Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

API Diversity for Microservices in the Domain of Connected Vehicles

Fabian Gajek

Course of Study:	Informatik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Dr. Johannes Wettinger, Michael Würster, M.Sc.
Commenced:	November 3, 2017
Completed:	May 3, 2018

Abstract

Web services in the domain of connected vehicles are subject to various requirements including high availability and large workloads. Microservices are an architectural style which can fulfill those requirements by fostering the independence and decoupling of software components as reusable services. To achieve this independence, microservices have to implement all aspects of providing the services themselves, including different API technologies for heterogeneous consumers and supporting features like authentication. In this work, we examine the use of a service proxy that externalizes these concerns into a sidecar that provides multiple APIs and common service functionality in a platform-independent manner. We look at how different kinds of API styles and technologies solve selected classes of problems and how we can translate between API technologies. We design and implement a framework for building gateways that enables the creation and composition of reusable components, in the fashion of Lego bricks, to maximize flexibility, while reducing the effort for building gateway components. We design and implement selected components of common and reusable API functionality enabling us to build a reference setup with a service proxy as a sidecar using our framework. Finally, we evaluate the proposed solution to identify benefits and drawbacks of the approach of using our framework as a service proxy. We conclude that the examined approach provides benefits for the development of many polyglot microservices, but splitting one service into two components adds additional complexity that has to be managed.

Kurzfassung

Web Services für vernetzte Fahrzeuge unterliegen unterschiedlichen Anforderungen, unter anderem einer hohen Verfügbarkeit und einem großen Datendurchsatz. Microservices sind ein Architekturstil, der diesen Anforderungen gerecht werden kann, indem er die Unabhängigkeit und Entkopplung von Softwarekomponenten als wiederverwendbare Services fördert. Zum Erreichen der Unabhängigkeit implementieren Microservices alle Aspekte der Servicebereitstellung eigenständig. Dazu gehört verschiedene API Technologien für heterogene Clients bereitzustellen und unterstützende Funktionalität wie Authentifizierung zu implementieren. In dieser Arbeit wird die Verwendung einer Proxy Komponente vor einem Service untersucht, durch welche die Bereitstellung verschiedener API Technologien und allgemeiner unterstützender Funktionalität aus dem Service extrahiert wird. Die Lösungen verschiedener API Technologien und Stile für ausgewählte Klassen an Problemen werden verglichen und mögliche Umwandlungen der verschiedenen API Technologien werden untersucht. Es wird ein Framework konzeptioniert und implementiert, das die Erstellung von Gateways durch Kombination von wiederverwendbaren Komponenten, wie das Zusammensetzen von Legosteinen, ermöglicht. Dieses Framework sorgt für eine hohe Flexibilität, während es den Aufwand bei der Erstellung von Gateways gering hält. Es werden ausgewählte wiederverwendbare Komponenten entworfen um eine Referenzimplementierung des Ansatzes umzusetzen, bei der allgemeine Funktionalität in einen parallel laufenden Proxy ausgelagert wird. Dieser Ansatz wird evaluiert, indem Vor- und Nachteile anhand eines mit dem Framework erstellten Proxys identifiziert werden. Das Fazit dieser Arbeit ist, dass dieser Ansatz bei Systemen mit vielen Microservices mit unterschiedlichen Programmiersprachen Vorteile bringt, aber die Trennung eines Services in zwei Komponenten eine nicht unerhebliche Komplexität einführt.

Contents

1 Introduction **15**
1.1 Thesis Structure 15
1.2 Terminology 16

2 Fundamentals and Related Work **17**
2.1 Internet of Things (IoT) and Connected Vehicles 17
2.2 Application Programming Interfaces (APIs) 17
2.3 Representational State Transfer (REST) 18
2.4 gRPC 19
2.5 GraphQL 22
2.6 Messaging 23
2.7 Service Computing 24
2.8 Enterprise Service Bus 25
2.9 Microservices 25
2.10 Gateways, Proxies and Service Meshes 27
2.11 Deployment Automation 28
2.12 Authentication and Authorization 29
2.13 API Bricks 32

3 Kinds of API Operations **33**
3.1 Request-Response 33
3.2 Asynchronous and Long-Running Requests 34
3.3 Event distribution 35
3.4 Stream or List of Results 37
3.5 Large Unstructured Data 38
3.6 Mapping and Transformation Approaches 39

4 API Translation **41**
4.1 Protocol Mapping 41
4.2 Protocol Modeling 42
4.3 API Remodeling 44

5 Framework for API Translation: Gateframe **47**
5.1 Framework Architecture 47
5.2 Execution Model 51
5.3 Plugin Concept 54
5.4 Deployment Approaches 57
5.5 Implementation 58

6 Framework Plugins for Common API Functionality: API Bricks **63**
6.1 REST and OpenAPI Adapter 63
6.2 REST and OpenAPI Connector 65
6.3 GraphQL Adapter 66
6.4 Interface Filter Intermediary 67
6.5 Authentication Intermediary 70
6.6 Authorization Intermediary 72
6.7 API-Level Monitoring Intermediary 72
6.8 Circuit Breaker Intermediary 73

7 Evaluation **75**
7.1 Preliminary Work 75
7.2 Gateframe as Service Proxy: API Translation and Common Service Functionality 80

8 Conclusion and Future Work **85**

Bibliography **89**

List of Figures

2.1 Messaging concept 23
2.2 Abstract OAuth 2.0 flow 31
2.3 Abstract OpenID Connect flow 31

3.1 Request-Response 33
3.2 Mapping messaging to task resources 36

4.1 Protocol mapping structure 42
4.2 Protocol modeling structure 43
4.3 API remodeling structure 45

5.1 gRPC as intermediary protocol 49
5.2 Gateframe instanziation 50
5.3 Call processing of the Gateframe 52
5.4 Lazy message transformation 53
5.5 Technical deployment options 57

6.1 HTTP to Protocol Buffers request mapping 64
6.2 Concept of the GraphQL adapter 66
6.3 Example of filter optimization preparation step 70

7.1 Data model of the gRPC example service 75
7.2 API talk test setup 76
7.3 Evaluation setup 81

List of Tables

4.1 REST API Azure Service Bus 43
4.2 Example REST API Remodeling of Messaging API 45

7.1 Measured data from HTTP performance evaluation 81
7.2 Measured data from gRPC performance evaluation 82
7.3 Aggregated data from performance evaluation 82

List of Listings

2.1 Proto example 21
5.1 Example of Googles HTTP rule [Goo17c] in option of method 55
5.2 Example of Googles HTTP rule [Goo17c] in gPRC API Configuration YAML . . 55
5.3 Example Gateframe Configuration 60
5.4 Request Parameters Interface 61
6.1 Example GraphQL schema generated from Protocol Buffers definition 68

List of Abbreviations

AMQP Advanced Message Queuing Protocol.

API Application Programming Interface.

ESB Enterprise Service Bus.

HTTP Hypertext Transfer Protocol.

IoT Internet of Things.

JSON JavaScript Object Notation.

JWK JSON Web Key.

JWT JSON Web Token.

PaaS Platform as a Service.

REST Representational State Transfer.

RPC Remote Procedure Call.

SOA Service Oriented Architecture.

URL Uniform Resource Locator.

1 Introduction

Web services that operate in the domain of connected vehicles are exposed to enormous data throughput requirements. Requirements greatly vary between emergency services and services for entertainment. Microservices aim to provide a solution to the increasingly complex and diverse requirements for large-scale web applications and are capable of tackling the challenges of connected vehicles. In a microservice architecture, many services are developed and operated independently. Each service must provide different Application Programming Interfaces (APIs) technologies and supporting features like security or monitoring themselves. Extracting API translation and common functionality from the service implementation reduces the development overhead for each service. The idea of API translation is to provide a single API that is automatically translated to other API technologies or styles. Libraries can provide such functionality for a specific platform or technology stack, but do not foster polyglot services which should be a benefit for using microservices. Out of process solutions that act as network proxy can provide standard functionality for services running on heterogeneous platforms.

In this work, we evaluate the approach of using reusable and composable components (called API bricks) to build these proxies. We do this by designing and developing a framework for building gateway using API bricks. We show the benefits of this approach compared to implementing the same functionality itself in the service and evaluate the impact of additional overhead in development and operation.

We examine the following approaches:

- Providing multiple API technologies by using generic translation components.
- Providing common and reusable API functionality in an API technology agnostic manner.
- Providing the above two using an out of process gateway in front of the service to be platform independent.

1.1 Thesis Structure

This work is structured in the following way:

In Chapter 2 we briefly introduce concepts and technologies that we reference and use and we look at similar problems. After that, we start taking a close look at the topic of API translation. In Chapter 3 we compare how types API requests are solved using different technologies. In Chapter 4 we define and discuss three types of API translation. In Chapter 5 we introduce the framework for building gateways that we designed and implemented in this work. The framework relies on reusable components that provide the actual functionality, called API bricks. In Chapter 6 we introduce some relevant examples of these API bricks. Finally in Chapter 7 we evaluate our approach and the

framework by comparing two setups of a service providing the same functionality with and without our method of using a gateway. In Chapter 8 we summarize the finding of this work and propose some additional work that could be done using the concept of API bricks and the framework for building gateways.

1.2 Terminology

As we use some terms that are not universally accepted and are something used with different meanings, we define how we use some terms in the context of this work:

Upstream The communication direction that is sending requests to a service. The direction that originates at the requestor and ends at the service.

Downstream The communication direction that is sending a response from a service. The direction that originates at the service and ends at the requestor.

Common service functionality or common API functionality Functionality that has to be implemented by most services, like authentication, monitoring or circuit breaking. Through the distributed nature of microservices, a central system like the Enterprise Service Bus (ESB) cannot provide this functionality.

API Brick A component of reusable functionality for APIs. These bricks provide customization through flexible configuration and a uniform interface to put minimal constraints on the combination options.

Application or Service We use this terms often interchangeably, as we mostly talk about applications that provide their functionality as a service over the network.

Host We a host as a machine that executes processes in a distributed system. We abstract from the notation of physical devices, such that even two containers sharing the operating system can be on different hosts. We define that two processes are on the same host if the can reach each other using the 'localhost' name.

Service proxy A gateway or proxy in front of an application that provides supporting functionality.

2 Fundamentals and Related Work

This chapter introduces concepts and technologies used in this work, as well as work that we built up upon or that focuses on similar problems.

