CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Using OpenAPI 3 Specifications of the 5G Core to Generate Validators in Erlang

Using Open API 3 to generate Erlang code to validate HTTP requests according to the network specifications

Master's thesis in Computer science and engineering

DAVIDE DONATO

HEYU QIN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Using OpenAPI 3 Specifications of the 5G Core to Generate Validators in Erlang

Using Open API 3 to generate Erlang code to validate HTTP requests according to the network specifications

DAVIDE DONATO

HEYU QIN

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Using OpenAPI 3 Specifications of the 5G Core to Generate Validators in Erlang
Using Open API 3 to generate Erlang code to validate HTTP requests according to the network specifications
DAVIDE DONATO, HEYU QIN

Supervisor: John Hughes, Department of Computer Science and Engineering
Advisors: Egon Larsson and Anders Björsson, Ericsson
Examiner: Magnus Myreen, Department of Computer Science and Engineering

Master’s Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Using OpenAPI 3 Specifications of the 5G Core to Generate Validators in Erlang
Using Open API 3 to generate Erlang code to validate HTTP requests according to the network specifications
DAVIDE DONATO, HEYU QIN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Most of the 5G Core network components are virtualized and designed as RESTful APIs, communicating between each other by sending HTTP requests. These requests are defined in the 3GPP 5G Core official specifications following the OpenAPI 3 format. In this thesis we explore the current status of Erlang code generation from OpenAPI 3 specifications and its use for validation in Ericsson's 5G Core system. The best candidate tool for the project was OpenAPI Generator, although it required several fixes to be used. Based on Ericsson's internal test environment, we show that it is possible to integrate the generated code for request validation in it. The integration was successful as it found two bugs in the system, but it highlighted the lack of support for some advanced features. These shortcomings indicate that Erlang code generation from OpenAPI 3 is good enough for testing but not for production.

Keywords: 5G Core, OpenAPI 3, Erlang, code generation, validation, HTTP requests, integration, thesis.

Acknowledgements

First we would like to thank our supervisor in Chalmers, Prof. John Hughes, for his continuous help and guidance. His feedback always inspired us to come up with better ideas and directed us towards the right way.

We would like to express our sincere gratitude to our advisors in Ericsson, Anders Björsson and Egon Larsson, who kindly helped us to get familiar with this thesis topic, generously shared their knowledge about the testing tools and systems, and enlightened us when we had trouble proceeding the thesis. Many thanks also to Daniel Karlsson, our manager during this time, for giving us the opportunity to study and work in Ericsson for this thesis project.

I, Davide Donato, want to dedicate this work in memory of my father Marco, who gave his everything to let me get where I am now, and in honor of my mother Chiara and grandmother Delia, who helped me finding the strength to keep going forward. I also want to thank my family and friends, for always cheering for me: Patrizia, Francesco, Clara, Vanni, Jacopo, Filippo, Marica, Daniele, Valentina, Viola, Martina, Antonio, Giovanni, Andrea G, Arianna S, Daniele G, Daniele Z, Federico R, Francesco C, Luca F, Matteo G, Mattia M, Pietro P, Pietro T, Eleonora F, Elisa P, Eva P, Filippo M, Giacomo R, Marco Alessandro C, Mattia C, Tobia Z, Alina D, Nils N, Denis F, Emy P, Alejandro V, Edin T, Ce L, Mengxuan L, Congrui L, Hua K, Jieer C, Tamami M, Florian W, India L. My old workmates at Intercom Solutions, who supported me even though that meant me leaving. A special thank you to Julian and Annika, who welcomed me in Sweden and treated me as part of the family. Heyu, for putting up with me for two years of working together. Oleksii, for being a trusted friend always ready to help. And finally, thank you Smaragda for being always encouraging and believing in me.

Davide Donato, Heyu Qin, Gothenburg, September 2019

Contents

1 Introduction

The 5G Core is the newest core network for broadband internet access, characterized by a higher data rate but most importantly, a considerably reduced latency. It is designed by 3GPP, a standard organization which focus on writing the specifications for telecommunication [1]. Compared to the LTE network core, there have been radical changes in the core's design philosophy. The newest system uses a Service Based Architecture (SBA), where all the network functions are integrated as virtualized nodes. These nodes are designed as RESTful APIs [38], meaning that the functionalities they provide are accessed through HTTP requests and executed without any pre-existing context. This design has been adopted as it promotes modularity, reusability and self-containment of the network functions [33] [13].

There are a lot of intricate communications happening among the core's nodes. The components communicate with each other by sending HTTP requests to the paths defined in each network function's RESTful API, granting access to their operations [34]. The paths, the linked HTTP methods, and the operations of the network functions are described in detail through OpenAPI 3, a specification standard specialized in defining RESTful API.

Maintaining such a complex system comes with a lot of difficulties. Any change of the specification can potentially require a manual reconfiguration of the whole system, which is very time-consuming and prone to bugs. That is what makes code generation so appealing, as it is a reliable and efficient tool for adapting the system automatically to the newest specifications.

For example, code 1.1 displays an extract from the 3GPP specifications of the Network-function Repository Function (NRF) [14]. This GET request [15] alone has 37 different parameters, each with their own distinct definition. The first parameter `'target-nf-type'` uses an externally defined schema `'NFType'` in file `'TS29510_Nnrf_NFManagement.yaml'`, making it difficult for the developers to follow the specifications. This kind of reference is very frequently used, which creates a chaotic reading experience jumping back and forth between files. To handle the requests, the developer also needs to manually implement and verify all the parameters. Although simple, this task can be very laborious and repetitive. A lot of parameters might be very similar to each other, with just slight differences. Therefore, it requires separate but almost identical handling functions which can easily introduce bugs through their highly repetitive yet different code.

```yaml
openapi: 3.0.0
info:
  version: '1.1.0.alpha-1'
  title: 'NRF NFDiscovery Service'
# ... servers and security information
paths:
  /nf-instances:
    get:
      summary: Search a collection of NF Instances
      operationId: SearchNFInstances
      tags:
        - NF Instances (Store)
      parameters:
        - name: target-nf-type
          in: query
          description: Type of the target NF
          required: true
          schema:
            $ref: 'TS29510_Nnrf_NFManagement.yaml#/components/schemas/NFType'
        # ... 35 parameter definitions
        - name: If-None-Match
          in: header
          description: |-
            Validator for conditional requests,
            as described in IETF RFC 7232, 3.2
          schema:
            type: string
      responses:
        # ... responses definition
```

Code 1.1: Extract from the 3GPP specifications of the 5G Core's NRF Discovery.

On the other hand, by using code generation it is possible to greatly reduce both the amount of work and the risk of bugs, allowing the system developers to focus on more critical parts such as the logic handlers. To follow this idea, an analysis of the generated code is required to understand how to integrate it into an existing system. To ensure that the generated code can be used without problems, its quality needs to be evaluated. An important part of it is to confirm up to what level the generated code covers the specification, since OpenAPI 3 provides a very rich and precise description of the services that might be hard to implement with code.

1.1 Research Question

- What tools are good candidates for generating Erlang code from OpenAPI 3 specifications? Does the generator produce quality code?
- How can the generated code be integrated into an existing system? Can it be integrated without major changes?

- What coverage of the specifications can be generated by the code generator? Are there features that can be described by OpenAPI 3.0 but cannot be generated?

1.2 Limitations

- Ericsson is currently implementing the 5G Core in Erlang, so this project will only focus on code generation from OpenAPI 3 to Erlang. The code generator might give different results with other target languages.
- The specifications that are used for generation in this project are mainly from the 3GPP 5G Core specifications. In particular, we will only consider the components which have been implemented in the company's system.
- We will focus on using the generated code for HTTP request validation. It is the section of the code which is the most dependent on the details of the specifications and it would be the most convenient part to integrate in an already existing system.
- For evaluating the correctness of the generated code, we will only apply it to validate HTTP requests in the system to see if the validation gives the expected result.

1.3 Contributions

- Currently, the only tool available for code generation in Erlang from OpenAPI 3 is OpenAPI Generator. Even after our work, it is still lacking support for some widely-used advanced features. The shortcomings make the Erlang generator too immature for generating complete production-level code, but it has enough functionalities to be used for testing.
- We confirmed the generated code for validation can be integrated into an existing system with minimal changes. Integrating generated code for validation can find issues that are not identified by manually written test cases.
- The Erlang generator has been improved as it now supports features that are used extensively by the 3GPP 5G Core specifications. Also, our added functionalities, a customized body reader and callback module, make the generated code flexible for integration in systems with varying requirements.

2

Background

2.1 5G Core

The 5G Core (5GC) is the core/backbone network [44] [43] of 5G, the newest cellular network for broadband access. The 5GC has been redesigned from the previous generations to be “cloud native”, inheriting many of the technology solutions used in cloud computing and virtualizing its traditional network elements. For instance, Network Functions Virtualization technologies [39] are applied in 5GC, which blur the physical boundaries between traditional network components. This redesign aims to make the 5GC open and flexible enough to meet the diversity of service and business requirements of the 5G era [19]. To achieve this, the 5GC Network has been based on what is called “Service-Based Architecture” (SBA).

2.1.1 Service-Based Architecture

Service-Based Architectures (SBA) provide a modular framework from which common applications can be deployed using components of varying sources and suppliers [35]. 3GPP defined an SBA for 5GC Network and it is specified in 3GPP technical specification 23.501 [2]. Figure 2.1 is the representation of SBA. In this representation, control-plane functions, shown in the blue box, are authorized to access each other’s services via Service Based Interfaces (SBI) [5]. This is enabled by one key Network Function (NF) of SBA called Network Repository Function (NRF). NRF provides NF service registration and discovery, allowing NFs to identify appropriate services in one another [12]. This new architecture provides much more flexibility for adding or modifying services that fulfill the network requirements of the vast and diverse 5G use cases. It becomes possible for a component to connect to other components without introducing specific new interfaces or modifying existing ones.

In addition, SBA not only makes 5GC more flexible but also more open, i.e. making 5GC Network functionalities available to 3rd parties such as service providers. This service is provided by the Network Exposure Function (NEF). Exposure of functionality is a very common concept used for web-services which are offered over the internet. For this reason, 3GPP decided to use the REST architecture design paradigm, which describes the design of distributed applications and more specifically, Application Programming Interfaces (APIs) [34]. Thus, the interface provided

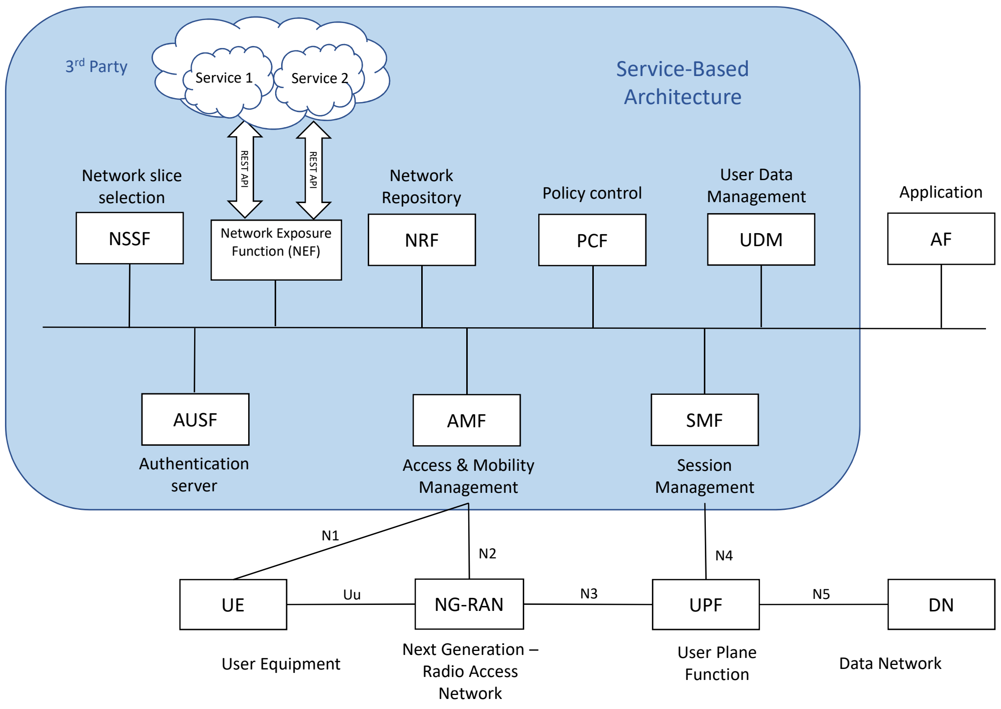


Figure 2.1: Serviced Based Architecture of the 5G Core network.

by the NEF to 3rd party should be based on RESTful APIs, as shown in figure 2.1.

After deciding to use RESTful APIs for 3rd party functionality exposure, 3GPP also chose to use RESTful APIs over SBI. Therefore, the internal communications within 5GC Network have the same principles as the functional exposure, allowing a harmonized and comprehensive technological approach of the complete 5G system [34]. RESTful APIs nowadays are intending to be designed by OpenAPI Specification, which is a specification standard for interface files to describe and visualize RESTful web services. The newest version of OpenAPI Specification is OpenAPI 3. More information and details will be introduced in Chapter 4. Furthermore, the development of web-based technologies described through OpenAPI are supported by development tools, specification tools, code generators, etc.

2.1.2 Network Functions of 5GC

To understand more about the 5GC Network, it is very important to know the NFs that are related to the core network. Some of them are very similar to their corresponding functions in the previous generations' core networks. This is because the network has some basic functions such as: communicating with the User Equipment (UE), storing its subscriptions and credentials, allowing access to external networks and services, providing security, and managing the network access and mobility. Moreover, there are some new functions which are needed for enabling the new network paradigms such as slicing and service-based networking [11].

The main 5G NFs are listed below (note that not all functionalities of each function are presented): [5] [11] [14]

- **Access and Mobility Management Function (AMF)**: supports registration management, connection management, mobility management, access authentication and authorization, security context management.
- **Network Slice Selection Function (NSSF)**: supports selection of the Network Slice instances to serve the UE. It also determines the allowed Network Slice Selection Assistance Information (NSSAI) and determines the AMF set to be used to serve the UE. NSSF is the one of the new functions that is not presented in previous core networks.
- **Session Management Function (SMF)**: supports session management according to network policy (including session establishment, modification, and release), UE IP address allocation and management, QoS and policy information delivery to RAN via the AMF, and traffic steering configuration for User Plane Function (UPF) for proper traffic routing.
- **User Plane Function (UPF)**: supports packet routing and forwarding, packet inspection, and QoS handling. UPFs can be deployed in various configurations and locations, according to the service type.
- **Policy Control Function (PCF)**: supports a policy framework incorporating network slicing, roaming, and mobility management. It also provides policy rules to Control Plane functions and accesses subscription information for policy decisions in UDR.
- **Authentication Server Function (AUSF)**: acts as an authentication server and stores keys.
- **Unified Data Management (UDM)**: supports generation of Authentication and Key Agreement (AKA) credentials, user identification handling, and access authorization. Stores subscribers' data and profiles.
- **Network Exposure function (NEF)**: supports exposure of capabilities and events. It works as an API gateway that allows secure provision of information from external applications to 3GPP network and provides translation of internal/external information.
- **NF Repository function (NRF)**: supports service registration and discovery function so that NFs can discover each other and communicate via APIs. It maintains NF profile and available NF instances. NRF is one of the new functions that is not presented in previous core network.
- **Application Function (AF)**: supports application influence on traffic routing, accessing NEF, interaction with policy framework for policy control.

2.1.3 Example Scenario

This section will depict an example scenario that has a very common pattern of communication between services within the 5G SBA. This example can also exemplify how RESTful principles are used by 3GPP. Note that the given example is simplified to just give an overview.

