop on Rapid System Prototyping. RSP 2001* (2001), pp. 84–89.

[33] Schmidt, D. C. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY- 39*, 2 (2006), 25.

[34] Sebastián, G., Gallud, J. A., and Tesoriero, R. Code generation using model driven architecture: A systematic mapping study. *Journal of Computer Languages 56* (2020), 100935.

[35] Selic, B. The pragmatics of model-driven development. *IEEE Software 20*, 5 (Sep. 2003), 19–25.

[36] Shepherd, D. C. The Value of Applied Research in Software Engineering. `http://blog.ieeesoftware.org/2016/09/the-value-of-applied-research-in.html`.

[37] SlackAPI. slackapi/python-slackclient: Slack Python SDK. `https://github.com/slackapi/python-slackclient`.

[38] Software, S. Swagger Codegen. `https://github.com/swagger-api/swagger-codegen`.

[39] s.r.o., J. Http Client - Clients - Ktor. `https://ktor.io/clients/index.html`.

[40] s.r.o, J. Ktor - asynchronous Web framework for Kotlin. `https://ktor.io/`.

[41] s.r.o., J. Pipeline - Advanced - Ktor. `https://ktor.io/advanced/pipeline.html`.

[42] Suter, R. NSwag. `https://github.com/RicoSuter/NSwag`.

[43] Tan, W., Fan, Y., Ghoneim, A., Hossain, M. A., and Dustdar, S. From the service-oriented architecture to the web api economy. *IEEE Internet Computing 20*, 4 (2016), 64–68.

[44] Upadhyaya, B., Zou, Y., Xiao, H., Ng, J., and Lau, A. Migration of soap-based services to restful services. In *2011 13th IEEE International Symposium on Web Systems Evolution (WSE)* (2011), pp. 105–114.

[45] VMware, I. Spring | Home. `https://spring.io/`.

[46] Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. A note on distributed computing. In *International Workshop on Mobile Object Systems* (1996), Springer, pp. 49–64.

[47] Watson, D. *A Practical Approach To Compiler Construction*, 1st ed. Springer International Publishing, USA, 2017.

[48] Welsh, M. Academics, we need to talk. `http://matt-welsh.blogspot.com/2016/01/academics-we-need-to-talk.html`.

[49] Xinyang Feng, Jianjing Shen, and Ying Fan. Rest: An alternative to rpc for web services architecture. In *2009 First International Conference on Future Information Networks* (2009), pp. 7–10.

Appendix A

Interview Notes

This appendix contains the notes taken during each interview.

Please note that the notes are paraphrasings of the answers given by the interviewees.

Interview 1, June 7th - Software Developer #1

Introduction

Question: What is your background in software development?

The interviewee is a software developer with a master's degree in computer science. They have 6 years of academic experience and 3,5 years of professional experience.

In their job, they primarily work with full-stack development using Angular and ASP.NET Core.

Question: Do you have any experience with full-stack development of web applications?

They have worked on developing and maintaining full-stack projects for 2 to 2,5 years.

Question: Have you ever worked with projects where the client and server applications were built from a shared codebase?

Several of the projects mentioned previously are intranet websites, and thus the client (web page files) is served from the back-end.

Question: Have you worked with projects using domain-driven design?

They have tried to use it during development, but believes that progress in implementing it is very limited due to resistance from team members, as well as a lack of knowledge on how to apply it.

Question: Have you worked with reflection / code generation?

They have used reflection through annotation processing in a C# project for modifying methods during compilation.

Moreover, they have used a specification-based approach to generating API client libraries and web API documentation for the intranet projects mentioned previously. They state the it is very convenient, since it saves the time to manually write the API client library, but has limitations, e.g. when encountering edge cases that the generation is not capable of handling. In these cases, the option of either overriding or implementing the method manually must be available.

Introduction to the Thesis

The interviewee is introduced to the thesis context as described in the introduction.

Introduction to the Study Project

The interviewee is introduced to the library functions through the study project.

After ensuring that they understand the project structure and library functionality, they are asked to make changes to an interface annotated with `ApplicationService` and to add the `RequireAuthentication` annotation to a method.

Evaluation

Question: What do you see as the advantages / disadvantages of using a library as this?

They see the advantages of this library being the ease of use, stating that simply annotating a class or interface is simpler than manually maintaining a web API specification.

Moreover, automatically generating both the web API and API client libraries both saves the developer time from having to do so manually, and reduces the number of errors occurring due to code mistakes, e.g. mistyping URLs or using wrong types.

They support using a template-based approach over manually implementation, arguing that it ensures high quality and consistent code, given that the template is high quality.