2.1 Internet of Things (IoT) and Connected Vehicles

The term Internet of Things (IoT) describes the integration of sensing, actuating, and connective technology into daily items [GBMP13]. Through the connectivity, these items extend the edge of the Internet to even tiny devices like weather or air sensors. These devices produce sensor values in large quantities which often the devices cannot process themselves, because of the reduced resources on the device or the missing context needed for the intended computations. Instead, the small devices transfer the data to other machines that have the necessary capabilities. A scalable infrastructure like the cloud provides the storage and processing capabilities necessary to consume the enormous amount of data produced by these devices [TD15].

Connected vehicles are part of the transportation application of IoT. The connectivity for vehicles consists of communication within the vehicle, with other vehicles or with the Internet [LCZ+14]. We will focus on the last. The integration of connectivity into cars enables new applications like driving assistance, entertainment, or safety. Emergency systems like the eCall system of the European union [Eur15], that automatically alerts emergency services, show the possibilities for road safety that connected vehicles provide. The safety-related services need to have a high degree of availability. Assistance, or entertainment related service, e.g., live traffic information or music streaming, often are data intensive. Thus these services need reliable and scalable supporting applications that typically are deployed in cloud environments.

2.2 Application Programming Interfaces (APIs)

The term Application Programming Interface (API) describes the mechanism how an application or application component allows another program to use its functionality [Mul18]. The method of communication between the component providing an API and the component using it can take various forms, e.g, local procedure calls for using a library, interrupts for calling the kernel of the operating system, or network communication to use the functionality of a remote server. We only look at the last kind of API of an application providing functionality over the network. This type is also called web API, but we will use the term API synonymously for web API. Application Programming Interfaces often provide a detailed description of their interface. This description can contain elements like available methods and data structures for parameters and results. This interface definition is intended for the easier consumptions by applications in contrast to a human client of the application, who is more robust against unknown or varying interfaces.

2.3 Representational State Transfer (REST)

Roy Fielding introduced the architectural style Representational State Transfer (REST) in his dissertation [Fie00]. It is the architectural style of the world wide web and widely used for building web services using Hypertext Transfer Protocol (HTTP). While REST is an architectural style and not a technology or bound to a specific technology, it is often used synonymously for its use over HTTP [FR14b]. In our context, we only look at REST over HTTP.

First of all, REST follows the client-server communication paradigm style and uses synchronous communication that means the only way of interaction is a client sending a request to a server and the server sending a response immediately. This interaction is stateless in REST, meaning the history of interactions between client and server is not taken into account by the server when responding to the client. The clients send all information the server in the request in a self-contained manner. This is the reason for 'State Transfer' in the name. Fielding argues that statelessness induces visibility, reliability, and scalability. Visibility is achieved through the direct correlation of a request and its response. Reliability is achieved as it allows easier recovery from failures, i.e., it is easier to retry a single request than a complete interaction. The scalability is induced by the statelessness allowing a request to be processed by different servers.

The REST style requires responses to be cachable. The server marks the replied data implicitly or explicitly as cacheable. For example, in HTTP we achieve this by using headers like the 'Cache-Control' header, which directly controls the behavior of caches.

An essential aspect of the REST style is the uniform interface. The uniform interface allows the creation of components that can be independent of the functionality of an application, e.g., proxies. The following four elements make up the uniform interface of REST [Fie00]:

Identification of Resources The targets of generic interactions are resources that can be uniquely identified. For HTTP this is done by Uniform Resource Locators (URLs) [BFM05]. Resources can be documents like HTML or images, but also other abstractions like collections of resources or services can be resources.

Manipulation through Representation A client manipulates resources by sending completed or partial representations of them. E.g., using the HTTP verb 'PUT' and a complete representation of the resources lets a client update the resource to the state described by the representation.

Self-Descriptive Messages The messages contain all information needed to process them. Requests include all necessary data for a server to create a response and no information must be extracted from the context, e.g., from the history of interaction, etc. For responses that means a cache can complete further requests from seeing a response.

Hypermedia as the Engine of Application State (HATEOAS) The basic idea of HATEOAS is that the server can provide information on the available operations of resources by providing them using references. HATEOAS relieves the client of knowing URIs of resources (other than entry points) and fosters the loose coupling between client and server.

The REST style uses a layered system, interactions between clients and servers can go through multiple components that can understand the communication. These components are not low-level network components but can understand the semantics of the executed operation. This can, for example, be a cache that can return a response in substitution of the server due to the visibility of the interactions or an authorization proxy that only relays requests with correct credentials.

REST also has an optional code-on-demand constrain, that allows the server to augment the behavior of the client.

The maturity model of Leonard Richardson breaks down how elements of HTTP are used for RESTful web services [Fow10]. It defines levels of increasing adaption of REST in HTTP.

Level 0: Swamp of Pox HTTP is used only as a method of transportation. Requests typically are made only to a single URL using 'POST' as the verb. Only the bodys of the interaction contain its meaning.

Level 1: Resources Additionally, we add visibility of resources through the use of URLs for different resources. For example, we assign different methods of an Remote Procedure Call (RPC) style service to different URLs.

Level 2: HTTP Verbs Now we add generic and visible interactions to our resources. By using HTTP verbs like GET, PUT, or DELETE we can clearly communicate whether we want to read, update, or delete a resource.

Level 3: HATEOAS To decouple the client of a REST endpoint from the structure of the API, we provide links to declare possible operations. To achieve this services often provide options to list resources. The list of resources then not only contains some information, but also references to URLs where operations like reading or deleting of a specific resource can be executed.

2.3.1 OpenAPI

OpenAPI is a standard aimed at describing REST APIs over HTTP [Ope18a]. The goal of OpenAPI is to provide a service definition that can be understood by humans and machines. Thus for the latter allowing code generation for clients and servers.

The standard provides a specific contract for REST APIs. This eases the use of specialized clients that need to know exactly what to expect from the API. This is in contrast to general clients, like a web browser, which allows very generic interactions with an HTTP based service.

2.4 gRPC

gRPC is a Remote Procedure Call (RPC) framework that originated from Google's Stubby [The18] and reflects Google's understanding of how an API protocol for connecting microservices should look. It is based on HTTP/2 [BPT15] and thus is compatible with HTTP/2 network components and utilizes the benefits of the protocol. An RPC framework allows the calling of another function on a remote machine, with classical implementations often trying to make these calls appear as local calls [BN84]. These lead to problems with the handling of network errors or increase the difficulty

of calling procedures on other platforms [WWWK96]. gRPC addresses issues of traditional RPC approaches, e.g., gRPC is platform independent, and there exist implementations for most common languages. Also, the nature of the call being remote is not hidden from a caller of a method.

The interface of a provided service is described in proto files [Goo17f]. These files are part of Protocol Buffers, a serialization format that gRPC uses primarily for transferring requests and responses. Both server and client code can be generated from interface definitions. gRPC methods only take a single complex type as request and response. This also differs from local calls that often use an ordered list for multiple parameters, which results in a tighter coupling. gRPC uses metadata to provide additional information about API calls. In gRPC, a client can send metadata before the first request. The server can send metadata before the first response and together with the status code after the last response. This differs from HTTP/1.1 where the server can only send status code and headers before sending data in the body [FR14b].

2.4.1 Request types

gRPC defines the following four types of request-response interactions for calling a method:

Unary The client sends a single request, and the server returns a single response.

Client Streaming The client sends multiple requests, and the server returns a single response after the processing of all requests from the client.

Server Streaming The client sends a single request and the server replies with multiple responses.

Bidirectional Streaming The client sends multiple requests and the server replies with multiple responses. The number of requests and responses do not have to correlate, and requests and responses can be sent interleaving.

2.4.2 Protocol Buffers

Protocol Buffers [Goo17f] are a platform-agnostic binary serialization format that is used by gRPC as the default wire format. gRPC can also use other wire formats like JavaScript Object Notation (JSON) [Ecm17].

The primary goal of Protocol Buffers is to create a simple, fast, and efficient data serialization format. As the format is binary, a description is needed for serialization and deserialization. These descriptions are defined in so-called proto files that are also used for defining the gRPC services (services are collections of methods). Listing 2.1 shows an example of such a proto file. The service in the example consists of four methods with each being of different type regarding streaming.

The most recent version 3 of Protocol Buffers uses generic default values for each type; That means the standard defines the default values and we can not redefine them in the proto file, as was the case in version 2. For example, the type int32 has a default value of 0. If a value is not available in the serialization the default value is returned. A serializer can use this behavior to omit fields that are set to their default values.

Listing 2.1 Example proto file describing two message types and a gRPC service.

```
syntax = "proto3";

package foo;

message Request {
  string param_1 = 1;
  int32 param_2 = 2;
}

message Response {
  bytes data = 1;
}

service ExampleService {
  rpc Unary(Request) returns (Response);
  rpc ClientStreaming(stream Request) returns (Response);
  rpc ServiceStreaming(Request) returns (stream Response);
  rpc BidirectionalStreaming(stream Request) returns (stream Response);
}
```

While Protocol Buffers are just a serialization format, there also exist some predefined types. Some of these types are referred to as 'well-known types' because they are very generally usable and implement best practices using Protocol Buffers. For example, there exist wrappers for primitive types [Inc16] that are complex objects containing only a single field called value. These wrappers can be used to differentiate between the absence of a value and a default value.

2.4.3 JSON/HTTP mapping

Protocol Buffers support conversion to a JSON representation. Protocol Buffers define a mapping from and to JSON objects [Goo17f]. As both are structurally very similar, this mapping is simple. The names defined in the proto files are used as keys (the lower camel case version for JSON) for the mapping, the textual representation is used for enum values, etc.

Well-known types sometimes define an individual mapping to a JSON representation. For example, the wrappers for primitive types [Inc16] essentially only represent a value of a primitive type, and in JSON the differentiation between an absent value, and the default value can be made without wrapping the value in an object; thus they are mapped to just the equivalent of the primitive value they represent.

This mapping does not only allow JSON as a wire format for gRPC, but it also allows to call gRPC APIs using JSON over HTTP [Goo18c]. The default mapping is an HTTP API that uses the fully qualifying method names in the URL, POST for all methods, and the request is created from the body. But it is possible to use options in proto files to annotate the use of other verbs, URLs, and map request values from path and query. The configuration of the HTTP mapping allows the creation of APIs in the style of REST. The mapping annotations look for example like the following:

```
rpc Unary(Request) returns (Response) {
  option (google.api.http).get = "/v1/unary/{param_2}";
}
```

In this example if we assume the type 'Request' originates from Listing 2.1 then the object for the request is created by getting the value of 'param_2' from the path and the value of 'param_1' from a query parameter with the same name.

2.5 GraphQL

GraphQL [Fac16] is a query language for client-server applications, which allows clients to specify their data requirement in a declarative way. A typical use for GraphQL is enabling clients to retrieve all the data needed to display information to a user in a single request-response interaction. To achieve this, GraphQL provides the ability to specify the exact information-need to the server.