Let's consider a user who wants to use one of the services, e.g. to browse the web. The involved network functions in this procedure are AMF, SMF, and NRF. AMF serves as the single-entry point for a UE for all its communications, so when the user starts his/her equipment, AMF is responsible for assigning a SMF which manages the user's session context. Therefore, the AMF needs to discover a suitable and available SMF for the service. This is where the NRF is needed as the SMF is obtained via the Service Discovery procedure performed between the AMF and the NRF [34]. However, for the discovery to be processed successfully, the SMF needs to be registered with the NRF in advance. These steps mentioned above are illustrated in figure 2.2.

3GPP TS 23.501 defines the roles of the service consumer and the service producer [34]. The service consumer is the network function which sends requests for service, while the service producer is the network function which exposes/provides the requested service. So, in this example, the AMF is the service consumer and the SMF is the service producer. The three different procedures in this scenario are:

1. **Service Registration**: the SMF registers its services that will be provided to NRF.
2. **Service Discovery**: the AMF sends the request to the NRF for a suitable and available SMF. In return, the AMF receives the address of the SMF which was registered in step 1.
3. **Session Establishment** [34]: the requested session is established on the control plane level by the AMF via the SMF.

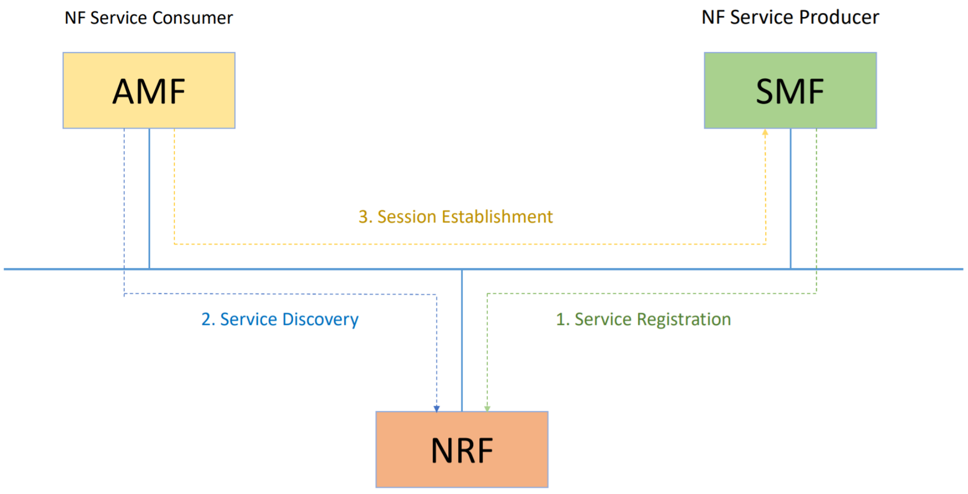


Figure 2.2: Example Scenario of SBA Service Framework.

2.2 Related work

In this section, we will introduce code generation and the advantages that it brings. With it, we will also mention the previous attempt of the company on generating Go code, from both OpenAPI 2 and OpenAPI 3. This is meant mainly as an addition to motivate both the project itself and some of the choices taken during the project.

2.2.1 Code generators

A code generator is a program which performs a model-to-text transformation that ensures the automatic transformation of a model into code [42]. The concept of code generation originated with compilers, but it then diverged into two different branches: compilers, which produce machine-executable programs from higher level programming languages, and automatic programmers, which produce user-readable code from specifications [37].

Code generation gained popularity in software organizations as a tool to speed up software development. Normally, maintaining a system is a challenging process due to the indirect connection between system specification and implementation. Also, some optimizations of the code might be done by taking advantage of some undocumented knowledge about the system, making it very hard to understand the final code without that additional information. Abstraction of code helps mitigating this issue, since it allows the developer to focus on structuring a high-level model or template of the program logic rather than the implementation details. The abstraction is then given as an input to a code generator, which provides the corresponding implementation as a result. On top of promoting a more abstract implementation, code generation further simplifies maintenance of the source code since a change of specifications would just require regenerating and replacing the code [3].

There are many techniques to generate code, but a popular one is template-based code generation (TBCG). What makes it appealing is not just the easier development of code generators themselves, but also its favor towards the "write once, produce many" principle. Each template is a mixture of a static part, which is left unchanged, and a dynamic part, which is replaced by the template engine with the information extracted from the input. The core concept of TBCG is to take advantage of these dynamic components, iteratively expanding them with the available data into the final code [42]. This technique is the one adopted in the code generators mentioned in this thesis.

2.2.2 Golang generation from OpenAPI 2

Before the introduction of OpenAPI 3, Ericsson had already tried to generate Go code from OpenAPI 2, the previous version of OpenAPI Specification. The most popular code generators for Go were Swagger Codegen and Go Swagger. However, the Swagger Codegen project had very poor performance as it could only generate a go client which only supported flat models (models that can only represent simple relationship between data). The main reasons why Swagger Codegen could not work as expected are as follows [18]:

- Go's limited type system does not fit well in the model of Swagger Codegen
- Go's idea of polymorphism does not reconcile very well with a tool that was designed for languages which support inheritance.

Compared to Swagger Codegen, Go Swagger could provide better performance and more complete features from the specifications for the code generation, although it still had some obvious shortcomings. One of the unsupported cases is the encoding and the decoding of complex types, like lists of unions of lists. Another major limitation was the lack of some essential properties in OpenAPI 2, meaning manual implementation was required for specific functionalities. One of these is API with `'callbacks'`, which are widely used by core network components as they allow the server to act as a client and the client to act as a server. A typical example is a subscription functionality that would enable, for instance, notifications about events that a user subscribes for in a service.

2.2.3 Golang generation from OpenAPI 3

With the introduction of OpenAPI 3, 3GPP decided to use it as the interface definition language for designing the SBA of the 5GC network. The new features of OpenAPI 3 could be applied to describe precisely the complex structures and interactions needed in the Core, which was not fully possible before (e.g. callbacks). Ericsson tried to work on Golang generation from the new specifications with their available tools, but there was not enough support for this use as Go Swagger does not support OpenAPI 3.

After looking into other OpenAPI 3 related projects, Ericsson found a Go command line tool, named Google Gnostic, which could convert JSON and YAML OpenAPI specifications from and to equivalent Protocol Buffer representations[1]. Gnostic's Protocol Buffer models could be used to generate code which included data structures with explicit fields for each of the elements in an OpenAPI description. With this feature, Gnostic could read OpenAPI descriptions into designated data structures, report errors, resolve internal dependencies, and write the results into a binary form that could be used in any language supported by the Protocol Buffer tools. The generator flow is shown in figure 2.3.

With more research on the generation process, it was revealed that many OpenAPI features, including the new features, were not fully supported by Gnostic. The generation still did not cover complex data types and callbacks. The problematic area is marked in figure 2.3. It is shown that the Surface API-Model and Gnostic Go

[1]Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data

Generator are the components that could not provide all the needed functionalities. There were two options to improve this generation process: patch these two components to add the missing features (figure 2.4) or make a new Go code generator which would work as expected and replace the incomplete components with it (figure 2.5). The first option would require the developers to follow the structure of these components whenever a new feature was added. The second option did not have this restriction, but it required a new coupled framework. In the end, the company chose second option, implementing their own generator to make it capable of supporting callbacks and other needs.

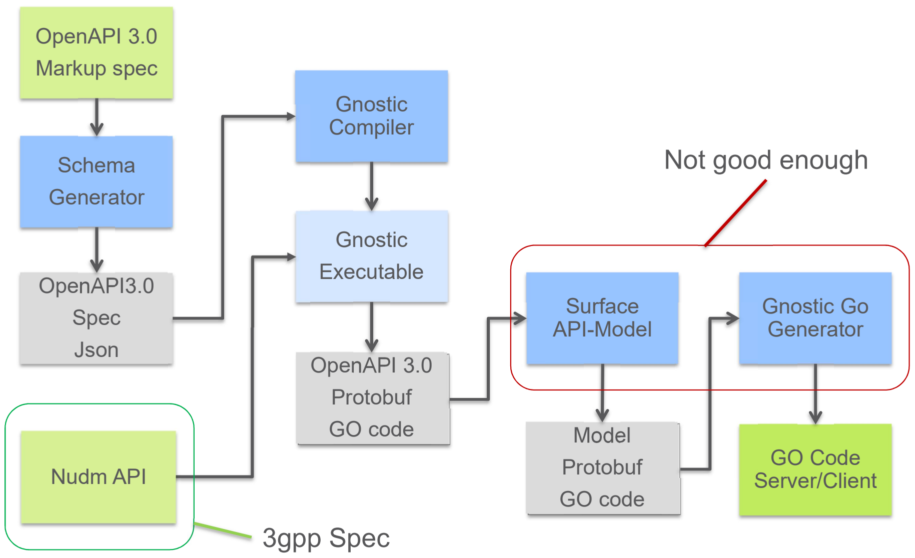


Figure 2.3: Generator flow for Golang code generation with Gnostic

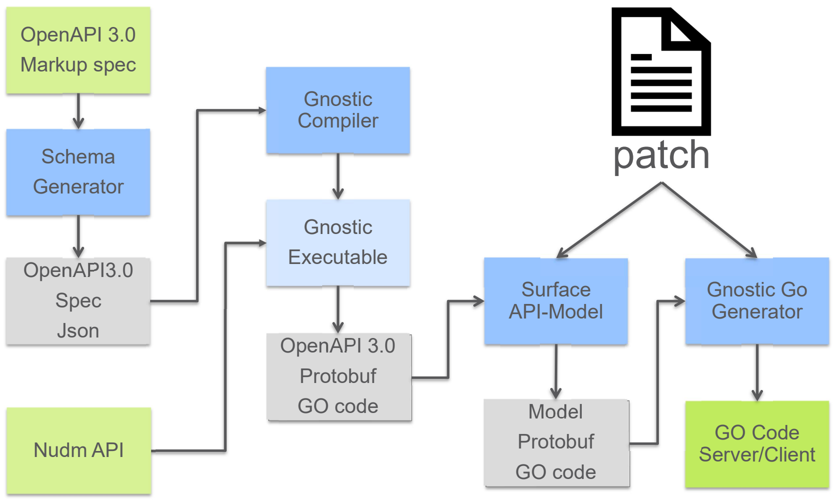


Figure 2.4: Option 1 to fix the generator

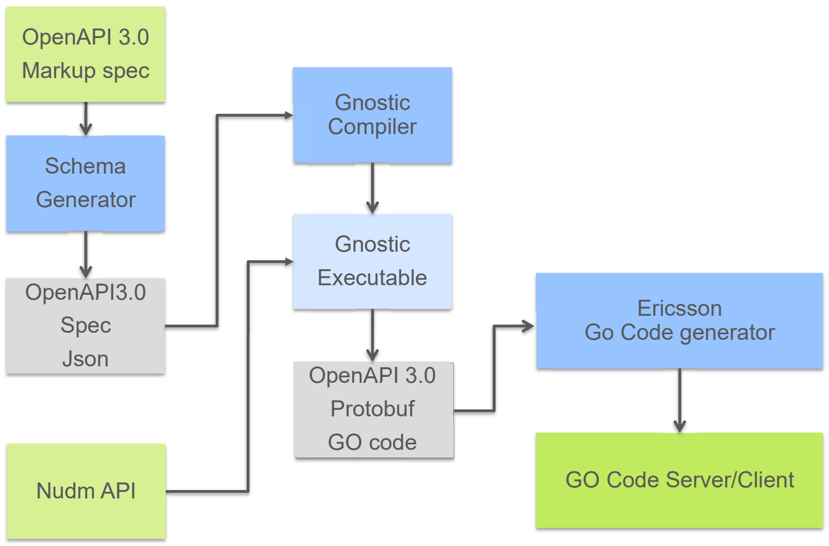


Figure 2.5: Option 2 to fix the generator

3
Approach

The main idea of this project is to select an appropriate code generator that supports Erlang generation from OpenAPI 3, see whether it is possible to integrate its generated code into an existing system, and understand which changes are required to make it work.

For the selection of a generator, we need to make sure that it satisfies a set of essential requirements, meaning that it should at least be able to generate runnable code. The next step from there is to analyze the support of OpenAPI 3 for both the problematic features, such as ENUMs and regular expressions, and advanced features, such as composed schemas and callbacks. In the case there is no generator meets all the requirements, we will select the least problematic one and refine it to fit the needs.

Once the selected generator works as expected, the rest of the project will be about integrating the generated code into an existing system with minimal changes. For this project we used GTT, Ericsson's internal test tool that provides a simulated network for testing the implementation of the network's nodes.

The logic inside of GTT follows a standard HTTP server structure, as shown by the green boxes in figure 3.1, which is based on the Erlang's web server Cowboy. The server is configured for each 5G core component with the corresponding paths defined in the 3GPP specification. Each path has a handler function assigned to it, which is responsible for collecting the request's data and passing it to the logic handler. For example, consider the specification for NRF Network Function Discovery (NFDiscovery). When added to GTT, its server's router in Cowboy will redirect the GET requests sent to the path `'/nf-instances'` to a handling function `'nrf_nf_instances_handler'`. The handler will collect the data from the GET request and pass it to its corresponding logic handler.

After analyzing the structure, we realized two things: first, the server and its router were built according to the company's needs, so we cannot replace them with a generic generated server. Second, the logic handler cannot be generated as it depends on what the programmers want to do with the received request. With these observations, the request handlers become the best place to integrate the generated code.

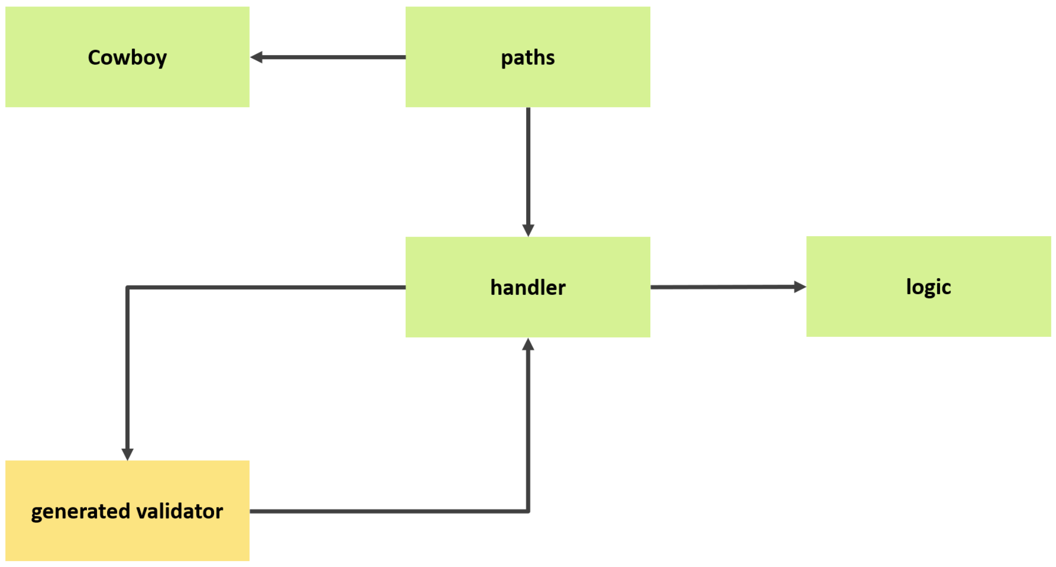


Figure 3.1: Integration of the generated code into GTT.

While examining the generated code, we considered that the most suitable component for integration would be the validation for requests (yellow box in figure 3.1). If the validation is integrated into the handler, invalid requests can be discarded before passing data to the logic handler, while valid requests stay unaffected. In this way, the integration would not influence the existing flow of the system. Furthermore, the content of validation functions depend exclusively on the specifications and can have very repetitive code, making them an ideal candidate for code generation.