They see the main disadvantages as being the lack of control of implementation for edge cases or cases where there must be a total control over the implementation, e.g. in security-critical projects.

They give an example of an endpoint that served a file, which the specification-based code generation tool was unable to serialize correctly. The code was still generated, but did not function as expected.

Moreover, they explain that for security-critical projects, they would not trust the implementation used for the generated code, primarily because the template used is not immediately available.

Question: Do you consider the abstraction of the web API / client network requests to be beneficial or harmful to the quality of a project and / or the developers understanding of the project, and can you imagine scenarios where you prefer one approach over the other?

They believe that this library could be beneficial for certain types of projects.

Of the two approaches that were tried out in the demonstration (first generating from an interface in the shared kernel, secondly by generating from the implementation in the server), they believe that the first holds the most promise, because it ensures that not only is the API client library and web API kept consistent, but the client and server also have to use the same interface, providing a much-desired consistency. They explain that a common issue in the development of the intranet projects is having to redefine object types in the client that have already been defined in the client, even though the project holds both. Being able to use a shared interface, despite the client and server being developed in different languages, would greatly improve the workflow in development of client-server systems.

Question: Can you imagine scenarios where you would prefer this "black-box" approach over manually implementing the web API?

As answered previously, they would prefer manually implementing the web API in security-critical projects, as they would want full control over the implementation, especially regarding authentication, authorization and ensuring that objects are serialized correctly to avoid including sensitive data.

However, they are are leaning towards code generation being a better approach, primarily because it saves the developer a lot of time and improves the experience of developing client-server systems.

Question: Can you imagine scenarios where this approach would not work?

They repeat that they would not use it for security-critical projects.

Additionally, they explain that one of the advantages of using a specification-based approach that this library lacks is the adaptability to any project due to the extensive amount of tools. This library being restricted to Kotlin and the platforms it can compile to restricts use in e.g. Python projects without having to go through additional steps to adapt native code to the language.

Question: Do you think abstracting the endpoints behind a service invoker / service provider increases or decreases understanding of the project?

They find that if you are already familiar with the project and library functions, it does not decrease understanding of the project, since the web API and client methods can be considered "glue code" to allow server functions to be called from the client.

However, a developer who is not aware of the library being used or who does not understand how the library works may make false assumptions on e.g. latency and delivery guarantees. Thus, it should be made clear that the generated functions are simply wrappers for networking calls, and that providing guarantees must be implemented by the developer, e.g. by making asynchronous calls, providing error handling etc.

Question: When would / wouldn't you use the library?

Given that the generated code would be available for multiple platforms, they would be interested in using the library for web applications where

They would not be interested in using the library for security-critical projects or projects where the developer cannot trust a third-party template, as is the case with this library, but wants to have manual control over the implementation. They add that, in general, they only trust generated code for simple cases such as wrapping a function call with primitive parameters and returns types in a client-server application running in a secured environment.

Question: How would you compare this approach to existing approaches, e.g. specification-based API client library and documentation generation?

They find that web API specifications, specifically OpenAPI, are difficult to write manually and are prone to developer errors. However, this can be mitigated by generating the specification from an existing API.

Thus, instead of writing and maintaining the specification directly, they prefer writing the web API, generating the specification from this and the remaining artifacts from the specification. This ensures compatibility between the web API and the client, and since the web API is compiled with the services, static checking is available to improve consistency between the web API and the underlying services. They concede that this does require manual maintenance of the web API along with the underlying services, as opposed to generating it from the services.

They explain that despite this, they would still use a specification-based approach over the approach proposed by this library simply because the tool support for the specification-based approach is far more mature. Moreover, as answered previously, specification-based approaches are adaptable to multiple languages, which allows them to use it despite the languages used by the client and server. Being restricted to developing the shared kernel in Kotlin and compiling it to multiple platforms reduces the interest in the library.

Question: What additional features would you want from a library as this, and what would need to be different for you to want to use the library?

They answer that they would want to be able to customize the template used, allowing the generated code to be customized to the needs of the individual projects. Notably, they would want fine control over serialization and to be able to choose between different content types.

As a general note, they point out that generated code is an excellent tool for saving time and reducing errors. However, when the template used does not support a specific case and they have work around it, the interest in code generation decreases notably. Thus, either being able to "override" the template or easily integrating manual implementations would provide a lot more interest.

Interview 2, June 8th - Master's Student #1

Background

Question: What is your background in software development?

The interviewee is a master's student in computer science specializing in software engineering. They are currently writing their thesis.

They have 5,5 years of academic experience as a a software developer and 3,5 years of professional developer, having held a student job related to software development.