GraphQL aims to solve two problems of REST APIs, that for associated resources often multiple requests have to be made and that responses often contain information that is not needed by the client. Changing the granularity of a REST API (i.e., by splitting or merging resources) to improve one problem, worsens the other.

2.5.1 Queries

A GraphQL query allows the exact definition of the information a server should send to a client. The following example from the specification [Fac16], could retrieve the name of the currently logged in user:

```
{
  user(id: 4) {
    name
  }
}
```

The server would return the exact information needed, for example, with the following JSON:

```
{
  "user": {
    "name": "Fabian"
  }
}
```

GraphQL defines two types of requests 'queries' and 'mutations.' The above example is a query. Mutations use the same syntax as queries (except for the mutation keyword). The only distinction is the state changing server implementation. GraphQL only allows declaring the information returned by the server to the client for a mutation.

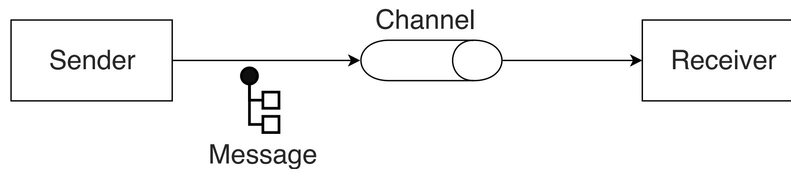

Figure 2.1: This figure illustrates the basic concept of messaging. Sender and receiver of messages are decoupled by a channel. The sender sends messages only to the channel and the receiver retrieves the message from the channel.

2.5.2 Type System

GraphQL defines a type system that describes the possible queries that are allowed. Two basic types are scalars and objects, amongst others. We can see a GraphQL query as a tree. Scalar types, e.g. string or integer, are used for leaf values like 'name' in the above example. Object values represent intermediary nodes that can contain other objects or scalar values.

The two special root types 'query' and 'mutation' are used to indicate entry points for queries.

The type system itself is a graph that can contain loops. A graph with loops allows queries of arbitrary depth (should be restricted by the server) for simple schemas like the following:

```
1 type User {
2         name: String
3         friends
4 }
5 type Query {
6         me: User
7 }
```

For this schema the following query could retrieve the name of all friends of friends of the currently logged in user:

```
1 {
2   me {
3     friends {
4       friends {
5         name
6       }
7     }
8   }
9 }
```

2.6 Messaging

Messaging is an asynchronous communication pattern that looses coupling between sender and receiver of a message [HW04]. A fundamental concept in messaging are channels as depicted in Figure 2.1. Channels are logical network elements that are provided by a messaging system. Messages are sent to and received from channels. There exist two types of channels, the first being point-to-point (queue) and the other publish-subscribe (topic). With the first channel, a message is

received by exactly one receiver, in the latter, a message is received by zero or multiple receivers which have declared their interest in the message. Messaging is asynchronous because the receiver of a message does not even have to be available at the time the message is sent, it can retrieve the message from a channel at a later time. Sender and receiver do not need to know each other as they only communicate over the channel. As the channel a sender is sending a message to does not need to be the final destination, a message can be moved from channel to channel and be processed on the way to allow transformation or routing. All these properties allow messaging to be a very loosely coupled communication technology. This loose coupling is in contrast to synchronous RPC, especially classical implementations. The benefits of asynchronous communication over RPCs is that the sender does not need to know the receiver; it does not have to wait for a response; the sent message does not need to have exactly the format that the receiver expects; and sender and receiver do not have to share the same platform, which is often the case for RPC implementations. Modern RPC implementations increase some of these properties, but it remains that the caller has to open a network connection to the receiver that has to be kept open for the duration of the method call.

In messaging the logic for mediation of the messages are kept centralized in a component that is called a message broker [HW04]. The message broker receives messages from multiple sources, applies routing and transformation logic, and then send the messages to their destinations. The message broker bears the risk of integrating many unstructured routing rules leading to an unclear system. This complexity can be managed by reducing the centralization through the use of a hierarchy of brokers.

2.6.1 AMQP

The Advanced Message Queuing Protocol (AMQP) [AMQ11] is a standardized protocol to enable the interoperability between messaging systems of different vendors. We use AMQP as an example for a messaging protocol and technology.

2.7 Service Computing

Service computing is a distributed computing paradigm [Erl08], in which applications provide and use general and reusable functionality called services. Services are provided over the network using APIs in different styles and technologies like REST over HTTP, gRPC, or messaging. Service Oriented Architecture (SOA) it the architectural style used in service computing. The terms are often used with varying meaning depending on the context [Erl08]. In an SOA, service providers publish their services to a registry. The service requestor can find the provided service in the registry and then bind to it to use the specific service. This relation between the three elements service provider, service requestor, and service registry is called 'SOA triangle' [SHLP05]. Another essential element of service computing are services that coordinate other services to provide new functionality, which is called 'orchestration.'

2.8 Enterprise Service Bus

The Enterprise Service Bus (ESB) is a message based integration platform often based on web services technology [Men07]. The ESB provides implementations for message routing and translation, invocation of services, security, and many more supporting features. The ESB provides enhanced capabilities for the binding part of an SOA [SHLP05]. Through the mediation of messages, it allows service requestors and service providers to talk to each other, without the service provider exactly matching the interface and requirements of the requestor.

In the ESB service requestors and service provider both register metadata about the services they use and provide [SHLP05]. The ESB implements functionality to connect requestors and providers very dynamically, even if the requested and provided service are not entirely compatible. To achieve this, the ESB includes processing of messages on the wire, called 'mediation.' The possibilities for mediation range from simple monitoring, validation, to routing and enrichment of messages. Many of these patterns originate from Hohpes Enterprise Integration Patterns [HW04]. Schmidt et al. [SHLP05] present the following patterns amongst others that can typically be found in ESB:

The 'monitor' pattern is a mediator that does not change the messages of interaction but only reads them. This allows the inspection of production service invocations and interactions to provide insight into the state of services or to collect business-relevant metrics.

The 'transcoder' pattern is a mediator that changes the format of messages, e.g., it converts a Protocol Buffers message to a message in JSON format[1]. It does not change the meaning of the elements in the message but only changes the technical representation.

The 'modifier' pattern changes the content of a message. A typical application is a transformation. Other than the transcoder pattern, does not alter the representation but the structure of the elements within the message. This is relevant if requestor and provider use different structures (i.e., different schemas) to represent the same information.

Mediators are similar to API bricks used in this work. The differences to mediators in ESB also depend on how they are deployed and managed. In this work, we look at deploying them together with the service or as a service, where they lie in the responsibility of the service and not like with the ESB at a network location independent from the service.

2.9 Microservices

Microservices are a relatively new topic in service computing. Microservices are the consequence of new trends and technologies, like DevOps or infrastructure automation [New15]. Microservices are small services build around clearly cut business capabilities. The complete life-cycle of microservices are kept as independent as possible from each other. The microservice architecture can be seen as a particular kind of SOA, but it has its own name as it uses different concepts for some aspects [FL14]. Microservices are often compared to monolithic applications, where most elements run in a single process.

[1] We chose this example as it uses formats we introduced in this work.

One characterization of microservices from Fowler is ’smart endpoints and dumb pipes’ [FL14]. Products like the ESB provide sophisticated network capabilities like routing, message transformation, or even service orchestration. In service orchestration, the ESB calls multiple services in a process to create a new business function. Microservices only use ’dumb pipes,’ meaning network communication that only provides basic networking features like HTTP or simple messaging, meaning messaging systems that do not provide any additional features other than providing channels. The orchestration of other services is not done in network components, i.e., the ESB, but only in services themselves that use simple communication, e.g., HTTP, to call other services.

An essential aspect, that summaries aspects from Fowler [FL14], is to maximize decentralization. This means decentralizing governance, allowing each team and service to decide on the technology of their choice. This means decentralizing the data, each service manages their data, and can use appropriate database technology for their data. Microservices use virtualized and automated infrastructure to allow easy deployment for fast introduction of new features and elastic scaling. Microservices are designed for failure, as failures happen in distributed systems. Microservices also maximize the degree of distribution. To keep the complete system from crashing due to the failure of a subsystem, microservices should be robust in case of a failure of their dependencies.

In the following, we introduce some microservice patterns that are relevant to this work.

2.9.1 API Gateway

An API gateway provides a technically uniform interface of a microservices system to the outside [Ric17]. It resides at the edge of the network and hides the distribution and volatility of the service from the clients. If services provide APIs over technologies that are not usable for the web or services offer various API technologies the gateway can translate the APIs to ensure a uniform API that is suitable for the web.

A variation of the API gateway is Backends for Frontends [Ric17]. The idea is to provide an API gateway for each different client (or group of similar clients). This allows us to optimize the API gateway for the needs of a specific client. For example, Netflix lets its frontends teams also develop a service that provides the API for their clients [Bry17; Xia17]. These services are developed in Node.js [Nod18b], as many of their clients are also written in JavaScript.

2.9.2 Sidecar

In microservices common service functionality like monitoring or logging has to be implemented by each service. To reduce the implementation overhead and improve the consistency of multiple services this functionality can be provided by libraries, but libraries can typically only be used by a single platform. The sidecar pattern solves the problem of providing common service functionality in a platform independent manner [Mic17]. The common service functionality resides in a separate process that is deployed alongside the service, similar to a sidecar on a motorcycle. The sidecar can use its own platform and is independent of the platform of the service. As the service communicates with the sidecar using interprocess communication, e.g., through sockets, this pattern can introduce non-negligible performance overhead.

2.9.3 Service Mesh

The idea of the service mesh is to decouple the service from the network [Ind17; Mor17]. Instead of a service calling different remote services, it calls them locally, i.e., using 'localhost' as the host. A local process, a sidecar, routes the network calls to their destination. The service mesh can also provide additional network resiliency like retries or circuit breakers. This decouples the service from the environment it is running in and reduces the need for configuration in the service itself. A service mesh hides the complexity of the network and provides the network services as if they were locally available.

2.10 Gateways, Proxies and Service Meshes

In this section, we list a number of relevant proxy implementations that were designed with web services or microservices in mind. The difference of our approach to all solutions is the flexibility and adaptability. We conceptualize our approach to be able to add new functionality through plugins easily. Thus the proxies in this section are not aimed to be used in the Backends for Frontends pattern [Ric17].

2.10.1 Extensible Service Proxy

The extensible service proxy is developed by Google and is used on the Google Cloud platform with Cloud Endpoints [Goo18a]. The extensible service proxy provides common service functionality for HTTP and gRPC based APIs. Technically the extensible service proxy is based on NGINX HTTP reverse proxy server using modules to add intermediary processing. Thus the general features provided by the extensible service proxy can be used any HTTP based endpoint. It provides service supporting features like token authentication, monitoring, logging, etc.