4

Tools

4.1 The OpenAPI 3 Specification

The OpenAPI Specification (OAS) [31] defines a standard, language-agnostic interface for RESTful APIs. It allows both humans and computers to discover and understand the capabilities of a service without access to its source code or documentation. When properly defined, a consumer can understand and interact with the service with a minimal amount of knowledge about the implementation logic.

An OpenAPI document is a JSON [32] object represented through the YAML [4] or JSON format which conforms to the OAS. These are based on the JSON Schema Specifications with some modifications aimed to specialize its use towards the description of RESTful APIs.

4.1.1 Root objects

There are 8 root objects in the OpenAPI 3.0 Specification: `‘openapi’`, `‘info’`, `‘servers’`, `‘paths’`, `‘components’`, `‘security’`, `‘tags’` and `‘externalDocs’`. Among these, only `‘openapi’`, `‘info’` and `‘paths’` are required [28]. The content of each object is predefined by the OAS to standardize and optimize the representation of RESTful HTTP applications. The section in which programmers have the most freedom to define their own structures is on the definition of parameters and contents for both requests and responses. These definitions allow to deeply customize APIs based on the service's needs while maintaining a standardized interaction between applications.

openapi

The semantic version number of the OAS. The patch version should not be considered by tooling, making no distinction between `‘3.0.0’` and `‘3.0.1’`.

info

The `‘Info Object’` [22] provides metadata about the API. It requires the definition of the title and the version of the API, but it can also contain optional information such as contact information, license, terms of service and a description.

servers

An array of '`Server Objects`' [30] which define the URLs of the target hosts. The URLs may either be absolute, referring to an external host, or relative, referring to a host which can be found in the same location of the OAS document. It is also possible to define in the object additional variables needed by the host such as ports or default paths.

paths

The '`Paths Object`' [27] is the heart of the APIs definition as it is where the API's functionalities and their relative paths are defined. Each path is described by a '`Path Item Object`' [26] which lists the available REST '`Operation Objects`' [24] (HTTP request methods) and describes their individual behavior. OpenAPI defines a unique operation through a combination of path and HTTP method: this means that two distinct operations with the same HTTP method under the same path are not allowed. For each operation it is possible to define the expected parameters and their locations, the body of the request, and all the expected responses. It is also possible to redefine the operation's host and even mark the presence of callbacks, which are a collection of '`Operation Objects`' themselves.

components

The '`Components Object`' [21] holds a set of reusable objects that can be referenced through the property '`$ref`' followed by the object's path. These references can be both from the same file or an external file, and they follow the pattern

```
$ref: '[path/to/file]#/components/<object path>'
```

If '`[path/to/file]`' is excluded, then the path will refer to the '`components`' defined in the same file. The defined objects have no effect on the API unless they are explicitly referenced from properties outside of the '`Components Object`'.

security

A list of security mechanisms used across the whole API. They can be specialized at the operation level if needed, which in that case will override the global security definition only for the single operation. Although it is possible to define multiple '`Security Requirement Objects`' [29], only one of them needs to be satisfied to authorize a request.

tags & externalDocs

Components used mainly to add metadata and information to the API. The '`Tag Objects`' may also be used to force a parsing order on the parsing tool.

4.1.2 Examples

This section illustrates the main features of OpenAPI 3 used by 3GPP in the 5GC specifications through three examples. Their purpose is to give an overview of what specifications might look like and what it is necessary to know when writing your own specifications. For the scope of this project, it is not necessary to know OpenAPI 3 in detail. However, having an idea of how they are defined might be useful to understand better some issues described in the following chapters.

To avoid redundancy between examples, we will only highlight the content of the `'paths'` object, which would be included in the specification of code 4.1. Normally the schemas are defined under components and then referenced through the `'$ref'` object. Nevertheless, in the examples, they will be inlined for clarity. For code generation it is better to avoid inline definitions for complex objects and define them under `'components'`. This ensures that every object has a unique identifier and allows the generator to build reusable models more easily in the target language.

```yaml
openapi: '3.0.0'
info:
  version: 1.0.0
  title: Example
servers:
  - url: http://localhost:8001/
# ================
# paths definition
# ================
components: # List of reusable user-defined objects
  schemas: # Category of the objects
    Message: # User defined unique identifier
      type: object # Type of the object
      properties:
        message: # Locally defined unique identifier of the property
          type: string
          pattern: '^[a-zA-Z0-9 ]*$'
      required: [message] # List of the object's required properties
  responses:
    '200':
      description: Expected response to a valid request
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Message'
    DefaultResponse: # Only description is required
      description: unexpected error
```

Code 4.1: The skeleton for our examples of OpenAPI Specifications. It is missing the definition for the `'paths'` object, which will be defined separately for each example.

Regex and ENUM values

Code 4.2 describes the path `‘/regex-enum’` which expects a GET request with two required parameters in query, `‘pin’` and `‘optionParam’`. The parameter `‘pin’` is a string which follows the pattern `‘NNN-NN-NNNN’`, where `‘N’` is any digit from 0 to 9. The parameter `‘optionParam’` is also a string but it can only have `‘first’`, `‘second’`, or `‘third’` as value. After sending a well-formed GET request to `‘<host>/regex-enum’` we expect either of these two replies: a successful code '200' with a `‘Message’` as response body or a default response. We can find the definitions for Message and the responses in `‘components’` of code 4.1.

In this case we expect values through the `‘parameters’` object, which is an array of `‘Parameter Object’` [25]. Each of these objects requires the name and the location of the parameter, which can be any among `‘query’`, `‘header’`, `‘path’`, or `‘cookie’`. There are plenty of parameter-specific properties, but the most frequent ones are whether it is required (it is mandatory and set to true if the location is in `‘path’`) and what is the schema for the object's value.

It is not possible to pass values with complex schemas through path parameters, so the values are restricted to simpler types such as lists of base types, objects, and base types. For lists and objects there is the option `‘explode’`, which generates separate parameters for each value of the array or key-value pair of the map. This behavior is set as default for `‘query’` and `‘cookie’` parameters.

```yaml
paths:
  /regex-enum: # Relative path
    get: # Request method
      operationId: TestRegexEnum # This GET operation's unique ID
      parameters: # Array of expected parameters
        - name: pin # Name of the parameter
          in: query # Location of the parameter
          required: true
          schema: # pin is a string that follows a regex pattern
            type: string
            pattern: '^[0-9]{3}-[0-9]{2}-[0-9]{4}$'
        - name: optionParam
          in: query
          required: true
          schema: #optionParam must have one of the listed values
            type: string
            enum: [ first, second, third ]
      responses: # Expected responses from the current operation
        '200':
          $ref: '#/components/responses/200'
        default:
          $ref: '#/components/responses/DefaultResponse'
```

Code 4.2: An example of request which expects parameters that follow regex patterns or ENUM values.

Callbacks

The ‘`callbacks`’ object is one of the latest additions to OpenAPI 3. It allows programmers to define the list of callbacks linked to the current request and specify which parameters they are related to.

Code 4.3 could represent the requests required for building a subscription service. After receiving a GET request with the ‘`callbackUrl`’ parameter, the server can then send callback requests to the received URI. This kind of interaction can be described through the ‘`callbacks`’ property, a list of ‘`Callback Objects`’ [20] related to the original request.

The definition of a ‘`Callback Object`’ is very similar to the one of a ‘`Path Item Object`’, but it defines its path differently. A ‘`Path Item Object`’ has a predefined explicit path, while a ‘`Callback Object`’ references a variable containing a URI. Each ‘`Callback Object`’ is also characterized by a unique identifier, normally the name of the event. In code 4.3 we can see that the callback is identified with the ID ‘`onTopicUpdate`’ and it uses the expression ‘`$request.query.callbackUrl`’ as path to reference the query variable ‘`callbackUrl`’.

```yaml
paths:
  /subscribe:
    get:
      operationId: TestCallback
      parameters:
        - name: callbackUrl
          in: query
          required: true
          schema:
            type: string
            format: uri
      responses: # ... same as Regex & Enum Example
      callbacks:
        onTopicUpdate: # Unique identifier of the callback object
          '{$request.query.callbackUrl}': # Received URL
            post:
              requestBody:
                description: subscription payload
                content:
                  application/json:
                    schema:
                      type: object
                      properties:
                        notification:
                          type: string
                      required: [ notification ]
              responses: # ... same as Regex & Enum Example
```

Code 4.3: Callbacks define the requests that are expected by a received URL for a callback operation.

Complex data types

Code 4.4 highlights another new feature of OpenAPI 3, the support for polymorphism and combination of schemas. This is introduced by the keywords ‘`oneOf`’, ‘`anyOf`’, and ‘`allOf`’, which validate the value respectively against *exactly one of*, *one or more of*, or *all* the subschemas defined or referenced in them [40].

Code 4.4 expects a POST [15] request, so most of the data is passed through the request body. Generally, the request body’s ‘`content`’ is described through one or more ‘`Media Type Objects`’ [23], where each of them is a mapping of a media type string to its schema. For example, in code 4.4 ‘`application/json`’ is mapped to an inline object schema.

As the media type is ‘`application/json`’, its schema needs to have a set of label-value pairings so that a JSON can be formed. Because of this, the “root” schema must be of type ‘`object`’. This object is expected to have either (oneOf) a username or an email AND (allOf) at least (anyOf) a password, a pin, or both. The responses can be either the same as in code 4.2 or customized with alternative definitions.

```yaml
paths:
  /complex-data:
    post:
      operationId: TestComplexData
      requestBody:
        content:
          'application/json':
            schema:
              type: object # We expect a JSON object...
              properties:  # ... with these properties
                username:
                  type: string
                email:
                  type: string
                  format: email
                password:
                  type: string
                  format: password
                pin:
                  type: string
                  pattern: '^[0-9]{3}-[0-9]{2}-[0-9]{4}$'
              allOf: # All of the properties need to be met
                - oneOf: # Exactly one property must be met
                  - required: [ username ]
                  - required: [ email ]
                - anyOf: # At least one property must be met
                  - required: [ password ]
                  - required: [ pin ]
      responses:
        # ... same as Regex & Enum Example
```

Code 4.4: The definition of a request that expects a complex data structure, in this case an object with several properties and variable requirements.

4.2 OpenAPI Generator

The richness of OpenAPI can be used for purposes other than just providing a detailed description of a web API's functionalities. By taking advantage of its strict structure, it is possible to build tools that generate documentation and functioning code from OpenAPI documents. Code generation from OpenAPI Specifications is quite powerful as it can produce several server components in just a few seconds without the risk of human error. This shifts the focus on to the design of specifications rather than fixing the existing implementation.

While there are plenty of code generation tools from OpenAPI specifications [9], we consider two in particular because of their potential capability of generating Erlang code: Swagger Codegen [41] and OpenAPI Generator [7]. The first one is an open-source tool developed and supported by SmartBear, while OpenAPI Generator is a fork of the project maintained by the community.

Although very similar in structure, OpenAPI Generator has more contributions due to its community-driven nature, resulting in better and more frequent maintenance. For this reason, several languages that are not supported by Swagger Codegen for OpenAPI 3, are supported by OpenAPI Generator. Erlang is one of them, as Swagger Codegen can only generate Erlang code from OpenAPI 2 specifications. This limits us to use OpenAPI Generator for the rest of the project.

4.2.1 The generator's structure

Most of the generator's work is handled by two classes: `‘DefaultGenerator.java’` and `‘DefaultCodegen.java’`. The first class is dedicated to managing the global configuration of the generator, handling the file generation and parsing the input file with the external library `‘swagger-parser’`. The second class processes the parsed YAML/JSON file so that it can be used for generation.

It is possible to add support for a new language to the code generator by extending `‘DefaultCodegen’`. The subclass should override the encoding functions and adapt them to the target language's syntax and conventions. For example, it needs to make sure that variable names in Erlang start with an uppercase letter. Such subclasses are classified into clients and servers, ensuring the separation between the two architectures as required by the RESTful guidelines. Finally, they link the language-specific templates that are used to generate code.

The template files are written in the logic-less (it does not provide any statement to build the program's logic), templating language Mustache [45]. Each file is a combination of the target language's code and Mustache tags, where the target language builds the program's logic and the tags are expanded using the values provided in a hash or object. Each template can be used to produce one or multiple files: for example, `‘model.mustache’` of the Erlang client is used to create one Erlang module for each user-defined schema that has been referenced through `‘$ref’` in the input specification.

```
1 {{#apiInfo}}{{#apis}}{{#operations}}{{#operation}}
2 request_params('{{operationId}}') ->
3     [{{#allParams}}{{^isBodyParam}}
4         '{{baseName}}'{{/isBodyParam}}{{#isBodyParam}}
5         '{{dataType}}'{{/isBodyParam}}{{#hasMore}},{{/hasMore}}
6     {{/allParams}}];
7 {{/operation}}{{/operations}}{{/apis}}{{/apiInfo}}
8 request_params(_) ->
9     error(unknown_operation).
```

Code 4.5: The generator's template for creating the '`request_params`' function.

Code 4.5 is an extract from a template of the Erlang Server generator. To understand the code, it is necessary to know a few things about mustache's syntax:

- Variables are accessed by writing the variable name between two or three curly brackets, depending on whether the value must be escaped or not. For example, if a parameter is named '`name>`', '`{{baseName}}`' will return '`name&gt;`' while '`{{{baseName}}}`' will return '`name>`'.
- Boolean variables can be used to write conditional blocks through an opening and a closing tag. There are two kinds of opening tags: either starting with # or with ^. In the first case, the content of the block is accessed only if the variable is true, in the second case only if the variable is false. For example, the mustache commands '`{{^isBodyParam}}{{baseName}}{{/isBodyParam}}`' will print the variable's base name only when '`isBodyParam`' is false, while by writing '`{{#isBodyParam}}{{baseName}}{{/isBodyParam}}`' the variable name will be printed only when '`isBodyParam`' is true.
- Objects and lists also use two opening tags just like booleans, but they behave differently. If the opening tag uses #, it will either give access to the object's properties or iterate through the list's items. For example, the template '`{{#allParams}}{{baseName}} {{/allParams}}`' will iterate through all the items of '`allParams`' and, for each one of them, write the value of '`baseName`' followed by a space. If the opening tag uses ^, it will behave like a Boolean and write the content of the block only if the object doesn't exist or the list is empty. For example, '`{{^operation}}undefined{{/operation}}`' will write '`undefined`' only if the object '`operation`' doesn't exist.

With an idea of how Mustache syntax works, the example of code 4.5 becomes easy to understand. The template produces a function that pattern matches over the operation ID and for each case it returns the list of parameter names for the corresponding operation. The mustache tags on line 1 access the API specifications object and iterate through all the operations, starting a function case's definition for the current value of '`operationId`'. The function's body is only a list of atoms, with each atom corresponding to either a data type or a parameter name, depending on whether the considered parameter is part of the content body or not. Finally, the tags on line 7 close the tags that have been opened on line 1 and the function definition finishes with a predefined base case.