Question: Do you have any experience with full-stack development of web applications?

They have worked with both front-end (client) and back-end (server) projects, mainly React-based front-end development and ASP.NET-based back-end development.

Question: Have you ever worked with projects where the client and server applications were built from a shared codebase?

They explain that the previously mentioned project was a website application where the back-end provided the client for the front-end.

Question: Have you worked with projects using domain-driven design?

They have heard of it, but never used it.

Question: Have you worked with reflection / code generation?

They state that they have used specification-based tools for generation of API client libraries and documentation.

Introduction to the Thesis

The interviewee is introduced to the thesis context as described in the introduction.

Introduction to the Study Project

The interviewee is introduced to the library functions through the study project.

After ensuring that they understand the project structure and library functionality, they are asked to make changes to an interface annotated with

`ApplicationService` and to add the `RequireAuthentication` annotation to a method.

Evaluation

Question: Do you consider the abstraction of the web API / client network requests to be beneficial or harmful to the quality of a project and / or the developers understanding of the project, and can you imagine scenarios where you prefer one approach over the other?

They states that they like this approach, as it hides the implementation details of the web API, instead allowing the developer to focus on the underlying domain logic.

Moreover, they state that, since the goal of the web API simply is to facilitate communication, abstracting the web API does not reduce understanding of the project.

They state that they would prefer generating both the web API and the API client library over manually implementing them because it:

- Saves time due to having to write less code.
- Reduces developer errors, e.g. from type mismatching or use of wrong URLs.

Question: Can you imagine scenarios where this approach would not work?

They cannot.

Question: Comparing this approach to specification-based generation, which pros and cons do you see for each approach over the other?

They would prefer using the specification-based approach because they are familiar with them, and because they are at a much higher maturity level. It is emphasized that trust in the capabilities of the tool is important. The benefits of this approach, i.e. saving time and reducing the number of developer errors, are also achieved when using the specification-based approach.

Question: What additional features would you want from a library as this, and what would need to be changed for you to be interested in using the library?

They state that they would want both authentication and authorization to be available in the web API, e.g. by authenticating the user to create a principal, then using this in the authorization process.

Interview 3, June 8th - Master's Student #2

Background

Question: What is your background in software development?

The interviewee is a master's student in computer science currently writing their thesis.

They have 6 years of academic experience and 3 years of professional experience. During their professional experience, they have worked primarily with mobile applications (Android and iOS) and developing libraries/frameworks for use in mobile applications.

Question: Do you have any experience with full-stack development of web applications?

They have worked on web application using the LAMP stack (Linux, Apache, MySql, PHP)

Question: Have you ever worked with projects where the client and server applications were built from a shared codebase?

The LAMP application they had worked on was a web service project, and thus served the client (web page files) from the back-end.

Question: Have you worked with projects using domain-driven design?

The projects they have worked on during their student job are likely developed using DDD, but that they have not participated in the design process itself.

Question: Have you worked with reflection / code generation?

They have some understanding of reflection, and have attempted to use code reflection for a project written using Dart.

They wanted to generate methods for specific types, but that the tool support for Dart did not support doing so.

Introduction to the Thesis

The interviewee is introduced to the thesis context as described in the introduction.

Introduction to the Study Project

The interviewee is introduced to the library functions through the study project.

After ensuring that they understand the project structure and library functionality, they are asked to make changes to an interface annotated with `ApplicationService` and to add the `RequireAuthentication` annotation to a method.

Evaluation

Question: Do you consider the abstraction of the web API / client network requests to be beneficial or harmful to the quality of a project and / or the developers understanding of the project, and can you imagine scenarios where you prefer one approach over the other?

They state that generating code has several advantages over manually developing it, mainly because it reduces developer errors, e.g. due to mistyping URLs or using wrong types.

Moreover, they state that automatically generating the web API is useful, because it saves the developer from having to understand how the web API is implemented, and allows them to focus on the services it publish and the underlying domain logic.

They state that this is especially useful for larger projects.

Question: Can you imagine scenarios where this approach would not work?

They state that they would not use the library for smaller projects, because the overhead from understanding the annotations processing etc. outweighs the overhead from manually implementing and maintaining the web API.

When it is mentioned that other interviewees brought up authorization and security issues, they stated that they would solve this at a different level than at the web API, as they would only use the web API for transferring of data, not as a method of implementing logic.

Question: Can you imagine scenarios where you would / wouldn't you use the library?

They repeat the answer from before.

Question: What additional features would you want from a library as this? What would need to be changed for you to be interested in using the library?

They request that it should be clarified that the generated code is not to be modified.