The support for gRPC contains the general features mentioned above, as well as JSON to gRPC transcoding [Goo17f]. The extensibility, platform flexibility, and performance originate from NGINX as the base platform. It allows the integration of additional modules and provides the interfaces and support for processing HTTP requests.

While the extensible service proxy provides JSON/HTTP to gRPC translation, it differs from our approach that is not indented to support API translation. Its primary focus point is the general support features for services. We do use a concept that sees more of the structure of the API. Specifically, the plugins in our approach know the structure of requests and responses and do not only work on the HTTP layer. Our approach can be flexibly deployed at different locations, while the extensible service proxy is intended to be used only directly before a service.

2.10.2 Envoy Proxy

The envoy proxy is a service and edge proxy for microservices and implements the service mesh pattern [Env18]. Its goal is to hide the network topology from the services. An envoy is deployed for each service locally, and all network communication is local from the view of the service. The

service itself does not need to differentiate between environments like development or production as it only sends a request to localhost. Envoy routes the request to the destination with features like service discovery, load balancing, circuit breaking, etc.

Envoy provides support for TCP connections and has additional support for HTTP/1.1, HTTP/2, and gRPC. Features are added through filters, that can use different interfaces depending on the connection type. As envoy can transparently proxy between HTTP/1.1 and HTTP/2 (in both directions), HTTP filters operate on a general interface that hides the underlying connection. The following gRPC protocol translations are provided using filters:

gRPC HTTP/1.1 Bridge Enables invoking unary gRPC methods through an HTTP/1.1 interface using Protocol Buffers as the body.

gRPC-JSON Transcoder Implementation of the specified gRPC to HTTP/JSON mapping using the HTTP rule annotation in the proto definition.

gRPC-Web Translation of gRPC to gRPC-Web that allows the usage of gRPC from a web browser.

The difference from a gateway view to our approach is that envoy acts as a proxy for incoming and outgoing request calls. Our approach is intended to be used only for incoming requests calls. The envoy proxy operates on the network and connection layer as well as the application layer (HTTP, gRPC). Our approach works on the application layer of gRPC. Generally, our approach intends to be a flexible framework and not a finished solution that has a particular use case in mind. For example, with our framework, a developer can build a gateway that aggregates different services to hide the service distribution of the upstream services.

2.10.3 grpc-gateway

The grpc-gateway project is a plugin for the Protocol Buffers compiler that generates golang code for an HTTP/JSON API that is translated from gRPC [Aut]. The gateway uses the HTTP rule annotations for the structure of the API. Additional to the generation of the gateway code the project also provides a generator for Swagger 2.0 (OpenAPI 2.0) definition. It is intended to be deployed in front of a single gRPC service to provide a REST API for it.

The grpc-gateway provides very similar functionality to the REST adapter of our implementation. Our adapter generates an OpenAPI definition for the REST API, as well. Bur our adapter does not work on the network gRPC communication, but on the internal representation of our framework. It integrates into our framework for building gateways, which can provide additional functionality.

2.11 Deployment Automation

As microservices aim to easily and quickly integrate new changes, as well as react to changing workloads, we should keep the deployment simple by using automation [FL14]. We introduce the concepts of container-based virtualization and platform as a service.

2.11.1 Container Virtualization

Virtualization, in general, is the abstraction of resources, using logical components [Wal07]. For example, a virtual machine can run on different hardware, as it only sees an abstraction of the actual hardware. Container-based virtualization is an operating system level virtualization [Wal07]. The container does not see details of the operations system it run on and multiple container running on the same system cannot see each other. As the container share the kernel of the host operating system, this kind of virtualization can use resources more efficiently than hardware virtualization.

Docker [Doc18] is a prominent technology for container virtualization. Docker provides an easy interface on top of existing container virtualization of operating systems. The developer defines a file called a Dockerfile that accompanies the application and described the steps necessary to create a container image from the application. The image allows instantiation of a container that then runs on any docker enabled host.

2.11.2 Platform as a Service

Platform as a Service (PaaS) is a pattern of providing a platform on which developers can easily run their applications as long as the applications only use languages and dependencies (libraries, middleware, etc.) that are supported by the platform [FLR14; MG11]. The developer has no direct control of the underlying infrastructure (servers, network, operating system, etc.), but can influence some aspects of the deployment through a configuration interface. Prominent examples are Heroku [Sal18], Google App Engine [Goo18b], or Elastic Beanstalk from Amazon Web Services [Ama18].

The Cloud Foundry [Clo18b] platform is an implementation of the PaaS pattern. It can be deployed on many infrastructure solutions. Thus it is often used in private cloud deployments. Cloud Foundry uses buildpacks to support different languages. Buildpacks can automatically detect the used language for an application to deploy at build time. The buildpack then packages all necessary dependencies into a so-called droplet. The droplet then can be executed on the platform, very similar to docker images. Cloud Foundry also provides middleware like databases or messaging as services that can be used by applications.

2.12 Authentication and Authorization

In the context of services, authentication means the identification of the entity (principal) that calls a service. Authorization is the determination if the principal is allowed to use the service or is permitted to execute a specific operation. Authentication of a principal can help to determine authorization, e.g., a service can resolve whether the subject is the owner of a business object. Authentication is not necessarily a requirement for authorization.

Monolithic applications often have all elements of authentication and authorization in the same container as the application. They manage users and rights themselves. For microservices this approach is also possible, but often not applicable. In microservice authentication and authorization

information is often needed by a service, but a dedicated service manages the identity information and not the service itself. Thus we need options to share this security information between services, also meaning a service should be able to send its authentication context to an upstream service.

2.12.1 Pre-Authenticated Requests

One possible way we can relay this security information is by using the assumption that all requests to services come from a local network and are therefore trustworthy. A gateway authenticates a requestor against the identity provider and then relays the request augmented with security information to the service. We call this request with cleartext security information pre-authenticated requests.

We can improve the protection against man-in-the-middle attacks by using only encrypted connections (i.e., HTTPS). But this does not prevent an attacker from within the network from sending pre-authenticated requests with manipulated security information.

2.12.2 Token Based Authentication

Very generally speaking tokens are just strings that allow authentication of the requestor. Tokens can be passed along the call chain, while an attacker can not manipulate or create these tokens.

We will present OpenID Connect [SBJ+14] as the example of token-based authentication. Based on OAuth 2.0 [D H12] and other web standards, it is a specification for allowing an identity service [2] to authenticate the end-user against clients. These clients can be, amongst others, web-based services, mobile-, or web-apps. As OpenID Connect was designed for the web, it fosters the independence of the individual elements and therefore making it suitable for usage in microservices. OpenID is based on different web standards with various configurations. This allows us only to use the setup that is suitable for our specific system. e.g., just using JSON Web Token (JWT) for passing security information between services.

OAuth 2.0 [D H12] defines an authorization framework to enable a client to act on behalf of the resource owner (i.e., the user). The user allows the client to retrieve an access token from an authorization service, as depicted in Figure 2.2. The client can then use this token with another service to access resources accessibly to the resource owner. Depending on the flow, the client and the resource service do not have access to the authentication credentials of the resource owner. OAuth does not define a from of the access token. E.g., it can be an opaque token that the resource service can use to request identity and authorization from the identity service.

The OpenID Connect [SBJ+14] uses the OAuth 2.0 flow to provide authentication on top of it. OpenID specifies additional details that allow the authorization service to relay information about the user to the client. OpenID uses Internet standards like JSON Web Token (JWT), JSON Web Signature (JWS), and JSON Web Encryption (JWE) to enable this. Figure 2.3 depicts the general flow of OpenID Connect. The authentication response contains an ID token, a JWT, which provides the relying party with verifiable claims about the user. This information can be anything, and OpenID Connect specifies general user information like the name. The JWT can also contain other

[2] We use the term service in this context instead of the term server, that is used in the OAuth 2.0 specification.

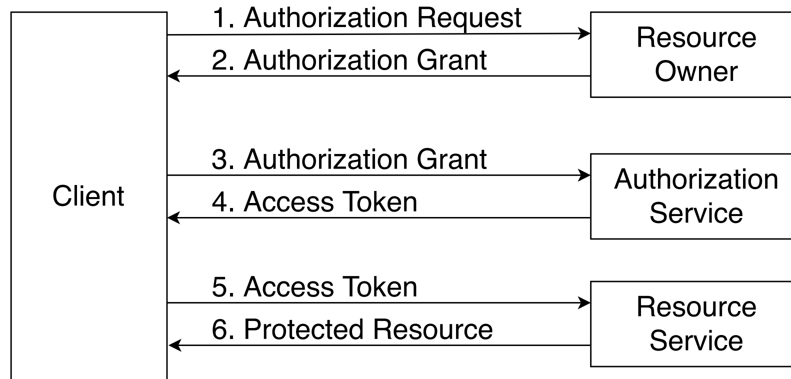

Figure 2.2: The abstract OAuth 2.0 flow as described in section 1.2 of the RFC 6749 [D H12]. The client request access from the resource owner (i.e., the user), e.g., by requesting credentials or by redirecting to the authorization service. After authorization is granted, the client receives an access token that it can use to access secured resources.

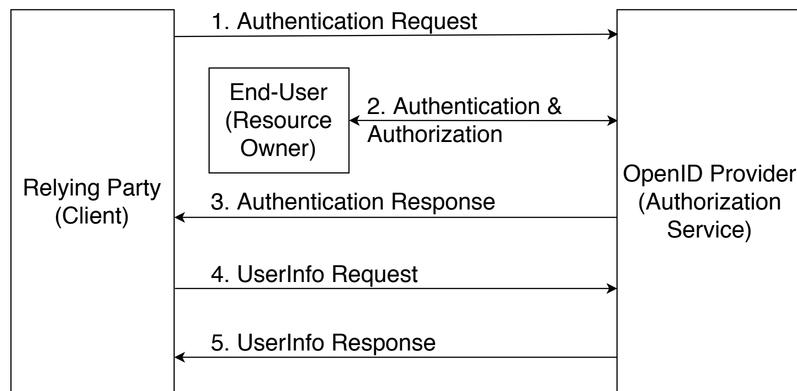

Figure 2.3: The abstract OpenID Connect flow as described in section 1.3 of the OpenID Connect Core specification [SBJ+14]. The terms in brackets are the terms used in OAuth 2.0. The relying party request authentication of the user from the OpenId provider. The provider consolidates with user and responses with the authentication information. If the relying party needs additional information, it can request it using the access token returned in 3.