4.2.2 Erlang code generator

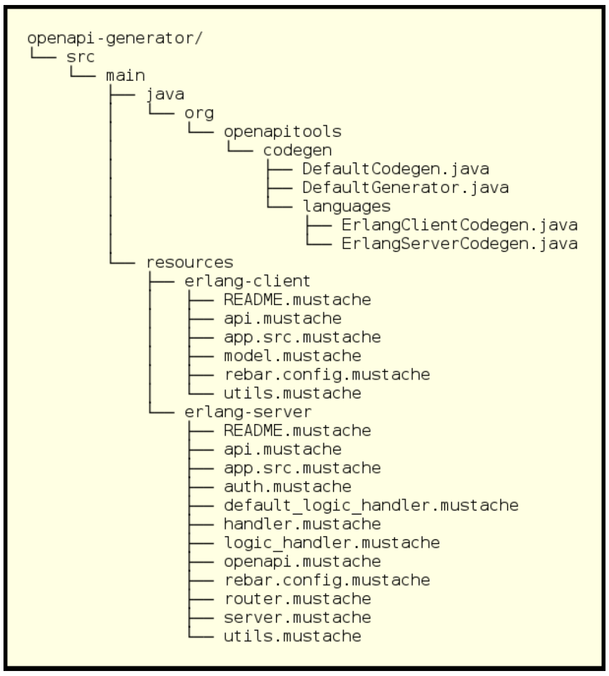
There are three different generators for Erlang code in the OpenAPI Generator: `‘erlang-server’`, `‘erlang-client’`, and `‘erlang-proper’`. Erlang proper is an Erlang client with property-based testing, but we did not use it because our examples were either very simple and did not need extensive testing, or too complex and required the support of GTT. We focused mainly on `‘erlang-server’` and occasionally used `‘erlang-client’` for sending requests when testing the code generated from simple specifications.

Among the client's files, the most important ones are:

- `‘model’`: used for generating the client's models. Models are the Erlang representation of the OpenAPI schemas.
- `‘api’`: the template of the functions which are used to send requests. There are two function definitions for each operation: one with only the required parameters and one with all the parameters. By default, these requests are sent to `‘localhost:8001’`.

For the server, the relevant files are:

- A set of templates for separating the server's functionalities, namely `‘server’`, `‘router’`, `‘handler’`, `‘logic_handler’`, `‘default_logic_handler’`, and `‘auth’`.
- `‘openapi’`: a template which creates the JSON file processed by the generator.
- `‘api’`: it contains all the functions dedicated to request and response validation. It also extracts and parses the values from requests.



```
openapi-generator/
└── src
    └── main
        ├── java
        │   └── org
        │       └── openapitools
        │           └── codegen
        │               ├── DefaultCodegen.java
        │               ├── DefaultGenerator.java
        │               └── languages
        │                   ├── ErlangClientCodegen.java
        │                   └── ErlangServerCodegen.java
        └── resources
            ├── erlang-client
            │   ├── README.mustache
            │   ├── api.mustache
            │   ├── app.src.mustache
            │   ├── model.mustache
            │   ├── rebar.config.mustache
            │   └── utils.mustache
            └── erlang-server
                ├── README.mustache
                ├── api.mustache
                ├── app.src.mustache
                ├── auth.mustache
                ├── default_logic_handler.mustache
                ├── handler.mustache
                ├── logic_handler.mustache
                ├── openapi.mustache
                ├── rebar.config.mustache
                ├── router.mustache
                ├── server.mustache
                └── utils.mustache
```

Figure 4.6: The original structures of `‘erlang-client’` and `‘erlang-server’`.

5

Details

5.1 Analysis of the generator

As mentioned previously, the only tool available for Erlang code generation with OpenAPI 3 is OpenAPI Generator, although its Erlang components are rarely maintained. This makes it necessary to analyze the features and limitations of the OpenAPI Generator to confirm whether it is at par with other languages, or needs revisions. Once we confirm that the generation works for simple specifications, we will mainly generate from the 5G Core 3GPP specifications to try to find issues and narrow them down into minimal failing specifications. With this approach single issues can be isolated, making it easier to investigate their causes and fix them when possible. We will consider the generator to be in a good state when it can successfully generate code from all the needed 5G Core specifications.

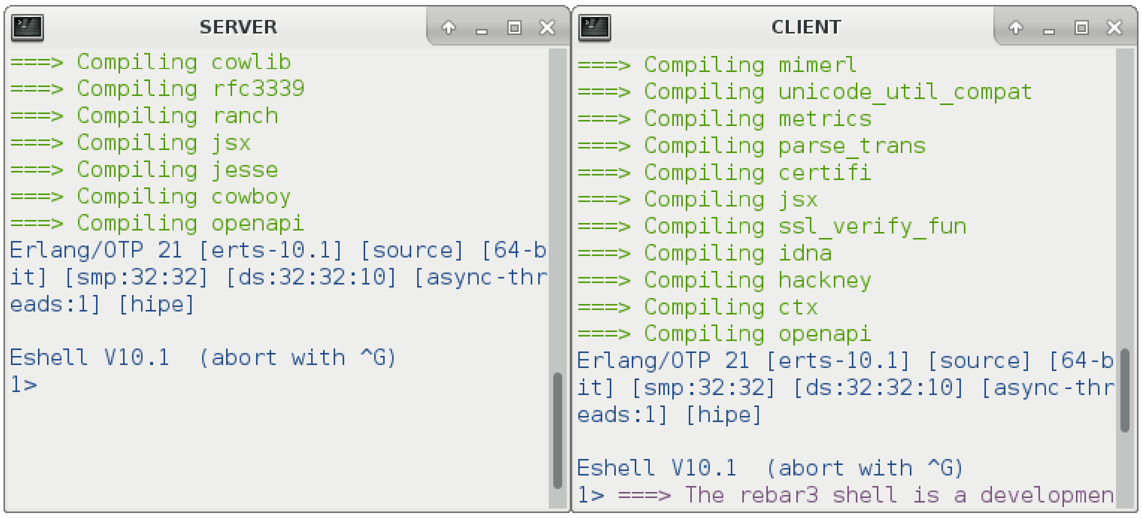
5.1.1 Testing base functionalities

We started by checking whether the generator can generate Erlang code successfully and by learning how to run it. For this purpose, we wrote specifications that only described very simple requests to see if they are sent and handled correctly by the client and the server respectively.

The source code is built with `‘mvn clean install’`, and that creates the JAR [36] file `‘openapi-generator-cli.jar’` which is used to generate code. The JAR's `‘generate’` function requires three options to work: **`‘-i’`** for the path of the input YAML file, **`‘-g’`** for the target language, and **`‘-o’`** for the output folder. There are many other options available [8], but the only ones that we used were **`‘-c’`** for language-specific configuration and **`‘-v’`** for debugging.

```
java -jar </path/to/openapi-generator-cli.jar> generate \
    -i <input-file.yaml> \
    -g <language> \
    -o <output-folder> \
    [-c <config.json>] \
    [-v]
```

Code 5.1: The skeleton of the command used for generation.



```
===> Compiling cowlib
===> Compiling rfc3339
===> Compiling ranch
===> Compiling jsx
===> Compiling jesse
===> Compiling cowboy
===> Compiling openapi
Erlang/OTP 21 [erts-10.1] [source] [64-b
it] [smp:32:32] [ds:32:32:10] [async-thr
eads:1] [hipe]

Eshell V10.1  (abort with ^G)
1>
```

```
===> Compiling mimerl
===> Compiling unicode_util_compat
===> Compiling metrics
===> Compiling parse_trans
===> Compiling certifi
===> Compiling jsx
===> Compiling ssl_verify_fun
===> Compiling idna
===> Compiling hackney
===> Compiling ctx
===> Compiling openapi
Erlang/OTP 21 [erts-10.1] [source] [64-b
it] [smp:32:32] [ds:32:32:10] [async-thr
eads:1] [hipe]

Eshell V10.1  (abort with ^G)
1> ===> The rebar3 shell is a developmen
```

Figure 5.2: Rebar3 shells after compiling the generated packages.

In this project we generate code in “two” languages, namely ‘`erlang-client`’ and ‘`erlang-server`’. They both produce a rebar package, so we will use ‘`rebar3`’ to compile and execute our tests locally. For testing, running ‘`rebar3 shell`’ in the package’s root directory is enough since it will download the dependencies, compile them together with the generated code, and set up an Erlang shell where we can either run the server or send requests as a client depending on the package.

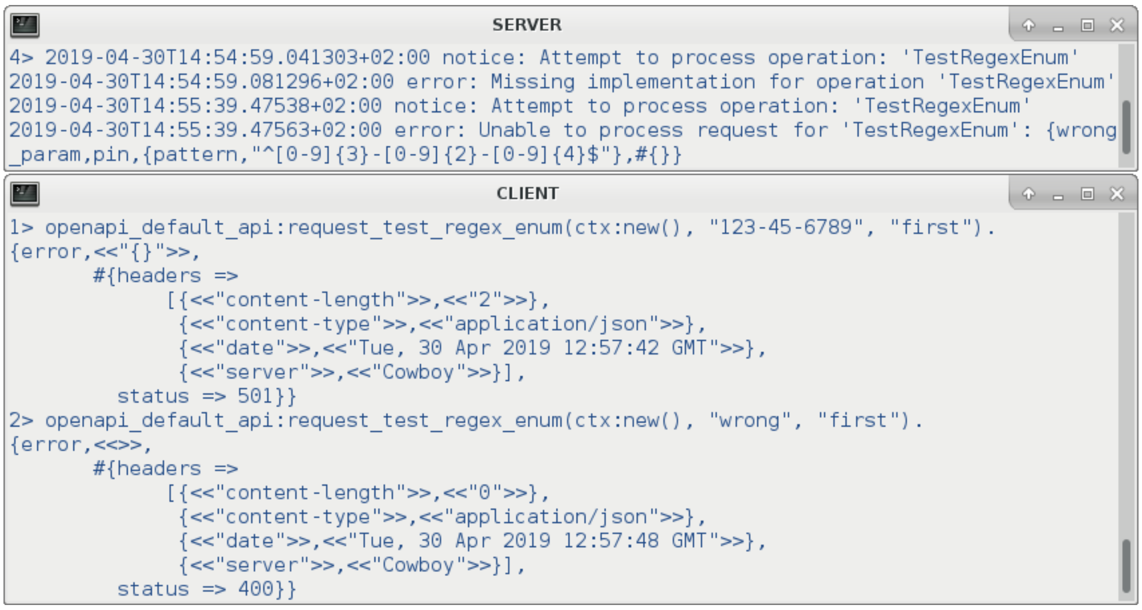
Once both of the shells are done compiling, it is possible to run the server by calling the function ‘`openapi_server:start/2`’ and send new requests from the client’s module ‘`openapi_default_api:request_<operationId>`’. The operation’s ID is obtained from either the combination of the operation’s path and HTTP method or, if defined, its homonymous YAML field.

```
% Server, Runs on localhost with port 8001
application:ensure_all_started(openapi).
openapi_server:start(server, #{ip=>{127,0,0,1}, port=>8001, net_opts=>[]}).

%Client, sends two requests (Regex and ENUM example)
%openapi_default_api:request_text_regex_enum(Ctx, Pin, OptionParam).
openapi_default_api:request_test_regex_enum(#{}, "123-45-6789", "first").
openapi_default_api:request_test_regex_enum(#{}, "wrong", "first").
```

Code 5.3: Commands used to test the generated code from 4.2.

The server side validates the requests to make sure that they follow the specifications before sending them to the logic handler. If the request is not well-formed, an error is sent as a response. For instance, ‘`415: unsupported media type`’ indicates that the content type is different from the expected JSON, while ‘`400: bad request`’ denotes that at least one of the parameters does not meet the specification’s requirements. Otherwise, if the request is correct, it is accepted and passed to the logic handler. The logic handler is where the programmer needs to implement the operation’s behavior, so it’s also where the generator will stop its work. By default, the generated code returns the error message ‘`501: not implemented`’



```
4> 2019-04-30T14:54:59.041303+02:00 notice: Attempt to process operation: 'TestRegexEnum'
2019-04-30T14:54:59.081296+02:00 error: Missing implementation for operation 'TestRegexEnum'
2019-04-30T14:55:39.47538+02:00 notice: Attempt to process operation: 'TestRegexEnum'
2019-04-30T14:55:39.47563+02:00 error: Unable to process request for 'TestRegexEnum': {wrong
_param,pin,{pattern,"^[0-9]{3}-[0-9]{2}-[0-9]{4}$"},#{}}
```

```
1> openapi_default_api:request_test_regex_enum(ctx:new(), "123-45-6789", "first").
{error,<<"{}">>,
       #{headers =>
             [{<<"content-length">>,<<"2">>},
              {<<"content-type">>,<<"application/json">>},
              {<<"date">>,<<"Tue, 30 Apr 2019 12:57:42 GMT">>},
              {<<"server">>,<<"Cowboy">>}],
         status => 501}}
2> openapi_default_api:request_test_regex_enum(ctx:new(), "wrong", "first").
{error,<<>>,
       #{headers =>
             [{<<"content-length">>,<<"0">>},
              {<<"content-type">>,<<"application/json">>},
              {<<"date">>,<<"Tue, 30 Apr 2019 12:57:48 GMT">>},
              {<<"server">>,<<"Cowboy">>}],
         status => 400}}
```

Figure 5.4: Sending a correct and an incorrect request to the server for code 4.2. First terminal: log of the server. Second terminal: the client sending requests and receiving responses.

when handling a request. Therefore, during testing we considered the error code 501 as an indicator of a well-formed request.

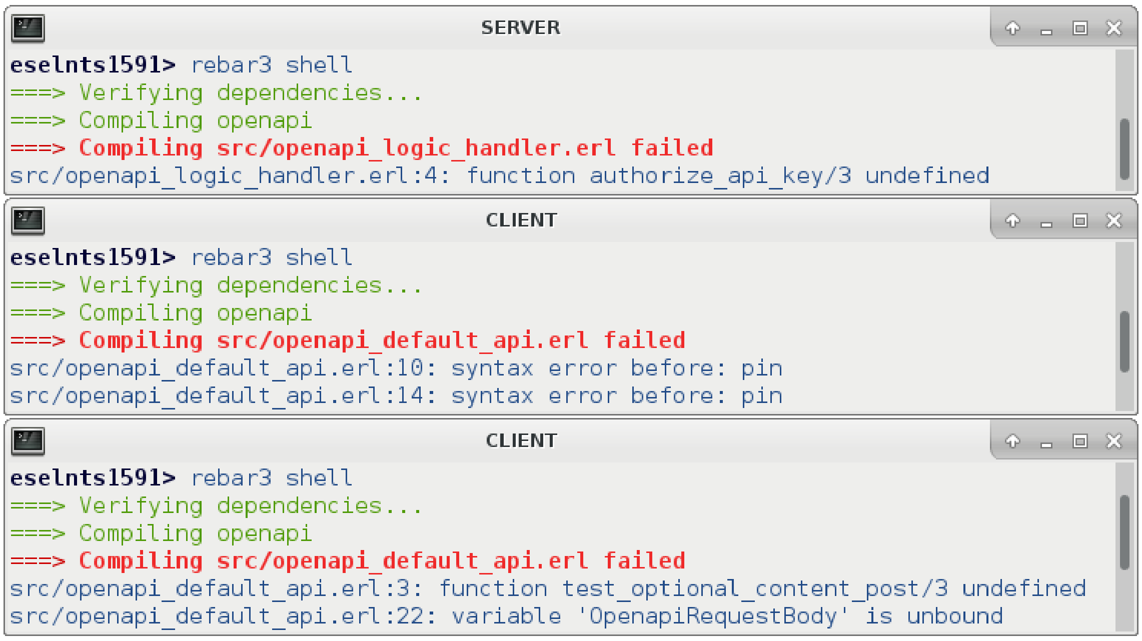
The functions for the client in code 5.3 only have the required parameters as arguments. To send the optional requests, the client must add one function argument to the function's call, the '`Optional`' map. With it it is possible to add several optional values to the request: the parameters defined in the specification that are not required, configuration variables for the HTTP client library hackney [6], or an alternative host to send the request to. Its full structure is represented in code 5.5, together with an example request where the information about to which port to send the request is given through the '`Optional`' map.

```
% Optional :: maps:map()
Optional = #{
    params => #{...}, % optional parameters
    hackney_opts => [{<name>,<value>}...], % hackney configuration variables
    cfg => #{
        host => "New target host" ,
        % hackney configuration variables, they will
        % be added to the other configuration variables
        hackney_opts => [{...}],
        % authentication settings
        auth => #{...}
    }
}
% Example of how to use the Optional parameter
openapi_default_api:request_test_regex_enum("123-45-6789", "first",
      #{hackney_opts => [{port, "8001"}]}).
```

Code 5.5: The definition of the parameter '`Optional`' with all its possible fields.