(custom) claims that allow various use cases. We use OpenID Connect as an example how JWT can be created and be used for authentication and authorization, but JWT can be used independently of OpenID Connect.

JWTs are the underlying token technology that allows the relying party to relay the information returned from the OpenID Provider to a resource service without the resource service needing to trust the relying party. JWT is a format for exchanging claims [JBS15], in the form of a JSON object. A JWT can ensure integrity by using JSON Web Signature (JWS) for signing the claims. Another party can then use the public key of the signer to verify that no attacker manipulated the claims. This integrity mechanism allows a service to authenticate a user without directly contacting

the authorization service while being sure that the authorization service provided the claims. A JWT can also be encrypted using JSON Web Encryption (JWE) to implement confidentiality of the claims. This allows scenarios were the token is transmitted over parties that are not trusted.

2.13 API Bricks

The idea of reusable and compensable components for building APIs are described Wettinger in the specification of Any2API [Wet17]. He calls the individual components API bricks, and we will use this terminology in this work. The idea behind the API bricks is to ease the development of API endpoints, especially those that contain relatively much functionality compared to the business logic of the service. Under the premise that specific functionality can be used for many services API bricks reduce the need to reimplement functionality by wrapping it in components that can be added and combined like Lego bricks. To be able to connect them in a very flexible order, the API bricks use universal interfaces. The API bricks need to implement their functionality in a general and configurable manner to allows their distribution over a repository from which developer then can retrieve them for their requirements.

Speth describes his CLARA framework that implements the API bricks concept [Spe17]. The framework uses similar concepts as the specification of Any2API. The implementation of Any2API differs from our approach in the aspect of using out of process communication (gRPC) between all components, while in our approach the API bricks are in the same process.

3 Kinds of API Operations

This chapter presents classes of API operations and characteristics of API requests. These classes are neither distinct nor do they cover all cases. The goal of this chapter is to show solutions in different technologies and styles for the problems request-response, asynchronous requests, event distribution, streaming multiple results, and sending large unstructured data. The styles and technologies discussed are REST/HTTP, gRPC, GraphlQL, and Messaging. We only add dedicated examples for these technologies if they fundamentally differ from another. We also provide some mapping ideas from one style to another.

3.1 Request-Response

Request-Response is the API operation of sending a single request to the server, which then sends a single response back to the requestor (see Hohpes Request-Reply [HW04]). These two messages have a strict correlation.

3.1.1 Client-Server (HTTP, gRPC, GraphQL)

Request-Response is the primary communication style of protocols that follow the client-server (or RPC) style. The requestor sends a request over a connection-based communication protocol (typically TCP) to later correlate the response to the request. Synchronous implementations even block the calling thread until the response arrives.

3.1.2 Messaging

Messaging is one-way communication, but Hohpes Request-Reply pattern describes a way to implement two-way communication on top of that [HW04]. The requestor sends a message to a request channel and receives the response on a response channel. Often the pattern called 'Correlation Identifier' is used to associate the response to the correct request context.

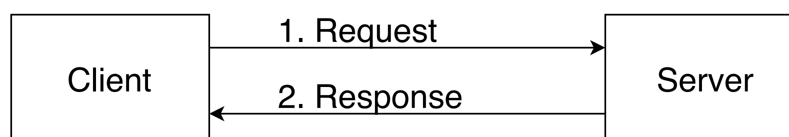

Figure 3.1: Structure of Request-Response. The clients send a request, and the server sends a single response in return that correlates to the request.

3.1.3 Mapping

If a service provides a client-server style request-response the mapping to a messaging interface is straightforward. The message consumer retrieves the message, calls the client-server upstream and after completion sends a message to the response channel. Some endpoint patterns like 'Transactional Client' [HW04] can be implemented to increase the resilience of the endpoint.

The other way around needs a translator to implement the client side of the Request-Reply pattern. The translator holds the connection to the requestor and waits (synchronously or asynchronously) for the message on the response channel. It is critical that the response message is eventually consumed by the instance holding the connection of the requestor. It is also important to mention that a requestor can cancel a connection and that there might not be a waiting consumer (i.e., translator component) for a response message anymore.

3.2 Asynchronous and Long-Running Requests

We do not differentiate between asynchronous and long-running requests. In our context, long-running requests are executed asynchronously and there exists no bound on the duration of an asynchronous request. We think the differentiation is not needed, as the solutions can be restricted to be called synchronously and have an upper execution time bound. In general we look at requests for which the time of availability of the results is unbounded or not in the near future (depending on the context and technology).

3.2.1 Task Resources

In REST/HTTP the clients expect an immediate response. Task resource help to decouple a long-running execution from the HTTP requests. The general procedure is the following [All10]:

1. The clients send a 'POST' request to the server.
2. The server returns a task resource with 202 (Accepted) and a URI in the 'Content-Location' header where the task can be retrieved.
3. The client uses URI to poll the status of the task and repeats until the task is not processed anymore (successful completed or failed).
4. If the task was successfully executed the 'GET' request on the task resource returns a 303 (See Other), with the 'Location' header set to URI where to retrieve the result.

This approach can also, with slight modifications, be used by GraphQL and gRPC. For gRPC Google has already defined message and service descriptions called operations [Goo17d].

3.2.2 gRPC: Keep Connection Open with Long Deadline

For this case, we differentiate between long-running requests that take a few minutes to complete and extremely long running requests that take multiple hours, days or even have no bound on the completion time. gRPC provides the possibility to wait for the completion of a request of the first kind with an upper bound on the completion time in the range of a few minutes.

As gRPC uses HTTP/2 streams [Aut18b] and an asynchronous execution model, it can in some cases apply to wait for a response to a long-running task. As often this is not the case, gRPC uses deadlines indicating the urgency of a response [The18]. Deadlines are usually in the range of a few seconds, mainly when an end user client issues the requests. So this option is only reasonable in a service-to-service invocation without end-user interaction, and the caller has to set a very long or possibly infinite deadline.

3.2.3 REST/HTTP: Callback

The general approach of task resources in REST/HTTP has the problems that come with periodic polling (sending a request periodically to check whether a state has changed). If the server can access the client, the server can issue a request to a resource URI of the client when the task is completed [Sma18].

3.2.4 Messaging

Asynchrony is built into messaging. So generally in messaging, there is no particular consideration needed when making long-running or asynchronous requests. It can be harder to retrieve and inspect the status of an operation in an asynchronous messaging system, but it depends on the implementation.

3.2.5 Mapping

There exist numerous options for implementing requests with a delayed response. The mapping from a message based API to task resources is common, as it is used for providing a RESTful API on the web, that internally uses messaging. The challenge with mapping task resources is the possible introduction of state into the component implementing the translating service. The example of mapping messaging to REST/HTTP in Figure 3.2 demonstrates that.

3.3 Event distribution

Clients or in general applications may be interested in information about some events. It cannot request the information from a service, as the time of creation of such events is not known beforehand. In microservices, a system that can distribute events is often used to decouple the services and reduce the need for orchestration [New15]. The solutions for events can intersect with asynchronous communication, as the technical solutions solve very similar problems. The significant difference

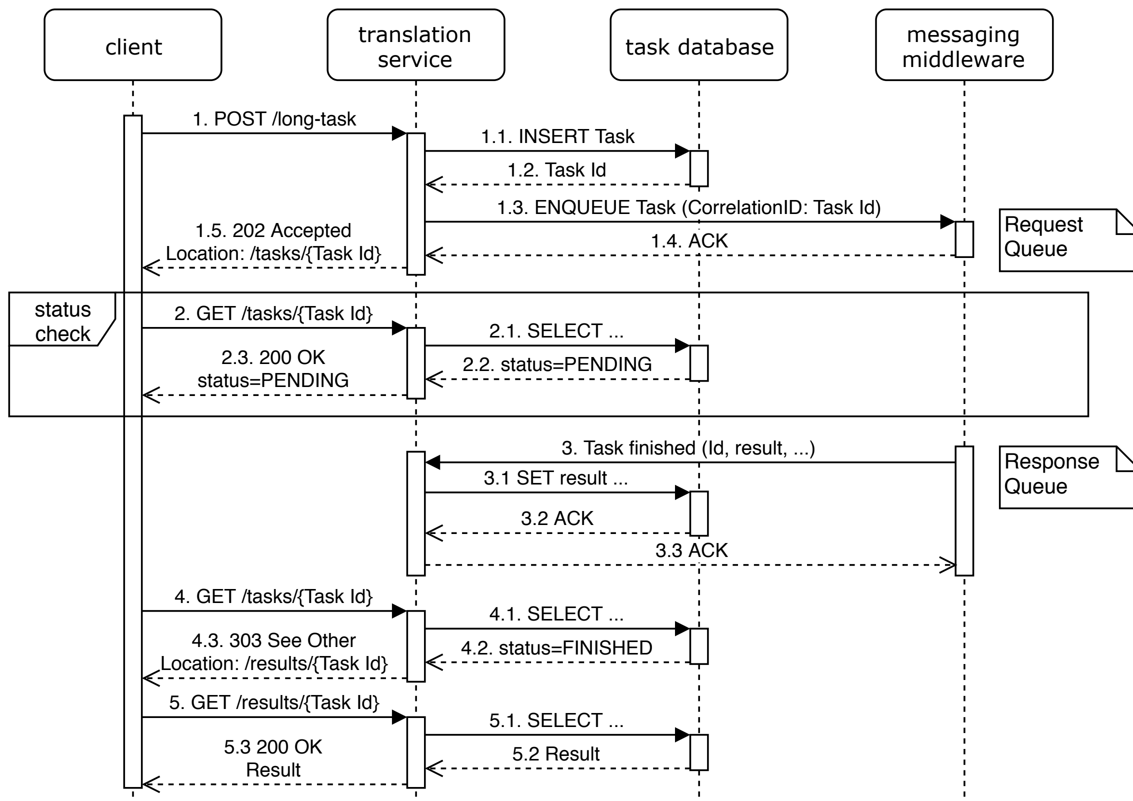

Figure 3.2: Sequence diagram for an exemplary mapping from REST/HTTP to messaging. We use REST/HTTP task resources to wrap a messaging API. The sending of a message is coupled with an entry in a database. The return of a correlated message updates this entry. All status requests and the final result are satisfied using the local database of the translation service.

to Section 3.2 is that multiple messages can be sent. Often for event distribution clients must be provided with the functionality to subscribe to events and to remove the subscriptions later when no longer needed.

3.3.1 Callback

Callbacks were described in Section 3.2.3 for long-running tasks. They can also be used to retrieve an arbitrary number of responses. The client can send a callback URI to a subscription endpoint [Sma18]. The server sends all relevant events to the callback URI until the client unsubscribes with the server. This concept is also called webhooks.