5.1.2 Initial fixes for the OpenAPI Generator

Initially, the code generated from most specifications crashed at compile time. There were mainly three errors causing this problem: an undefined function 'authorize_api_key/3' in the server's logic handler, multiple syntax errors, and undefined variables in the client's API, as shown in figure 5.6.



SERVER

```
eselnts1591> rebar3 shell
===> Verifying dependencies...
===> Compiling openapi
===> Compiling src/openapi_logic_handler.erl failed
src/openapi_logic_handler.erl:4: function authorize_api_key/3 undefined
```

CLIENT

```
eselnts1591> rebar3 shell
===> Verifying dependencies...
===> Compiling openapi
===> Compiling src/openapi_default_api.erl failed
src/openapi_default_api.erl:10: syntax error before: pin
src/openapi_default_api.erl:14: syntax error before: pin
```

CLIENT

```
eselnts1591> rebar3 shell
===> Verifying dependencies...
===> Compiling openapi
===> Compiling src/openapi_default_api.erl failed
src/openapi_default_api.erl:3: function test_optional_content_post/3 undefined
src/openapi_default_api.erl:22: variable 'OpenapiRequestBody' is unbound
```

Figure 5.6: First set of errors found, blocking the compiling of the generated code and preventing further testing.

5.1.2.1 First error: export of undefined function

In 'logic_handler.mustache', one of the server's templates, the definition of the function 'authorize_api_key/3' was added to the generated file only if the list of the authentication methods was empty or one of the methods was of type 'apiKey'. However, the case where the list of authentication methods is not empty but none of them is of type 'apiKey' was not covered. This would result in the function being exported without a definition. The fix was straightforward, it just required adding the same conditions used for the function's definition to the 'export' line.

```
% Original template
% No conditions used
-export([authorize_api_key/3]).
```

```
% If not empty and one uses API Key
{{#authMethods}}
    {{#isApiKey}}
-export([authorize_api_key/3]).
    {{/isApiKey}}
{{/authMethods}}
{{^authMethods}} % If empty
-export([authorize_api_key/3]).
{{/authMethods}}
```

Code 5.7: Left: original template. Right: template after the fix for the export.

5.1.2.2 Second error: syntax errors

The second error was caused by the incompatibility between the naming convention of OpenAPI/YAML and Erlang. In OpenAPI, it is possible (although not recommended) to name schemas, parameters, and unique IDs with strings that have digits as initials. However, this will create problems if those same names are used for variables or functions in Erlang. In the 5G Core specifications there are several parameter names and operation IDs starting with '`5G`', so when they appeared in the generated Erlang code as variable names they would produce syntax errors. A simple solution was to add a prefix to variables that start with a digit.

```
@Override
public String toVarName(String name) {
    // replaces - with _ and prepends Var_ to avoid numbers as initials
    String temp = name.replaceAll("-", "_");
    if (Character.isDigit(temp.charAt(0)))
        temp = "Var_" + temp;
    return sanitizeName(temp);
}
```

Code 5.8: The function used to convert schema names into variable names in both '`ErlangClientCodegen.java`' and '`ErlangServerCodegen.java`'.

5.1.2.3 Third error: unbound variables

Before explaining this error, it is necessary to know some details about the subclass '`ErlangClientGenerator.java`' and how it differs from the server generation. While all the server's functions have a predetermined name and number of arguments (arity), the client side needs to generate functions with varying names and arities. In fact, the client creates one function for each operation from the input specifications, where the name corresponds to the operation's ID and the arguments depend on the number of required parameters. Also, all these functions have two definitions: a complete one, with all the required parameters and the '`Optional`' map, and a short one, with only the required parameters.

To export these functions, it is necessary to know their arities. For this purpose, the client's generator has been extended with a local class which adds two properties to the mustache templates: one for the number of required parameters and one for the required parameters together with the '`Optional`' map. The problem was that the request body parameters were always counted as required no matter they were required or not, as shown in code 5.10. However, they were added as function arguments in the mustache template only when required (code 5.9).

```
% Original template
request_{{operationId}}(Ctx
    {{#allParams}}{{#required}}, {{paramName}}{{/required}}{{/allParams}}
    , Optional) -> ...
```

Code 5.9: The original template for the client's arguments

```java
// Original function in ErlangClientCodegen.java
int lengthRequired(List<CodegenParameter> allParams) {
    int l = 0;
    for (CodegenParameter o : allParams) {
        if (o.required || o.isBodyParam) l++;
    }
    return l;
}
```

Code 5.10: The original method for counting the number of required parameters.

There was a reason for the request bodies to be always counted. Even when not required, the function used the request body variable to build the request. However, it was not defined as a function's arguments unless it was required, as shown in code 5.12. This would result in the `'Unbound Variable'` error of the third terminal in figure 5.6.

```yaml
# Specification used to identify the issue
paths:                                      components:
  /test-optional-content:                     schemas:
    post:                                       RequestBody:
      requestBody: # not required                 type: object
        content:                                  properties:
          'application/json':                       val:
            schema:                                   type: string
              $ref: '#/.../RequestBody'         ...
      ...
```

Code 5.11: An extract of the specifications used to test the arity issue.

```erlang
% Extract from openapi_default_api.erl in the Erlang client
-export([test_optional_content_post/2, test_optional_content_post/3]).
% ...
test_optional_content_post(Ctx) -> test_optional_content_post(#{}).
test_optional_content_post(Ctx, Optional) ->
    ...
    Body1 = OpenapiRequestBody,
    ...
```

Code 5.12: The generated client function from code 5.11 uses the unbound variable `'OpenapiRequestBody'` as request body and it is exported with the wrong arities.

Two changes were needed to solve the issue, one in `'ErlangClientCodegen.java'` and one in the client's template `'api.mustache'`. In the java file we split the counting for arity between two functions, one for the "required only" definition and one for the "optionals" definition (code 5.13). With these two auxiliary functions, we can avoid counting optional body parameters in the arity for the "required only" function, while also maintaining the count for the complete definition.

In the Mustache template we added the body parameters as arguments of the function's complete definition, regardless of whether they are required or not. We always place the body parameters last to maintain the same logical order across the definitions, as seen in code 5.14.

Code 5.15 is the result of generating from 5.11 after the fixes. The complete function definition now includes the request body as argument, and both of the definitions are exported with the correct arity. One thing to also note is that the '`Ctx`' variable has been removed from the generated functions. This is because the variable was left unused even though it was added to every function. Another detail is that the request body is considered as missing with the empty list and as existing but empty with the empty map.

```java
// Fixed counting functions in ErlangClientCodegen.java
int lengthRequired(List<CodegenParameter> allParams) {
    int l = 0;
    for (CodegenParameter o : allParams) {
        if (o.required) l++;
    }
    return l;
}

int lengthOptional(List<CodegenParameter> allParams) {
    int l = 0;
    for (CodegenParameter o : allParams) {
        if (o.required || o.isBodyParam) l++;
    }
    return l;
}
```

Code 5.13: Auxiliary functions for counting the arity. The new '`lengthRequired`' function does not count optional request bodies.

```
% Modified template
request_{{operationId}}(Ctx
    {{#allParams}}{{^isBodyParam}}
        {{#required}}, {{paramName}}{{/required}}
    {{/isBodyParam}}{{/allParams}}
    {{#bodyParams}}, {{paramName}}{{/bodyParams}}
    , Optional) -> ...
```

Code 5.14: The modified template for the complete function definition.

```erlang
% Extract from openapi_default_api.erl in the Erlang client after the fix
-export([request_test_optional_content_post/0,
         request_test_optional_content_post/2]).
request_test_optional_content_post() ->
    request_test_optional_content_post([], #{}).
request_test_optional_content_post(OpenapiRequestBody, Optional) ->
    ...
    Body1 = OpenapiRequestBody,
    ...
```

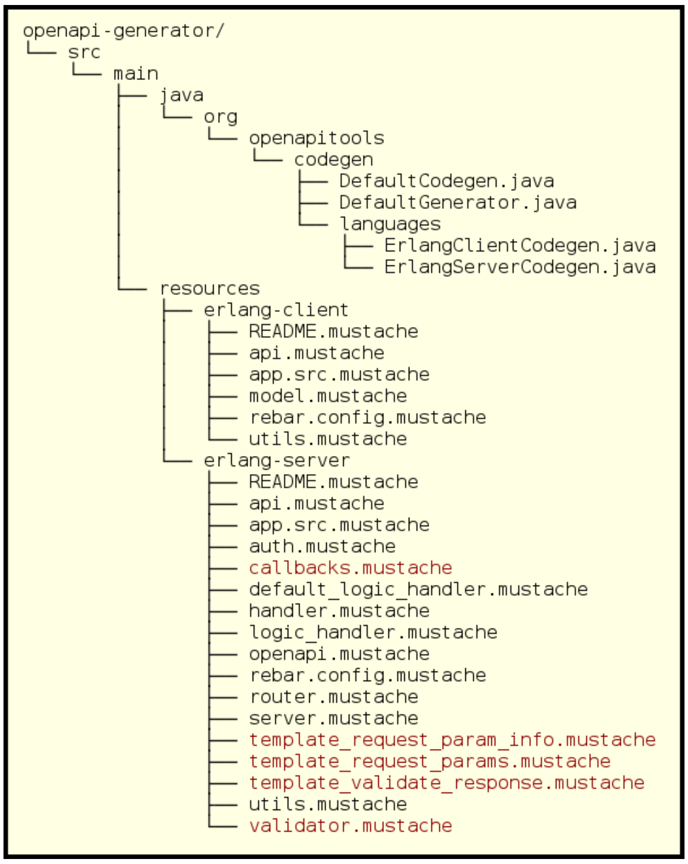
Code 5.15: The generated code after the fixes.

5.1.3 Generator's reorganized structure

To avoid some code repetition and have a better organization of the code, we restructured the '`api.mustache`' template in '`erlang-server`'.

Previously, there was no support for callbacks in the Erlang generator, although they were processed by the generator's core. To add that support, we added a separate module that can only validate callback requests rather than extending '`api.mustache`'. In this way, the module can be integrated into a system independently. More details will be introduced in later sections. Since the '`api`' and '`callbacks`' templates have almost the same code/structure for handling the parameters, we moved those functions to a separate template, '`validator.mustache`'.

What is left in '`api.mustache`' and '`callbacks.mustache`' are the functions for accessing the information about operations and parameters. These functions use the same templates but with different opening tags, since the callbacks are stored separately. Therefore, to avoid repetition, we moved the templates of the functions in separate files and included them in-between the corresponding opening tags with '`{{> filename.mustache}}`'.



```
openapi-generator/
└── src
    └── main
        ├── java
        │   └── org
        │       └── openapitools
        │           └── codegen
        │               ├── DefaultCodegen.java
        │               ├── DefaultGenerator.java
        │               └── languages
        │                   ├── ErlangClientCodegen.java
        │                   └── ErlangServerCodegen.java
        └── resources
            ├── erlang-client
            │   ├── README.mustache
            │   ├── api.mustache
            │   ├── app.src.mustache
            │   ├── model.mustache
            │   ├── rebar.config.mustache
            │   └── utils.mustache
            └── erlang-server
                ├── README.mustache
                ├── api.mustache
                ├── app.src.mustache
                ├── auth.mustache
                ├── callbacks.mustache
                ├── default_logic_handler.mustache
                ├── handler.mustache
                ├── logic_handler.mustache
                ├── openapi.mustache
                ├── rebar.config.mustache
                ├── router.mustache
                ├── server.mustache
                ├── template_request_param_info.mustache
                ├── template_request_params.mustache
                ├── template_validate_response.mustache
                ├── utils.mustache
                └── validator.mustache
```

Figure 5.16: The reorganized generator.

5.1.4 Specification validation

After ensuring that the generated code can compile successfully, next step is to test whether it validates the requests and responses according to the specifications. We started with checking simpler cases that used to be problematic (such as regular expressions and enumerations). Afterwards, we used the 5G Core specifications to find more complex cases and test them individually.

5.1.4.1 Regex and ENUMs

Regular expressions (regexes) and enumerations (ENUMs) are among the most commonly used OpenAPI features. We can observe that in '`CommonData`', a YAML file which collects common schemas referenced throughout all the 5G Core specifications. Among those schemas, 89 out of 196 have either regexes or ENUMs in their own inline definition, and most of the remaining ones use them through reference. Therefore, it is necessary to ensure that both regexes and ENUMs are verified correctly wherever they are in the request and whether they are referenced or not.

To test them, we wrote our own OpenAPI 3 specification that defined regexes and ENUMs in all the possible locations of a request. After validating the specifications, we ran the generated code and sent both correct and erroneous values to confirm that they were validated correctly or find possible bugs. We discovered that:

- Regular expressions were always marked as incorrect, even with correct inputs;
- Referenced ENUM parameters would consider any value as valid.

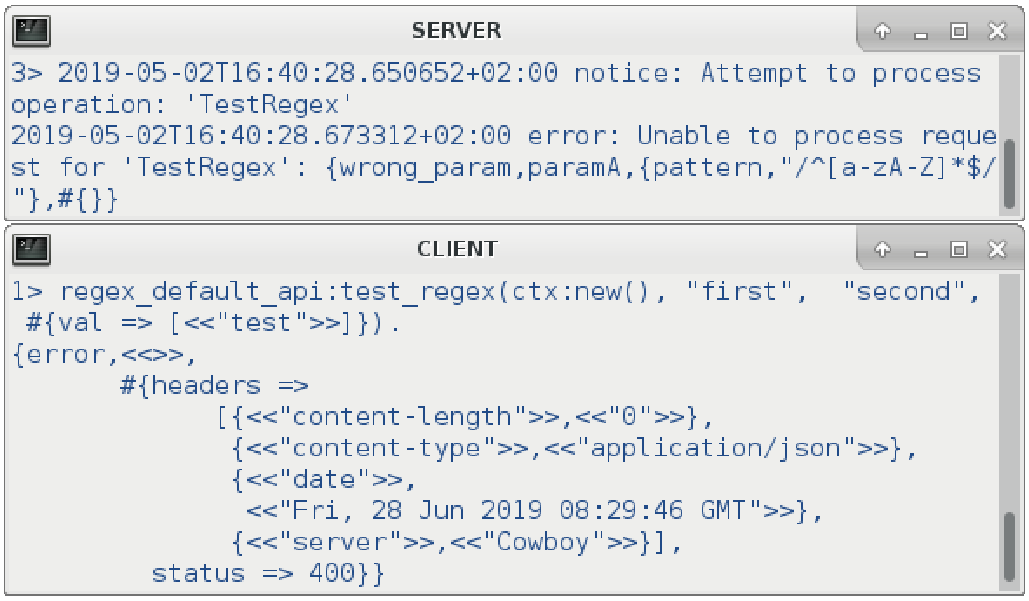
The bug that caused regular expressions to fail was shown directly by the server's log (figure 5.19), as the pattern used to check the input string was surrounded by two slashes '`/`'. These characters were there because the generator follows the ECMAscript standard for marking regular expressions. It requires the pattern to be surrounded by those delimiters instead of double quotation marks. However, Erlang does not follow the same standard, so the slashes were considered as part of the pattern string. Overriding the '`DefaultCodegen`''s function '`addRegularExpressionDelimiter`' (code 5.17) was enough to fix the problem.

```
@Override
public String addRegularExpressionDelimiter(String pattern) {
    //if (!pattern.matches("^/.*"))
    //    return "/" + pattern.replaceAll("/", "\\\\/") + "/";
    return pattern;
}
```

Code 5.17: The function in '`DefaultCodegen`' to override to add (or remove) language-specific delimiters for regular expressions. The commented code is what caused the issue.

```
/test:                                      components:
  post:                                       schemas:
    operationId: TestRegex                      Word:
    parameters:                                   type: string
      - name: paramA                              pattern: '^[a-zA-Z]*$'
        in: query
        required: true
        schema:
          $ref: '#/components/schemas/Word'
      # ...
```

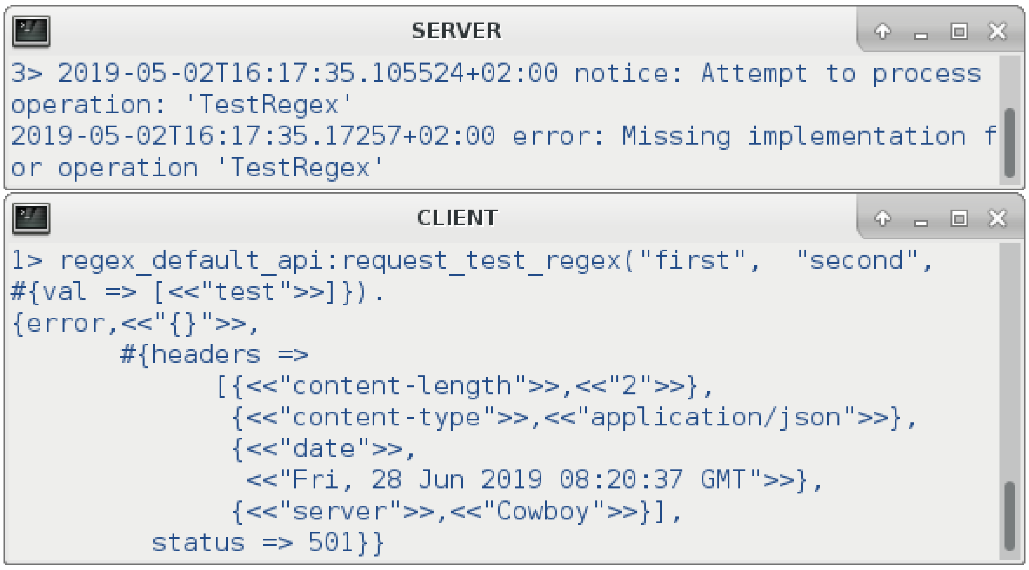
Code 5.18: The extract from the example case for testing regular expressions.



```
SERVER
3> 2019-05-02T16:40:28.650652+02:00 notice: Attempt to process
operation: 'TestRegex'
2019-05-02T16:40:28.673312+02:00 error: Unable to process reque
st for 'TestRegex': {wrong_param,paramA,{pattern,"/^[a-zA-Z]*$/
"},#{}}
```

```
CLIENT
1> regex_default_api:test_regex(ctx:new(), "first",  "second",
 #{val => [<<"test">>]}).
{error,<<>>,
       #{headers =>
             [{<<"content-length">>,<<"0">>},
              {<<"content-type">>,<<"application/json">>},
              {<<"date">>,
               <<"Fri, 28 Jun 2019 08:29:46 GMT">>},
              {<<"server">>,<<"Cowboy">>}],
         status => 400}}
```

Figure 5.19: Regular expressions were always failing for parameters. For example, ‘`first`’ follows the pattern ‘`^[a-zA-Z]*$`’, but the server marks it as incorrect.



```
SERVER
3> 2019-05-02T16:17:35.105524+02:00 notice: Attempt to process
operation: 'TestRegex'
2019-05-02T16:17:35.17257+02:00 error: Missing implementation f
or operation 'TestRegex'
```

```
CLIENT
1> regex_default_api:request_test_regex("first",  "second",
#{val => [<<"test">>]}).
{error,<<"{}">>,
       #{headers =>
             [{<<"content-length">>,<<"2">>},
              {<<"content-type">>,<<"application/json">>},
              {<<"date">>,
               <<"Fri, 28 Jun 2019 08:20:37 GMT">>},
              {<<"server">>,<<"Cowboy">>}],
         status => 501}}
```

Figure 5.20: By removing the surrounding ‘`/`’, the request is processed correctly.

The error for ENUMs was also fairly easy to identify, but its cause was less obvious. Code 5.21 lists two examples of the specifications for two ENUM parameters. The only difference between them is that '`paramA`' has an external reference, while '`paramB`' does not. However, the referenced schema '`EnumString`' has the same content as the inline schema for '`paramB`'. Therefore, we expected these two parameters to have the same generated code, but it turned out that '`paramA`' did not have the details about the type and the list of accepted values.

This happened because schemas are processed differently depending on whether they are referenced or not. The schemas with inline definitions are considered as ENUMs themselves, as the properties are attributed directly to them. On the other hand, parameters with referenced ENUM schemas should not be considered as an ENUM parameter, but just as a parameter with a referenced schema. This difference becomes problematic with the template's implementation, as it would expect both referenced and inline ENUM schemas to be validated through the ENUM rule (code 5.22). These rules are used in the generated code to validate parameter properties such as regular expressions and boundaries on list sizes.

```
% Erlang functions
request_param_info('TestEnum', 'paramA') ->
    #{
        source => qs_val,
        rules => [
            required
        ]
    };

request_param_info('TestEnum', 'paramB') ->
    #{
        source => qs_val,
        rules => [
            {type, 'binary'},
            {enum, ['fst', 'snd', 'trd'] },
            required
        ]
    };
```

```
# Spec for paramA
- name: paramA
  in: query
  required: true
  schema:
    $ref: '#/.../EnumString'
    # enum: [fst, snd, trd]

# Spec for paramB
- name: paramB
  in: query
  required: true
  schema:
    type: string
    enum: [ fst, snd, trd ]
```

Code 5.21: The generated code for two ENUM parameters.

```
validate(Rule = {enum, Values}, Name, Value) ->
    try
        FormattedValue = erlang:binary_to_existing_atom(Value, utf8),
        case lists:member(FormattedValue, Values) of
            true -> {ok, FormattedValue};
            false -> validation_error(Rule, Name)
        end
    catch
        error:badarg -> validation_error(Rule, Name)
    end;
```

Code 5.22: The case for ENUMs in the validation function.

The different processing results in the flag ‘`isEnum`’ being set to false for referenced ENUM schemas. We decided to change how the referenced ENUMs were handled, so that the parameter could be considered an ENUM. Some changes to ‘`DefaultCodegen.java`’ were needed to achieve this:

1. setting the ‘`isEnum`’ flag to true when referencing an ENUM schema;
2. checking the type of the referenced schema when it is processed.

The first point only needed a single statement to set the value of ‘`isEnum`’ to true. For the second point, we took advantage of the already existing type checking for the properties and moved it in a separate function, ‘`setPropertyType`’. We then used that function to set the property’s type based on the referenced schema’s type. The changes are represented by code 5.23, excluding the ‘`setPropertyType`’ definition, and they were enough to produce the expected result in code 5.24.

```java
setPropertyType(property, referencedSchema, name); //added
if (referencedSchema.getEnum() != null
    && !referencedSchema.getEnum().isEmpty()) {
    List<Object> _enum = referencedSchema.getEnum();

    property.isEnum = true;         //added

    Map<String, Object> allowableValues = new HashMap<String, Object>();
    allowableValues.put("values", _enum);
    if (allowableValues.size() > 0) {
        property.allowableValues = allowableValues;
    }
}
```

Code 5.23: The changes in ‘`DefaultCodegen.java`’

```erlang
request_param_info('TestEnum', 'paramA') ->
    #{
        type => 'binary',
        source => 'qs_val',
        rules => [
            { enum, ['fst', 'snd', 'trd'] },
            required
        ]
    };
```

Code 5.24: The resulting generated code for ‘`paramA`’ after the fix.

5.1.4.2 List Parameters

In the 5G Core, NRF Discovery is one of the network functions that has the most parameters for a single operation. It has 37 different parameters for a single GET request and many of them are lists whose items have referenced schemas. List parameters were not supported at all in the original mustache templates. In fact, after receiving a list parameter, the server would only use the first element of the list as value and ignored the rest of the elements in the list.

To improve the implementation, the server needed to:

- add '`list`' to the parameter type in file '`validator.mustache`';
- read and concatenate multiple values for a list parameter;
- have a case in the '`typecheck`' function to validate list parameters.

We defined the type for lists as a pair '`{list, Type}`', where '`Type`' is the type of the list's elements. We structured it like this to validate the type of both the list and its elements.

The function '`get_value`' in '`validator.mustache`' is used to read the requests and it can distinguish the cases depending on the parameter's location (body, query, header, or binding). To avoid making the reading of other parameters more complicated, we added a separate case only for lists which uses the correct request reader depending on the list's location. Then, with a recursive function, we extracted all the parameters that have the same name from the list and concatenate their values into a list. Both functions are shown in code 5.25.

```
% validator.mustache
get_value(Source, Name0, {'list', _}, Req, _BodyReader) ->
    {Name, List} = case Source of
        'qs_val' ->
            { {{packageName}}_utils:to_qs(Name0),
                cowboy_req:parse_qs(Req) };
        'header' ->
            { {{packageName}}_utils:to_header(Name0),
                cowboy_req:headers(Req) };
        'binding' ->
            { {{packageName}}_utils:to_binding(Name0),
                cowboy_req:bindings(Req) }
    end,
    { {{packageName}}_utils:extract_list(Name, List), Req };

% utils.mustache
extract_list(_Name, []) -> [];
extract_list(Name, List) ->
    case lists:keytake(Name, 1, List) of
        false -> [];
        {_, {_, Value}, List1} -> [Value | extract_list(Name, List1)]
    end.
```

Code 5.25: The added case and new function for reading list parameters.

Once the list is extracted from the request, what is left is to validate it. To do that we first check whether the value is a list, then map the '`typecheck`' function with the paired item's type in '`{list, Type}`' to the list of parameters. In this way we validated both the list type and its elements.

```
% validator.mustache
typecheck(Type = {'list', ItemType}, Name, Value, ValidatorState) ->
    case is_list(Value) of
        true ->
            lists:map(fun(Val) ->
                typecheck(ItemType, Name, Val, ValidatorState)
            end, Value),
            ok;
        false -> validation_error(Type, Name)
    end;
```

Code 5.26: The added case for validating list parameters.

5.1.4.3 Names mismatch

We went through the 5G core specifications and noticed that not all the schemas are referenced from ‘`components`’. Sometimes request bodies are very short and unique, so they are defined inline rather than with a reusable schema. This causes issues when the generator processes them into java objects, since they do not have a unique identifier. What the generator does is to create additional schemas named ‘`inline_object`’ whenever a request body has an inline definition, with an incremental number appended in the case of multiple inline definitions, such as ‘`inline_object_2`’.

However, the java objects processed from the JSON specifications have their names camelized [10] to follow the naming standards for java classes, converting names like ‘`inline_object`’ into ‘`InlineObject`’ for example. This difference results in a mismatch between the schema names in the JSON and the ones used by the Mustache files (code 5.27), which prevents the data from being validated as it cannot be associated with any schema. This issue did not happen only with ‘`inline_objects`’, but also with any schema name that had an underscore in it.

```
% openapi_api.erl
request_param_info
    ('RootPost', 'InlineObject') ->
    #{
        source =>   body,
        rules => [
            schema,
            required
        ]
    };
```

```
# openapi.json
"inline_object" : {
  "required" : [ "val" ],
  "type" : "object",
  "properties" : {
    "value" : {
      "type" : "string"
    }
  }
}
```

Code 5.27: The generated code’s schema name is different from the real schema name in the JSON, making the validation fail.

After going through the generator’s log information, we found out that the java objects representing the schemas did not have any reference to the original schema name, but rather they used only the camelized version. We considered adding a field ‘`schemaName`’ to the models, but that would have required changes across the whole system and need to be tested for all the available languages. What we did instead was to use the list of processed models to create a map from model name to schema name in the ‘`validator`’ template (code 5.28). This map is then used to get the correct name before validating the schemas, solving the mismatch issue (code 5.29).

```erlang
% openapi_validator.erl
-define(MAP_SCHEMAS, #{
    'InlineObject' => 'inline_object',
}).
```

Code 5.28: The generated map from model name to schema name.

```erlang
% openapi_validator.erl
typecheck(Type, Name, Value, ValidatorState) ->
    SchemaName = maps:get(Type, ?MAP_SCHEMAS),
    Definition =
        list_to_binary("#/components/schemas/"
                    ++ {{packageName}}_utils:to_list(SchemaName)),
    try
        validate_with_schema(Value, Definition, ValidatorState),
        % rest of the function ...
```

Code 5.29: The map used to fetch the correct schema name.

5.2 Integration in an existing system

With all the modifications to the generator, it becomes possible to generate Erlang code from all the 5G Core specifications. The next step is to understand if it is possible to integrate the code into a system and how.

The code generated through ‘`erlang-client`’ did not seem ideal for integration. It could only provide a set of functions for sending requests, and it did not provide any consistent form of validation on the provided values. In fact, it used only the function’s type signature to check whether the values of the arguments were of the correct type. To represent the schemas as types, it used models which do not support composed schemas. Because of that, the Erlang files produced from the ‘`model`’ template would often have empty definitions.

On the other hand, ‘`erlang-server`’ seemed to be more promising. Even though the server-related components it generated were all interconnected and not that useful for an already working system, they could be customized and optimized for the service’s needs. For instance, the request and response validation of the server appeared to be reusable. Validation is tightly related to the specifications, so its generated code can check whether the requests/responses follow their specifications strictly. Moreover, the validation is an interesting component to generate because implementing it manually can be very long and repetitive, making it particularly prone to bugs.

In the generated code, the request validation is executed through a single exported function, ‘`populate_request`’, with three required arguments: the operation’s ID, the request object, and a state of Jesse Schema Validator [16]. Code 5.30 is an example of the specification of ‘`populate_request`’.

- The operation ID is a string either defined in the specifications or, if missing, obtained by the pairing of path and http method.
- The request object is an instance of ‘`cowboy_req`’, a type exported by the HTTP server library Cowboy [17].
- The Jesse Validator State is used to validate more complex schemas. It is initialized through the function ‘`prepare_validator`’ in ‘`router.mustache`’.

The advantage is that the function does not have any dependency on the generated server components, so it can validate any request in other systems as long as it has the correct ‘`OperationID`’ and Jesse State. The same concept applies for the response validator, as the arguments required by the validating function can be obtained independently of the system it is in. This allows the reuse of the generated code for validation and confirms that we can actually integrate it into an existing system. In this thesis, we will only focus on the request validator.

The following sections will demonstrate how the generated code can be integrated into an existing system, how we integrated it into Ericsson’s GTT, and how we improved it to make integration more flexible to various systems.

```
-spec populate_request(
    OperationID :: atom(),
    Req :: cowboy_req:req(),
    ValidatorState :: jesse_state:state(),
    BodyReader :: {{package_name}}_validator:body_reader()
    ) -> {ok, Model :: #{}, Req :: cowboy_req:req()}
       | {error, Name :: atom(), Value :: any(),
          Reason :: any(), Req :: cowboy_req:req()}.
```

Code 5.30: The specification of ‘`populate_request`’ with the optional parameter ‘`BodyReader`’ introduced in section 5.2.2.1.

5.2.1 Tutorial - Integrating into an existing system

Ideally, the validation should be a filter for the requests that have been received by the server. If a request is not valid, it should be discarded immediately. Otherwise, it should be processed and sent from the handler to the logic handler, which will execute the request’s operation. This behaviour is achieved by integrating the request validation in the handler, right after it receives the request from the router.