3.3.2 HTTP: Long Polling

Callbacks improve on polling but are not applicable if the server cannot send an HTTP request to the client. Long polling is a workaround that uses HTTP to enable the server to push data to the client [Han14]: The clients send a request that the server does not immediately respond to, but

keeps open. When an event occurs, the server can send it to the client as the response to the pending request. After receiving the response, the client immediately sends another request, such that the server can respond to an unfinished request at all times.

3.3.3 REST/HTTP: Atom

The Atom Syndication Format together with the Atom Publishing Protocol is meant for sharing a list of resources called feeds [All10; New15]. It was initially intended for distributing displayable elements of web pages, like block posts. But it can be used for the delivery of arbitrary information: A producer of events publishes the event to a feed. A client that is interested in the event regularly polls the feed and checks for changes.

3.3.4 GraphQL: Subscriptions

Subscriptions are a specification of GraphQL that allow clients to specify queries like in the normal request-response style of GraphQL [Fac17]. These queries are then executed on the server every time a new event arrives from the base event stream, and the server pushes the resulting objects to the client. GraphQL does not define transport protocols. Thus GraphQL subscriptions can be used over any protocol that allows pushing of data (e.g., web sockets or HTTP/2).

3.3.5 Messaging: Point-to-Point and Publish-Subscribe

Messaging (especially publish-subscribe) is very suitable for event distribution [HW04]. If the event should be consumed by exactly one consumer, a point-to-point channel or queue should be used. Publish-subscribe enables the event distribution to multiple consumers.

3.4 Stream or List of Results

A standard API request is the listing of a collection. We look at the class of operations that have multiple results. The number of results is usually unknown, large, and possibly unbounded. Often it is not feasible or necessary to return all results of the request at once. Sometimes only a small number of elements of the result are enough to satisfy the need that resulted from the request, e.g., the first page of a search query is enough to find what you were searching.

3.4.1 Pagination

If the communication pattern is (synchronous) request-response, it is only possible to transmit a small number of elements for one request. Pagination splits the result of a request into pages of the same size, and a request returns one page. Often the requestor can specify the size of the page and the page index to retrieve [Par17]. Pagination is used in REST and GraphQL, but also in gRPC if only a small result is needed at a time.

3.4.2 gRPC: Streaming Response

In gRPC, we can return a stream of responses to deliver a large number of elements. This allows for the transmission of many elements, even without knowing about the number of elements beforehand. This can be useful when aggregating the results of upstream services. Stream responses that are unbounded in time and number of elements can only be used if the consumer is able to hold the connection and can consume large amounts of data. This means service-to-service communication in background tasks.

3.4.3 Mapping

The mapping from a paginated resource collection (e.g., accessible through REST/HTTP) to a streaming response is straightforward. An adapter iterates through the pages and writes the elements of each page to the stream. Through the nature of the stream, the adapter does neither need to keep any state nor needs to block. The other way around cannot be implemented generically. The stream of responses does not necessarily have a defined order, but elements can arrive in the order they were computed. Even if a fixed mapping to pages is possible, retrieving the full stream might not be an option, as it takes to long for a timely response, or the amount of data may be too large. For some services (but not applicable) it may be possible for the mapping component to store a copy of the stream result and regularly update it.

3.5 Large Unstructured Data

Some information like images or videos have no structure regarding the interface of an API but are just seen as a big amount of binary data. Binary data can be integrated into any payload of service interactions, i.e., through BASE64 in any string encoded field (this adds a significant data overhead of one third, as 6 bits are encoded using an 8-bit character). Often the technologies for small structured elements are not very suitable for large amounts of unstructured data. Thus requests with large binary data are handled differently. GraphQL is a transportation format that highly depends on the structure of information and has no direct support for binary data.

3.5.1 Unstructured Data in REST/HTTP

HTTP has first class support for binary data, as through content-negotiation a server can support arbitrary formats [All10].

3.5.2 gRPC: Chunking

gRPC is intended for small structured messages (gRPC for Java has a default max message size of 4MB) [Bra17]. We split a large amount of data into a stream of small messages each containing a chunk of the data. Each message is a structured message (Protocol Buffer) including a binary field

with the part of the data. The server splits the data into these messages, and the client reassembles them. Chunks are sent and retrieved in order, and the end of the stream can be used to indicate the end of data.

3.5.3 Messaging: Message Sequence

Messaging systems often contain a maximum message size. The message sequence pattern allows splitting a payload into a sequence of multiple messages to overcome this limitation [HW04]. Messages are not guaranteed to be retrieved in order, and the end of the sequence cannot be inferred from context. The pattern uses three fields to solve this:

Sequence Identifier Unique id for the message sequence, usually a correlation identifier is used with the request-reply pattern.

Position An index that identifies the chunks position for reassembly.

Size or End Indicator The end of the message sequence is indicated by a sequence size that is sent with every message or by using a flag on the last message. This allows detection of the completion of the sequence.

3.5.4 Mapping

For all implementations mentioned above, we can indicate the size of the data (in bytes) beforehand (it is possible with the protocol, not necessarily with the underlying system). If this information is kept consistent, then the mapping is possible with acceptable effort. I.e., problems arise if a translation component needs to know the size of the data (and it is not known), it has to buffer all data to send the size in a header. We can implement the mapping from HTTP to gRPC and the other way round by basically piping the binary data received to the outgoing connection. Splitting the stream into messages contains no challenge. To assemble a message sequence, the adapter may need to store chunks that are received out of order.

3.6 Mapping and Transformation Approaches

The list of possible solutions in this chapter for the same problems in APIs is not complete but shows different approaches through examples.

We think a common similarity of difficult approaches is those that introduce state into a component. In the sense of 'dumb pipes' these transformations should not be done in network components, but only in services themselves. The introduction of state means that often additionally the transformation component needs an external component to keep the state (i.e., a database). We think that these types of transformations should not be done in general network components and they, in general, should be avoided if possible. If they are needed then a transformation for this specific case should be developed in form of a service.

In the next chapter we describe different mapping categories and add another view on the problems of API translation.

4 API Translation

We define API translation as the conversion of one kind of API to another. The general problem can be described as making an API that is provided using protocol P reachable through protocol Q. In some cases, the structure of the API that is provided using Q may even be different than the structure when using the API with protocol P. This is relevant if Q is fundamentally different from P, e.g., REST/HTTP and messaging.

We think the need for API translation arises due to technical limitations and restrictions:

Legacy Protocols Gateways between legacy and new system need to translate old and new protocols.

Suitability for Environment The protocol may be suitable for communication between services in the same environment, but not for calls that originate from other environments. For example gRPC is not well suited for the web as it can not be directly called from a web browser.

Policies on Protocols Often enterprises only allow HTTP traffic into the network from the web [HW04].

Speciall Considerations for Clients API translation can provide interfaces that are adjusted to the needs of a particular client. For example network protocols can be optimized for mobile clients by using GraphQL to reduce the round trip times.

In this chapter, we define three protocol mapping types. Theses definitions provide a better understanding of how protocols can be translated on the wire and the problems that arise with those translations. In practice, these types are not necessarily distinct.

4.1 Protocol Mapping

We define protocol mapping as an API translation that only transforms the underlying protocol. The two protocols have to be very similar, meaning the protocols provide the same fundamental functions or the functions of one is included as a subset of the other. For example HTTP/2 supports all features of HTTP/1.1. Reverse proxies can translate the protocols if only a subset of HTTP/2 features are used.

4.1.1 Example: HTTP/2 to HTTP/1.1

With many microservices that are implemented using a REST API using HTTP/1.1, it would be tough to transition to HTTP/2 if every service had to provide HTTP/2 before the transition is possible. Instead, a proxy, e.g., NGINX [NGI15], at the edge can provide the client with an HTTP/2 API while sending HTTP/1.1 upstream requests to the individual services.

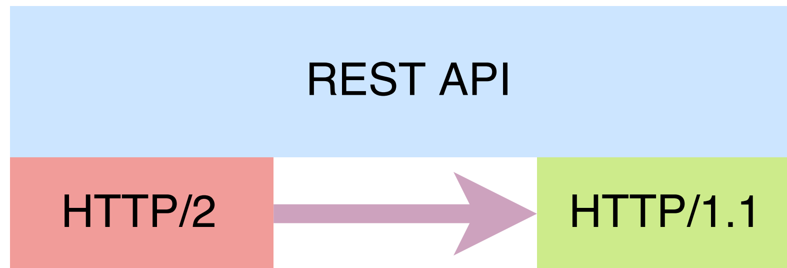

Figure 4.1: Request to an HTTP/2 endpoint can be transparently converted into upstream HTTP/1.1 requests.

4.1.2 Discussion

The benefits of this approach originate from the similar nature of the translated protocols. The translation is transparent to clients using the API. The payload of the messages is not changed. The original structure of the API is directly exposed. Thus the original design is not changed. The translator does not need information about the structure of the API. A translator does not need to retain state but converts and relays messages.

The approaches disadvantages lie in the differences in semantics and features of the mapped protocols. In most cases, the protocols do not support the same set of features, in some case one protocols features are a superset of the others. For example HTTP/2 is a superset of HTTP/1.1, as HTTP/2 adds support for server push [BPT15]. An adapter that transparently maps an HTTP/2 to HTTP/1.1 cannot support push and might restrict usage of features. Another problem arises from the semantics of fields used in protocols like the HTTP status code that is mapped to the gRPC status codes. As HTTP was designed with web resources and gRPC with APIs in mind, the codes have different granularity for some error cases. For example, the gRPC codes 'Invalid Argument' and 'Out of Range' are both equivalent to a 400 (Bad Request) HTTP status code [Aut18d; FR14a]. A possibility to improve on the unclear semantics is to weaken the transparency of the mapping. E.g., by adding information about the gRPC status code in the HTTP mapping.

The problem with this approach is the required similarity of the protocols. This leads to challenges of mapping semantics with similar protocols and makes the mapping of fundamentally different protocols impossible.

4.2 Protocol Modeling

The idea of protocol modeling is that we can see the protocol of a service API itself that resides on a lower level of abstraction. We can enable the communication with a service that uses protocol P through a protocol Q by modeling P so that we can send it on top of Q. We can visualize this as P being lifted to a higher communication layer or Q being downgraded to a transport protocol.

URI	Verb	Description
{...}/{queue-|topic-path}/messages	POST	Send message(s) to a queue or topic.
{...}/{queue-path}/messages/head {...}/{subscriptionName}/messages/head	DELETE	Destructive read message from a queue or a subscription of a topic.
	POST	Peek-Lock (non-destructive read) message from a queue or a subscription of a topic.
{...}/messages/{messageId}/{lockToken}	PUT	Remove the lock from message (Unlock), making it consumable again.
	DELETE	Remove the message from queue or subscription.
	POST	Renew the acquired lock on the message.