Figure 5.31 represents the conceptual integration of the request validator. In the handler, the requests are validated through the function ‘`populate_request`’ (code 5.30). This function is identical in both the ‘`api`’ and the ‘`callbacks`’ module, but its related functions are generated differently depending on the operations. It is

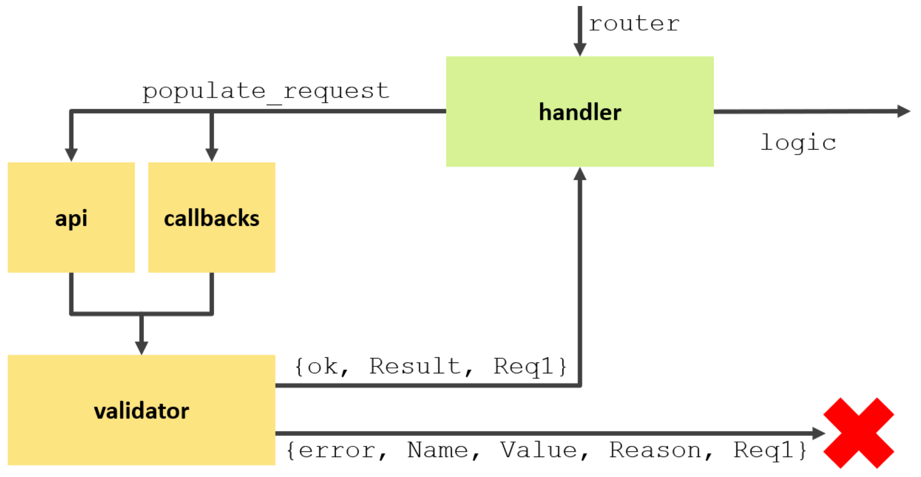


Figure 5.31: Representation of the integration of the generated code into an existing system.

possible to choose which one to use based on the handler's needs, but the request's elements are ultimately validated through the '`validator`' module.

The validator needs to have the correct '`operationID`' when validating each operation as it is required for fetching the list of request parameters and their structures. It is not always possible to get this ID automatically, so there might be the need to assign it manually to the handler's validator. Also, the Jesse JSON validator needs to be configured with the system specification's processed JSON, which can be done through the function '`prepare_validator`'. With these two elements, '`populate_request`' is ready to validate any Cowboy request.

The request validator returns either an OK, with '`Result`' containing all of the request's parameters, or an error, with information about the problem for debugging. Through a case analysis on the result it is possible to change the execution process, for example, terminating the request handling with an error or calling the logic handler if everything is correct.

To set up the validation, it is necessary to put the following generated files into the existing system's folders:

- '`api`' and '`callbacks`' for '`populate_request`'. It is not necessary to have them both since the choice of which one to use depends on the use case, and they are independent of each other, but they have the same dependencies.
- '`validator`' and '`utils`' since they contain most of the functions for processing and validating the requests.
- '`openapi.json`' is needed to configure the Jesse validator, while '`router`' is useful as it provides the function '`prepare_validator`' which initialize a Jesse state with the correct JSON. However, if the generated code is used outside of its rebar package, it is necessary to replace '`get_openapi_path()`' with the absolute path of '`openapi.json`' (code 5.32). Alternatively, an instance of '`jesse_state`' can be initiated manually as long as it points to the correct JSON file.

```
prepare_validator() ->
    R = jsx:decode(element(2, file:read_file(get_openapi_path()))),
    jesse_state:new(R, [{default_schema_ver,
                        <<"http://json-schema.org/draft-04/schema#">>}]).
```

Code 5.32: The definition of '`prepare_validator`' from the template '`router`'.

```
handler(Req) ->
    OpID = 'OperationID',
    Validator = {{packageName}}_router:prepare_validator(),
    case {{packageName}}_api:populate_request(OpID, Req, Validator) of
        {error, Name, Value, Result, Req1} ->
            ?DEBUG("Faulty request for ~p.", [OpID]),
            report_error(OpID, Name, Value, Result, Req1);
        {ok, Result, Req1} ->
            ?DEBUG("Valid request for ~p.", [OpID]),
            %... rest of the function
```

Code 5.33: An example of case analysis on the validator's result in a handler.

Code 5.33 is an abstract example of the integration of the validator. In this case we assume that '`prepare_validator`' has been modified so that it points to the correct location of the specifications '`openapi.json`'. We then use that and the corresponding operation ID to call the function to do a case analysis on its result. In case of an error, the execution will be interrupted and the error is reported back to the client with a '`report_error`' function. Otherwise, it can be processed as previously implemented with the processed data '`Result`'.

5.2.1.1 Our case: integration in GTT

To put our theory into practice, we generated and integrated code into Ericsson's internal test tool GTT, which simulates the Network Core system by running the core components locally as virtualized nodes. For each of these virtualized nodes there is a handler, where the requests are processed, and a logic file, where the operation is executed with the requested information. These files are part of the components' structure, so they are related to the system and not specific to one test case. In fact, by extending their logic handler, we can verify all the communications happening among the extended components, such as the ones represented in the example scenario 2.2. For example, when the extended NRF receives a request for function discovery, it will validate it regardless of whether it is from AMF or any other component. This makes our integration more meaningful: even though our implementation is on a test environment, it could be replicated on to the handlers of the live system to prevent it from handling incorrect requests.

When integrating code for multiple 5G Core components, we could not use the default configuration for the generator as that would produce modules with the same name. To avoid this, we took advantage of the '`-c`' flag to assign unique names to the generated modules. For example, for AMF Communication we used the JSON `{"packageName": "amf_c", "openAPISpecName": "amf_c"}`, prepending `‘amf_c’` to the name of the generated modules. This allowed us to copy all the needed files into GTT directly without any name overlapping.

Overall, we generated code from 8 specification files (the list can be found in chapter 6) and integrated it into the components following the structure of code 5.33. In the case analysis, we simply printed a message when the validation produced a negative result instead of using a secondary function like in code 5.33. For GTT, having the process closing without any result was enough to consider the test as failed.

Sometimes, the variable names used in the internal code would be slightly different from the names in the specifications (for example `‘ctxId’` instead of `‘ctxUserId’`), causing the validation to fail. When that was the case, we would manually rename the variable in the specifications and regenerate the code. Another problem was assigning the correct `‘operationID’` to each handler. To understand which operation ID to use, we had to manually compare the specifications with the implementation and the generated code. Once identified, we could connect the correct ID to the handler and initiate the validation.

Finally, we used the tests of GTT to confirm whether the validation was working. Every time we integrated code in a new component, we would find relevant tests that use the component's operations and ran them locally. Once the tests have passed, we ran the internal system-wide regression test to confirm that everything was working correctly.

5.2.2 Problems with integration and improvements

5.2.2.1 Customized body reader

During the integration in GTT, some improvements to the generator became necessary. The generated code, although correct, was not flexible enough as it required manual changes and lacked some needed functionalities during integration. For example, the generated code couldn't adapt to some inconsistencies of the GTT manual implementation.

An example of that is the different decoding used for maps in different components. All the request bodies need to be decoded before being used by either the generated code or GTT. Since in the specifications most of the request bodies are represented by JSON objects, they need to have objects as schemas in the specifications, which are ultimately decoded into Erlang maps. By default, the generator decoded the keys of the maps into atoms, which worked correctly for the first component we extended as it also used atoms. However, the second component that we extended decoded the keys into binary strings instead. This small difference was enough to make the system crash if the validator parsed the request using a different decoding.

Another shortcoming regarding request bodies was the lack of support for more complex media types such as `‘multipart/form-data’`. Several components expected this kind of request body, but the generated code could only read the request body if it was an ‘application/json’. Because of this, the validator would either read only part of the request body, discarding the rest, or produce an error.

To solve both of the problems, we converted ‘populate_request’ into a higher-order function in which it is possible, although not required, to provide a customized body reader function (code 5.35). The body reader function expects a request object and returns the content of the request body together with the processed request object. This provides much more flexibility in the integration as it becomes possible to adapt the generated code to the system’s needs, such as changing the decoding for map keys or making the validator read a request body with media type ‘multipart/form-data’.

```
% validator.mustache
-type body_reader() :: fun((Req0 :: cowboy_req:req())
                      -> {ProcessedBody :: any(), Req :: cowboy_req:req()}).
```

Code 5.34: The type signature for body reader functions.

```
% api.mustache and callbacks.mustache
-spec populate_request(
    OperationID :: {{packageName}}_validator:operation_id(),
    Req :: cowboy_req:req(),
    ValidatorState :: jesse_state:state(),
    BodyReader :: {{packageName}}_validator:body_reader()
    ) -> {ok, Model :: #{}, Req :: cowboy_req:req()}
       | {error, Name :: atom(), Value :: any(),
          Reason :: any(), Req :: cowboy_req:req()}.
```

Code 5.35: The extended definition for ‘populate_request’ with an additional argument for the body reader function.

5.2.2.2 Callback module

Another feature that was missing during the integration was the support for callbacks. In GTT, the implementation for UDM had only one request handler, and that handler was for an AMF Event Exposure callback. Because of that, there was no way to validate that request with the existing code. This was not because of the generator’s core implementation, but the poor maintenance of the Erlang templates.

The generator stored the callbacks information into a ‘callbacks’ field for each operation. There was no clear indication of how to access that information, so we had to go through the Java classes to understand the right combination of the needed mustache tags. Eventually, we found the right combination of tags that would allow us to iterate through all the callbacks, as shown in code 5.36.

```
{{#apiInfo}}{{#apis}}{{#operations}}{{#operation}}
{{#callbacks}}{{#urls}}{{#requests}}
{{> template_to_import}}
{{/requests}}{{/urls}}{{/callbacks}}
{{/operation}}{{/operations}}{{/apis}}{{/apiInfo}}
```

Code 5.36: The mustache code to access the callback definitions.

With this knowledge, we could add support to the Erlang templates. Since we were only interested validating the callback requests rather than sending them, the only problem left was how to integrate it. The obvious approach seemed to extend the '`api.mustache`' by making it not only validate normal requests, but callbacks as well. Nevertheless, having the validation for all the requests of one component in a single file is never needed because the callback request is received and validated in different component(s). For example, in the 5G Core, the AMF Event Exposure callbacks are sent by AMF, but they are received and validated by either UDM or SMF.

This was the reason why we made a new Erlang template, '`callbacks.mustache`'. Having a separate file allows the programmer to integrate the '`callbacks`' module independently of the '`api`' and it makes it clearer to see which requests are expected by the system. However, because these two templates perform almost the same tasks, they would have nearly identical code. To avoid such repetition, as mentioned previously in section 5.1.3, we split the original '`api.mustache`' into two templates, '`api`' and '`validator`'. The latter one contains all the functions and definitions needed by both '`api`' and '`callbacks`'.

The rest of the functions in '`api`' and '`callbacks`' are either responsible for starting the validation or describing the operations and their parameters. For example, the function '`request_param_info`' returns the information about a parameter (location, type, properties...) given the operation ID and the parameter name. Although the access tags to the processed data are different, the structure of the function definition is the same in both files. To avoid this repetition, we moved the mustache definitions into an external template, which are included through the tag '`{{> template}}`' (code 5.36).

With this new module, it becomes possible to integrate the validation for callbacks alongside or independently of the normal requests. By using the same function names, the implementation is very similar in both cases, with the only difference being the module used by the functions. An example of integrating a callback request validation is shown in code 5.37.

```erlang
udm_handler_amf_event(Req) ->
    OpID = 'OperationID',
    % Callback from AMF Event Exposure (amf_ee)
    Validator = amf_ee_router:prepare_validator(),
    case amf_ee_callbacks:populate_request(OpID, Req, Validator) of
        {error, Name, Value, Result, Req1} ->
            ?DEBUG("Faulty request for ~p.", [OpID]),
            report_error(OpID, Name, Value, Result, Req1);
        {ok, Result, Req1} ->
            ?DEBUG("Valid request for ~p.", [OpID]),
            %... rest of the function
```

Code 5.37: An example of integration of callbacks validation.

6
Results

With the integrated validators we were able to locate bugs and implementation inconsistencies in the system. Even though the generator struggles with some features, it is still able to generate functional components that can be useful for commercial-level systems. In fact, it resulted to be more effective to generate small functional components rather than fully-functioning server systems, since more advanced applications require major optimization and customization of the server components.

6.1 Results of the integration

To check whether our request validator could successfully validate the requests sent among components, we ran the integrated components in GTT. However, the system of the 5G Core in Ericsson was not fully implemented. Thus, we integrated the request validator only in the components with a more complete implementation and adequate test cases. The chosen components, the related OpenAPI 3 specifications, and the results of running the tests in GTT after integration are listed below (the specifications are from 3GPP 5G Core December 2018 release):

Components	Specifications	Results
AMF	`TS29518_Namf_Communication.yaml`	Passed
	`TS29518_Namf_EventExplosure.yaml`	Passed
AUSF	`TS29509_Nausf_UEAuthentication.yaml`	Failed
NRF	`TS29510_Nnrf_NFDiscovery.yaml`	Passed
	`TS29510_Nnrf_NFManagement.yaml`	Failed
NSSF	`TS29531_Nnssf_NSSelection.yaml`	Passed
PCF	`TS29507_Npcf_AMPolicyControl.yaml`	Passed
SMF	`TS29502_Nsmf_PDUSession.yaml`	Passed
UDM	`Callback from AMF EventExposure`	Passed

As shown in the table above, the test cases of the AUSF and the NRF failed. In other words, the integrated request validator managed to find issues in the communications between these two components. With the help of the testing tool, we discovered the invalid requests in the test cases which are not consistent with the specifications.

The issue with AUSF is that the value assigned to the `‘ResStar’` schema in the PUT request for path `‘/ue-authentications/ctxid/5g-aka-confirmation’` does not match the length defined in the specification. The `‘ResStar’` schema is referenced in the request body and its specification is shown in 6.1. It says that the value of `‘ResStar’` should be a string consisting of 32 characters which can be any digit between `‘0’` and `‘9’` and any letter between `‘a’` and `‘f’`, either lower case or capitalized. However, the validator found out that the system assigned as value of `‘ResStar’` a string with only 16 characters.

```
ResStar:
    type: string
    pattern: '[A-Fa-f0-9]{32}'
```

Code 6.1: The schema of ResStar.

The issue in the NRF’s `‘NFManagement’` was a bit less obvious to identify. The invalid request was a POST request under path `‘/subscriptions’` which required to have the object schema `‘SubscriptionData’` as request body. One of its properties, `‘subscrCond’`, is defined as follows:

```
subscrCond:
  oneOf:
    - $ref: '#/components/schemas/NfInstanceIdCond'
    - $ref: '#/components/schemas/NfTypeCond'
    - $ref: '#/components/schemas/ServiceNameCond'
    - $ref: '#/components/schemas/AmfCond'
    - $ref: '#/components/schemas/GuamiListCond'
    - $ref: '#/components/schemas/NetworkSliceCond'
    - $ref: '#/components/schemas/NfGroupCond'
```

This property is a composed schema that selects only one of the listed schemas under the keyword `‘oneOf’`. To see the issue more clearly, we printed the value of the request in the test case, and the value for the `‘subscrCond’` schema was written as:

```
<<"subscrCond">> => #[<<"nfTypeCond">> => #[<<"nfType">> => <<"SMF">>]]
```

Apparently, the selected schema for `‘subscrCond’` is `‘NfTypeCond’`. `‘NfTypeCond’` is an object with only one ENUM property, `‘nfType’`, which is assigned the value `‘SMF’` in this case. The expected value for `‘subscrCond’` is a direct mapping from the property name to its final referenced schema:

```
<<"subscrCond">> => #[<<"nfType">> => <<"SMF">>]
```

By comparing the expected and actual structure, we can see that the problem is an additional mapping from `‘subscrCond’` to `‘nfTypeCond’`. The implementation for the NRF and the test cases do not follow the correct way of representing the `‘oneOf’` feature.

6.2 Discussion

Following the idea of integrating the request validator in the existing system, we can draw the conclusion that it is not only possible, but also very quick and easy to both integrate the generated code and upgrade it when there are changes in the specifications. Thanks to the high modularity of the request validator, we simply need to regenerate the needed files and replace them in the system without major modifications. Moreover, the integration requires very little changes on the overall system, and it does not require knowledge about the system's details.

By integrating it into the internal testing tool, GTT, we could target the invalid requests promptly. The two issues we found were caused by different errors and located in different areas of the request. The first error occurred because the test cases have not been entirely updated according to the specifications. In fact, the implementation of `‘ResStar’` is not completely wrong, but follows the outdated specifications of the 4G Core Network. Without a careful inspection of every single attribute, it is not easy to notice this kind of problem.

For the second error, the extra mapping is added because of the `‘oneOf’` keyword, which introduces the concept of polymorphism in OpenAPI. Since this concept cannot be directly translated into Erlang code, the developers are free to represent it in their preferred ways. In this case, the chosen approach was to add a mapping when a new schema is introduced. That is why the property `‘nfTypeCond’` is mapped to the `‘NfTypeCond’` schema, as it is one of the possibilities for `‘subscrCond’`. However, this does not match the expected JSON specifications, since the value of `‘subscrCond’` becomes the composition of two objects rather than a single object, and it is inconsistent with the rest of the system. These issues are very difficult to avoid as there are no strict rules for decoding OpenAPI concepts, which makes manually writing code sensitive to inconsistencies and bugs.

These two errors represent why it is difficult to maintain such a large system. Updates are laborious to perform manually, and the code's consistency and correctness is not guaranteed when just using the specifications as a guideline for implementation. Since not all the specifications can be directly translated into the target language, some implementation is done according to the programmer's understanding of the specifications and the characteristics of the language. Although the implementation might pass the test, it may contain errors that are not easily noticeable. In a large project where multiple people contribute to the source code, these cases can create inconsistencies that lower the code's quality and maintainability in the long run.

These observations, combined with our results, display the potential of code generation for validation. Its strictness can assist in checking whether the implementation is built consistently and accordingly to the specifications. The issues can be discovered accurately and effortlessly, potentially reducing the time invested in searching bugs. So, the developers can shift the focus to fixing the issues, increasing the efficiency of maintaining the system. This approach does not only help with checking the system's code, but the tests as well, reducing the risk of having test cases returning false positive or false negative results.

6.3 Limitations

There are three main types of limitations in this thesis project: the specifications for the 5G Core are not fully completed and some of the files have incorrect definitions according to the OpenAPI 3 format; the generator cannot have full generation from the specifications and some modifications are based on this project's needs; the integration and effectiveness of the request validation is strongly dependent on the generated code's quality.

6.3.1 Incomplete specifications

The OpenAPI 3 specifications for 5G Core have not been updated to their final version, so there are issues in some files. The most frequent problem we came across during the project is the repeated definition of `‘operationId’` for distinct operations in the same file. According to the specifications, `‘operationId’` is an optional string property used to uniquely identify an operation. Therefore, repeated operation IDs invalidate the specification and prevent the generator from generating any code. The easiest way to work around this problem is to remove all the explicit `‘operationId’` definitions, as they will be composed automatically by combining the operation's path and method.

Not entirely correct specifications can be a big problem, as they might require manual changes before being used. These changes might introduce further errors into the specifications, resulting in incorrect code and potentially breaking the entire system. Nevertheless, we don't need to worry too much about it as they are under constant maintenance from 3GPP. The specifications used in this project for generation are from the 2018 December release, but since then 3GPP has already released the March and June versions. The June release has quite big improvements compared to the December release and solves the issue for the repeated `‘operationId’`.

6.3.2 Limited functionality of the generator

The OpenAPI Generator has been modified during this project to make it cover most of the use cases for the 5G Core specifications. However, it still has the following limitations:

- OpenAPI 3 allows to mark some properties as `‘readOnly’` and `‘writeOnly’`. The `‘readOnly’` properties should only appear in the responses but not in the requests, while the `‘writeOnly’` properties should be included in the requests but not in the responses. However, the Erlang generated code cannot distinguish between the two cases as it uses the Jesse schema validator, which checks only the structure of schemas unaware of whether it is used as a request or as a response body. This became problematic when running the request validator for NRF in GTT: request with required properties that had `‘readOnly’` were seen as invalid because they did not have those properties, even though they were supposed to have them.

- Our fix for the ENUM problem is not completely correct. Through a discussion with one of the maintainers of the project we determined that our modifications create a new property every time the same ENUM schema is referenced rather than creating it once and referencing it. This is inconsistent with the idea of referenced schemas used to implement the generator. Furthermore, even though it passes all the generator’s tests, it has been reported to affect negatively the generated code for some cases for other languages. The current Erlang implementation cannot function without this change because of the Jesse dependency and its need to have parsed inputs. The alternative would be to use models for validation, but they lack support for a lot of advanced features (composed schemas in particular) which would result in worse validation overall than the one currently available.
- Normally, it is just preferred to define an `‘operationID’` for each callback. However, that becomes necessary if there are multiple operations with the same HTTP method defined under the same callback key, as that would result in duplicated operation IDs. For example, if two callbacks with expressions `‘request.query.callbackEvent1’` and `‘request.query.callbackEvent2’` are defined under the same callback key `‘onEvent’`, and they both have a GET operation, they will produce the same operation ID `‘OnEventGet’`. This happens because the expression used as callback path is not considered for the generation of `‘operationID’`.
- The generator has very limited support for the composed schemas as well as their nested version. The client’s generator cannot generate equivalent code for composed schemas at all when their properties are defined inline. In addition, generating code from specifications with nested composed schemas (i.e. a `‘oneOf’` defined inside of an `‘anyOf’`) throws a `‘NullPointerException’` in most cases and stops the generation. We had to write a workaround to make it work for this project.
- In some specific cases, the generator used by the generator cannot resolve some external schema references. The problem appears when a referenced request body references a schema defined in another file: the parser will not be able to extract the schema definition from the external file; thus, it will maintain in the processed JSON the path to the external file, which will not work for validation. This problem is caused by the imported library `‘io.swagger.v3.parser.core.models.SwaggerParseResult’`. Since this is not part of the OpenAPI Generator project we cannot do anything other than manually modifying the JSON file, but we have reported the error to the developers.

6.3.3 Generated code's quality

As we pointed out in the discussion, our results showcase the power of using the generated code as a request validator, since we can ensure a better check for the requests in the system easily. On the other hand, the strong dependency between the quality of validation and the generated code might raise a problem.

Before proceeding with the integration, it is necessary to make sure that the Erlang code is correct. If the generated validator returns false positives or false negatives, it can mislead the programmer into "fixing" something that already works correctly. It also necessary to verify whether the code covers most of the use cases needed by the system. If the generated code requires too many manual modifications to fix issues or add features, that would remove the purpose of code generation.

The goal of getting fully correct Erlang code from the generator is not easy to achieve. Erlang, not being as popular as other languages, is subject to very little maintenance. This makes the generator more likely to have bugs and not support the newest added features, as fewer people test and improve it. In fact, the Erlang templates have been almost unchanged since their last Swagger Codegen commit on December 2017. They would need an even more extensive rework than the ones that we have performed to be at par of other supported languages.

7

Conclusion

In this paper we found and analyzed a suitable tool for generating Erlang from the OpenAPI 3 specifications of the 5G Core, OpenAPI Generator, with the purpose of generating components to integrate into the Ericsson's 5G Core implementation. Even though the tool can generate successfully most of the features described in the specifications, it is not mature enough to provide full support.

We started the project by focusing on fixing various bugs and adding the features needed by the 5G Core to the generator to produce correct and runnable code. By analyzing the structure and the needs of the internal system in Ericsson, we considered the HTTP request validator as the best candidate for integrating the generated code. Some additional changes were needed to make the generated code more flexible for integration and to add some additional features. Ultimately, the integration required surprisingly minor changes to the system, and its high modularity made it easy to adapt to the system upgrade.

The validator successfully identified some bugs in the test cases of GTT, the internal test tool, which we used as another way to evaluate the quality of the generated code. By running the tests, we could confirm whether the requests were processed correctly or, when the tests failed, if it was a false positive or not. This validation can not only be applied to this specific system in Ericsson, but also to any other existing HTTP based systems due to the high modularity of the validator. Our case of using the generated code is also just a simple example of all the possible use cases. There are many other ways to take advantage of the power of code generation from OpenAPI 3. The next step related to our project would be validating the HTTP responses instead of just the requests. Moreover, we can even try to generate a complete component of 5GC, but in this case further studies on the generator are needed to make sure it is fully capable of supporting a business-level project.

It is worth mentioning that we did not measure the performance of the integration. There are a lot of variables affecting the execution time of the tests which make it impossible to have consistent and precise measurements. By reasoning on the current code, we believe that adding the request validation might bring a considerable delay for the handlers as, for complex components, some of the generated functions might have hundreds of cases of pattern matching. An analysis of the impact on the system's performance and how to make it more efficient would be a good future research to work on.

7.1 Observations

By studying how to integrate those components we learned more deeply about code generation and templating in general. We got to learn about the templating language Mustache, allowing us to write templates based on the system's needs. Through the project, OpenAPI 3 was not only very good for designing complex systems such as the 5G Core, but also a reliable input for code generation due to its strict structure. However, its application is limited to RESTful-based systems.

The available generators are still too immature to produce a full system: some advanced features are not supported at all, while others depend on the target language's support. That is why it becomes important to analyze whether the needed features are supported. These features depend on the system, where certain features might be essential for some while superfluous for others. For instance, in Erlang the `'oneOf'` attribute is validated correctly by the server, yet the generator can not generate functioning code for it in the client. Since the focus of our project was on validation at the server-side, this unsupported case did not prevent us from using code generation. However, this limitation could make it unusable for other projects.

Finally, the successful integration of the validation on the server might raise a question: why integrating only the validation instead of using the generated code as the basis for the implementation of the entire server? The reason why we are not doing this is not just because we already have a functioning server, but more importantly because the current state of the generator does not provide enough support for generating a fully functioning server/client.

Even if the generated code covered all the specifications correctly, it would still be impossible to generate the entire system automatically. Since OpenAPI 3 describes RESTful APIs, the handling of the requests in the server always needs to be implemented manually. Also, for more complex applications such as the 5G Core, the generated server would still require major optimization (such as parallelization) since it uses a very simple and generic structure which covers only base functionalities. In this project, we instead showed how that same code can be used to improve an already existing system.

Bibliography

[1] 3GPP. About 3gpp. https://www.3gpp.org/about-3gpp.

[2] Jari Arkko. Service-based architecture in 5g. https://www.ericsson.com/en/blog/2017/9/service-based-architecture-in-5g.

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

[4] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml™) version 1.1. *yaml. org, Tech. Rep*, page 23, 2005.

[5] GABRIEL Brown. Service-based architecture for 5g core networks. *A Heavy Reading white paper produced for Huawei Technologies Co. Ltd.*, 1:2018, 2017. https://www.huawei.com/en/press-events/news/2017/11/HeavyReading-WhitePaper-5G-Core-Network.

[6] Benoit Chesneau. hackney - http client library in erlang. https://github.com/benoitc/hackney.

[7] GitHub contributors. Openapi generator. http://openapi-generator.tech/openapi-generator/.

[8] GitHub contributors. Openapi generator's options. https://github.com/OpenAPITools/openapi-generator#3---usage.

[9] GitHub contributors. Openapi tools. https://openapi.tools/.

[10] Wiktionary contributors. Camelcase. https://en.wiktionary.org/wiki/CamelCase#English.

[11] Marcin Dryjanski. 5g core network functions. https://www.grandmetric.com/2018/03/02/5g-core-network-functions/.

[12] Maria Düsing. Your quick guide to network functions in 5g core. https://www.ericsson.com/en/blog/2019/2/your-quick-guide-to-network-functions-in-5g-core.

[13] GSNFV ETSI. Network functions virtualisation (nfv); use cases. *V1*, 1:2013–10, 2013. https://www.etsi.org/deliver/etsi_gs/NFV/001_099/001/01.01.01_60/gs_NFV001v010101p.pdf.

[14] EventHelix. 5g service-based architecture (sba). https://medium.com/5g-nr/5g-service-based-architecture-sba-47900b0ded0a.

[15] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. Technical report, The World Wide Web Consortium, 1999. https://www.rfc-editor.org/rfc/pdfrfc/rfc2616.txt.pdf.

[16] GitHub contributors For-GET. Jesse: a json schema validator. https://github.com/for-GET/jesse.

[17] GitHub contributors For-GET. Nine nines. https://github.com/ninenines/cowboy.

[18] go swagger. Golang implementation of swagger 2.0. https://github.com/go-swagger/go-swagger.

[19] GSMA. Road to 5g: Introduction and migration. https://www.gsma.com/futurenetworks/wp-content/uploads/2018/04/Road-to-5G-Introduction-and-Migration_FINAL.pdf.

[20] OpenAPI Initiative. Openapi callback object. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#callbackObject.

[21] OpenAPI Initiative. Openapi components object. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#componentsObject.

[22] OpenAPI Initiative. Openapi info object. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#infoObject.

[23] OpenAPI Initiative. Openapi media type object. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#mediaTypeObject.

[24] OpenAPI Initiative. Openapi operation object. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#operationObject.

[25] OpenAPI Initiative. Openapi parameter object. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#parameterObject.

[26] OpenAPI Initiative. Openapi path item object. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#pathItemObject.

[27] OpenAPI Initiative. Openapi paths object. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#callbackObject.

[28] OpenAPI Initiative. Openapi root objects. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#oasObject.

[29] OpenAPI Initiative. Openapi security requirement object. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#securityRequirementObject.

[30] OpenAPI Initiative. Openapi server object. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#serverObject.

[31] OpenAPI Initiative. Openapi specification. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md.

[32] ECMA International. The json data interchange syntax. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[33] Frank Mademann. The 5g system architecture. *Journal of ICT Standardization*, 6(1):77–86, 2018. https://doi.org/10.13052/jicts2245-800X.615.

[34] Georg Mayer. Restful apis for the 5g service based architecture. *Journal of ICT Standardization*, 6(1):101–116, 2018. https://doi.org/10.13052/jicts2245-800X.617.

[35] Metaswitch. What is the 5g service-based architecture (sba)? https://www.metaswitch.com/knowledge-center/reference/what-is-the-5g-service-based-architecture-sba.

[36] Oracle. Lesson: Packaging programs in jar files. https://docs.oracle.com/javase/tutorial/deployment/jar/index.html.

[37] Charles Rich and Richard C. Waters. Automatic programming: Myths and prospects. *Computer*, 21(8):40–51, 1988.

[38] Alex Rodriguez. Restful web services: The basics. *IBM developerWorks*, 33:18, 2008. http://www.gregbulla.com/TechStuff/Docs/ws-restful-pdf.pdf.

[39] SDxCentral. What is a virtual network function or vnf? https://www.sdxcentral.com/networking/nfv/definitions/virtual-network-function/.

[40] SmartBear. Openapi 3 - oneof, anyof, allof, not. https://swagger.io/docs/specification/data-models/oneof-anyof-allof-not/.

[41] Swagger and SmartBear. Swagger codegen. https://swagger.io/tools/swagger-codegen/.

[42] Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. Systematic

mapping study of template-based code generation. *Computer Languages, Systems & Structures*, 52:43 – 62, 2018.

[43] Techopedia. Backbone network.
`https://www.techopedia.com/definition/3158/backbone`.

[44] Techopedia. Core network.
`https://www.techopedia.com/definition/6641/core-network`.

[45] Chris Wanstrath. Mustache: logic-less templates, 2009.
`https://mustache.github.io/`.