Table 4.1: Extract of operations of Azure Service Bus REST API showing endpoints for message creation and consumption [Mic15]. The API models elements of AMQP, like queues, topics, and messages as resources and enables CRUD-style operations on them (CRUD abbreviates create, read, update, and delete, the generic interaction on resources in REST).

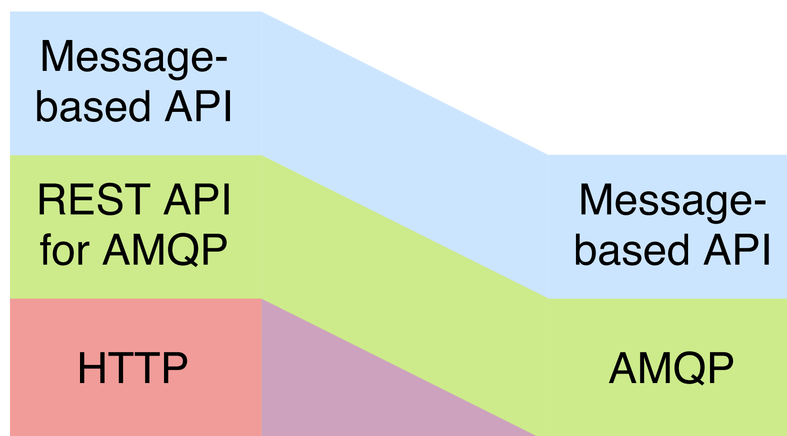

Figure 4.2: Protocol Modeling by the example of a REST adapter for AMQP. The adapter exposes the AMQP protocol to enable usage of the messaging API.

4.2.1 Example: REST API of Azure Service Bus

A good example of the idea behind protocol modeling is the REST API provided by Azure for its Service Bus [Mic18a]. The offering is a cloud-based messaging solution that provides point-to-point (queues) and publish-subscribe channels. It is compatible with Advanced Message Queuing Protocol (AMQP) [AMQ11]. Table 4.1 shows some operations that essentially model a REST API for AMQP. The API exposes the internal structure of the message bus.

4.2.2 Discussion

Protocol modelings main benefits lie in the universality of the approach. It applies to most protocols. An adapter directly exposes the protocol of a service and thus the API structure of the service. The adapter does not need any knowledge about the API on top of the protocol.

A disadvantage of the approach is the visibility of the mapped protocol to the client. We lose the benefits of API translation that decouples the provided from the consumed protocol. We also increase complexity by adding another level to the communication stack. Client developers need to understand two levels of communication, the transport protocol Q, and the modeled protocol P. The business logic API translated with this approach does not change its architectural style: If we make AMQP accessible over HTTP, we still have a messaging API.

The problem with protocol modeling resides in different interactivity attributes of protocols. For example streaming of gRPC allows an interactive bidirectional of client and server. While it is possible to model a streaming call using a REST API, it is hard to achieve the interactivity a server may expect from a client.

4.3 API Remodeling

The idea of API remodeling is to provide a translation of an API that only needs to keep the business semantics. This means the translator essentially repeats the process of modeling the API for another technology and style. The remodeling does not necessarily have to be using another technology. For example, we can remodel an RPC-style HTTP API as a REST API.

4.3.1 Example: REST task resource to messaging

We reuse the example from the previous chapter shown in Figure 3.2. In the example, an adapter provides access to a message based API through a REST style interface. The REST interface provides access to task resources that represent the asynchronous operations. On the creation of the task resource the adapter puts a message in a queue and asynchronously waits for the response message. The adapter saves the state and the result of tasks in a database. Table 4.2 shows how a resulting REST API could look like.

4.3.2 Discussion

API remodeling enables to hide all details of the original API while retaining the business logic of the API. As the translated API can be of any technology and style API remodeling allows the resulting API to precisely fit the needs of its intended use.

The problems with this approach are that it is hard to automate. A possible solution to the problem would be to map reoccurring patterns in APIs. This is still hard to automate as patterns not an exact way of structuring, but only guidelines that enable some interpretation. Also, the structure of the API has to be known to generate a translation.

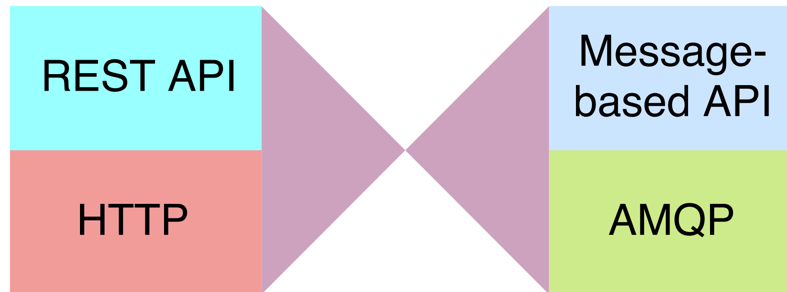

Figure 4.3: By complete remodeling of the API, a message based API can be transformed into a REST API. The APIs must only share the business logic, but not any technical structures.

URI	Verb	Description
/tasks	POST	Starts the execution of a long-running task. The translation service creates a task object in its database and starts the execution by putting a message into the task queue. Returns the task id in the body and the URI to retrieve the status in the 'Location' header.
/tasks/{task-id}	GET	Retrieves information about the status of the task. Possible values are RUNNING, FAILED, or SUCCESS. If the status is SUCCESS, a status code of 303 with the location of the result is returned.
	DELETE	Cancels the processing of the task. The task is removed from the database, and a message is put into a queue to start the compensation process.
/tasks/{task-id}/result	GET	Returns the result from the long-running task.
	DELETE	Removes the task and the results from the database to free up space.

Table 4.2: The table shows operations on a fictional REST API that is a remodeling of a message-based API. The endpoints are inspired by the translation service we introduced in Figure 3.2.

The main problem with this approach is the possible introduction of state into a translation component. As discussed in Chapter 3, the introduction of state increases the complexity of implementation and operation of translation components.

We think that API remodeling is correlated to the problematic cases of Chapter 3. Thus in microservices, this kind of API translation should not reside in network components (to conform with 'dumb pipes'). API remodeling should be (part of) a service that is owned by a development team to manage the complexity of the mapping.

5 Framework for API Translation: Gateframe

In this chapter, we introduce a framework for building gateways and service proxies that we call Gateframe (short for 'gateway framework'). In the last chapter, we introduced three types of API translation. We intend the framework to be unbiased, flexible and able to support all three types of API translations that we introduced. These properties mean the framework can be used in various scenarios where components process API requests and relay them to other components. An intended usage option is a deployment as a service proxy. API translation and common service tasks like authentication or monitoring are extracted into the service proxy to simplify service development while providing requestors with a rich API. Another intended use is the implementation of the Backends for Frontends pattern, where individual gateways are developed for different frontends to be more suited to their needs.

Approaches in microservices like service mesh provided functionality that was provided by systems like the Enterprise Service Bus (ESB) [Ind17]. This approach has the risk of introducing centralized management and bind the service to the technology. But microservices are intended to have decentralized governance and should be able to use different technologies and to change used technology stacks [FL14] easily. The Gateframe intends to reduce the work of the services to provide different API technologies and common service functionality. The Gateframe enables developers to efficiently create a service proxy that provides supporting features for the service. The flexibility of the Gateframe allows a developer to change behavior easily or even rewrite some part entirely to prevent binding the service to some central components.

Netflix implements the Backends for Frontends pattern in the following way [Bry17; Xia17]: They hide distribution of the microservices behind an edge API. Individual clients do not directly use this API, but there exists a gateway between each client and the edge API. These gateways are written in Node.js as the same development team owns the gateway which develops the client, and the client is typically written in JavaScript as well. The Gateframe is well suited to be used in this scenario. Our implementation also uses Node.js and provides a plugin system for functionality to be reused between different gateways. The Gateframe can improve the initial development time of these gateways, and by encouraging the use of generalizing code, it prevents the overambitious API gateway anti-pattern. The overambitious API gateway pattern describes the practice of introducing business logic into gateway components [Tho18].

5.1 Framework Architecture

This section describes concept and implementation of the Gateframe.

5.1.1 Intended Properties

The base architecture of the Gateframe is intended to fulfill the following requirements:

API Diversity through API Translation A primary focus of Gateframe is the support for API translation. We designed the Gateframe to support all three types of API translations that we introduced in Chapter 4.

Independece from API Technology The elements that provide functionality should work without modification with different API technologies. This is achieved by using a common platform-agnostic protocol. Thus the underlying used protocol is transparent to the API brick.

API Consistency We define API consistency as the usage of the same technologies and styles. This means in the best case everyone uses the same protocol and styles for their APIs. This is hard to achieve, especially with legacy systems directly. The Gateframe provided API translation allows the creation of proxy components that offer the service of another (legacy) service in the protocol and style of a subsystem.

Reusability The Gateframe uses API bricks (see Section 2.13) to provide common service functionality. Universally applicable solutions like API monitoring should be implemented once and be reused as much as possible.

Generality The Gateframe should foster the development of general functionality in API bricks. Bricks that solve service specific problems lead to overambitious gateways [Tho18]. These gateways, that contain business logic, lead to difficult to understand services.

Scalability The Gateframe is intended to be used in microservices and microservices are making use of automatic provisioning and horizontal scaling to adapt to varying workloads.

Great Developer Experience The Gateframe is not a finished gateway product that a developer only has to configure, but a framework for building a gateway. That means it is a component that is developed. To be able to provide a feasible alternative to of the shelve solutions, the development of a gateway with a the Gateframe has to be similarly simple as configuring some parameters.

Performance The Gateframe should avoid reducing performance and wasting resource whenever possible.

Extensibility The framework is intended to easily add new functionality.

Suitability for Microservice Environments While the Gateframe does not restrict the intended uses, its intended use is microservices. Meaning the Gateframe should be suited for environments that are decentralized and highly volatile. A consequence is, for example, the suitability for deploying the Gateframe on automated infrastructure.

Simplified Deployment The deployment of the Gateframe should be as simple as possible, as a goal of the Gateframe as a service proxy is to reduce the complexity of microservices. We should keep the additional deployment complexity that is introduced by the Gateframe as small as possible.

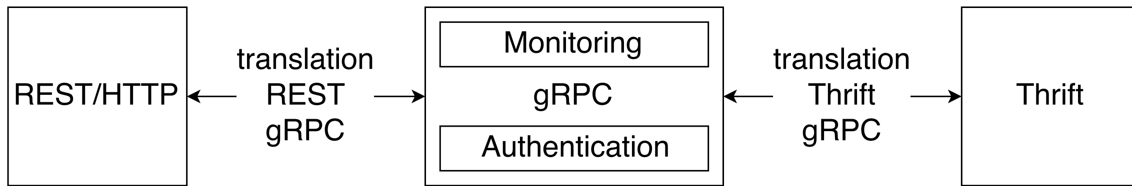

Figure 5.1: gRPC as an intermediary protocol: In the displayed exemplary mapping from REST to/from Thrift both protocols are only translated to gRPC. All common service operations like monitoring or authentication are executed on the common protocol and are independent of the mapped protocols.

5.1.2 Common Protocol Format: gRPC

We want to enable components (API bricks) that we define once and then reuse without modification for different protocols. To reach this goal, we can translate all protocols to and from a common protocol. We implement all common service functions for this general protocol. A common protocol format also optimizes API translation, as for each protocol we must only create a conversion to and from the common protocol format.

We use gRPC as a common protocol for the following reasons:

- gRPC is based on an openly available specification [Aut18b].
- gRPC has a required interface definition language [The18].
- gRPC is based on HTTP/2 [Aut18b], making it compatible with modern standard components like HTTP proxies.
- gRPC is suitable for microservices. It incorporates Google's experience in building microservices. Many companies and projects use gRPC for the service-to-service communication [The18].
- gRPC is platform independent, and there exists open source implementations for the most common languages [The18].

We did choose gRPC over REST/HTTP, mainly because it provides a standardized format with a clear and reduced interface definition language. The many alternatives in building REST APIs make it unfit for use as uniform protocol format. Also, the support for streaming makes gRPC more suitable as the intermediary format, to support more potential protocol mappings. Especially gRPC itself would be hard to map with REST/HTTP as an intermediary format.

5.1.3 API Bricks: Adapters, Connectors and Intermediaries

The framework defines the following structural components, that we subsume under the term bricks:

Adapter An adapter provides an API. The adapter translates API requests coming from a downstream requestor to the intermediary format.

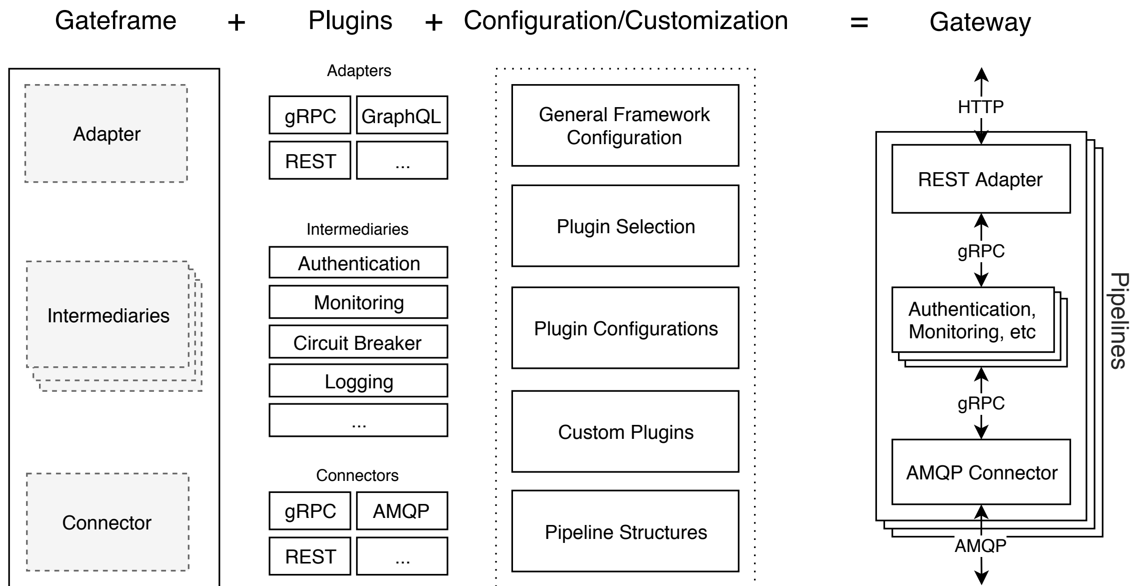

Figure 5.2: Concept of creating a gateway using the Gateframe. The Gateframe provides the general structure. Reusable API bricks exist in the form of plugins. A configuration can use the generally provided plugins or create custom plugins just for the specific deployment.

Connector A connector sends requests to an API of a service. The connector translates API requests from the common message format to the format of an upstream service.

Intermediary An intermediary provides common service functionality. An intermediary operates on the common protocol format to achieve arbitrary functions. An intermediary can change the API interface visible to downstream elements or handle requests.

API bricks provide a uniform interface that enables the flexible combination of the components. The term originates from the idea that those elements are like Lego bricks that can be stuck together in various combinations. The important property to achieve this flexibility is that the bricks provide a uniform interface. An intermediary brick implements two interfaces one in upstream direction and one in downstream direction (just like knobs and holes in Lego bricks). An adapter provides the same interface in upstream direction, while a connector provides only the downstream direction.

Adapters, connectors, and intermediaries form linear pipelines, as can be seen in Figure 5.2. The communication between these components is process-internal on a Gateframe internal representation of the gRPC protocol. The general structure is the following: The adapter transforms the downstream protocol into the internal gRPC representation. All intermediaries work on the internal representation of gRPC one after another in the configured order. Finally, a connector transforms from gRPC to the API protocol of the upstream service.

To enable extensibility, API bricks are provided as plugins for the Gateframe. The Gateframe itself only provides the interface definitions that the bricks have to implement and the functionality to configure and run the API bricks.

5.1.4 Configuration and Customization

The configuration of the Gateframe consists of the following parts:

General Framework Configuration General configuration of the framework, independent from the configuration pipelines. For example to configure the port of the information endpoint described in Section 5.3.2.

Plugin Selection We have to choose all plugins to integrate into the deployment of the Gateframe.

Plugin Configuration Configure and customize the behavior plugins to match the functionality needed. A plugin can be instantiated multiple times with different configurations.

Pipeline Structures A selection of configured plugins: An adapter, a connector, and an arbitrary number of intermediaries in a fixed order.

Custom Plugins Additionally to the general plugins provided by a central repository, a developer can create custom plugins to provide specialized functionality that is not provided by another plugin. Customized plugins maximize the flexibility and allow the integration of arbitrary code into the gateway. They are also a possible way for customization in the backends-for-frontends pattern.

We chose the configuration to be static, meaning to change the configuration of a Gateframe the process has to be restarted (potentially redeployed). This static configuration is in accord with the paradigm of immutable infrastructure [Vir17]. Immutable infrastructure describes the practice to recreate hosts instead of changing them. This has only become feasible through technologies like virtualization and cloud computing. We think a static configuration is appropriate for our intended use case, as for a service proxy the configuration is coupled with the service. In case of an edge API gateway, configuration should even be seen as code and mechanism used for deployment of services like testing should be applied in this case. Infrastructure automation, which is often used with microservices, is an important aspect that keeps the static configuration approach feasible.

Figure 5.2 illustrates how a gateway is created using the Gateframe, plugins, and configuration.

5.2 Execution Model

In this section, we describe how a pipeline consisting of an adapter, a connector, and multiple intermediaries behaves during initialization and while processing requests.

5.2.1 Initialization

During initialization, each plugin is instantiated using the provided configuration. For each pipeline, the Gateframe initializes the plugins in the order of the configured pipeline starting at the connector, and then in downstream direction through all intermediaries to the adapter. The reason for this order is the following: In addition to executing preliminary tasks, the connector emits a gRPC service definition that specifies the API of the provided service. Each intermediary gets this definition and can change the definition to reflect its own changes to the API interface. Each brick only sees

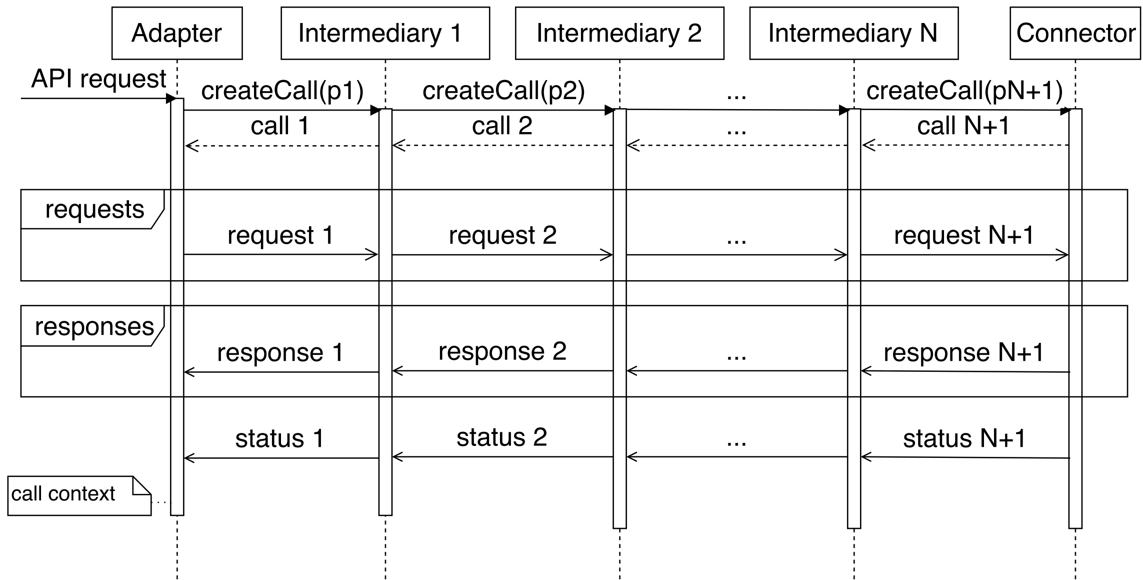

Figure 5.3: High-level concept of processing one request call: At first the call is initialized for each brick. Multiple requests and responses are handled potentially interleaved. Each intermediary can change all communication protocol elements.

the updated service definition from the immediate upstream brick. The adapter finally uses the service definition to provide its API interface, e.g., the REST adapter creates an OpenAPI interface definition.

5.2.2 Call Processing

We describe how the Gateframe processes an API call. The Gateframe definition of a call is a bidirectional streaming gRPC method call. Calls with a single request and/or response can be generalized into this view of streams as we can see a single element as a stream with precisely one element.

The general call processing in the Gateframe can be seen in Figure 5.3. An adapter initializes the processing of a call. At the time of creation of the call, the request payload does not need to be available, only the method that should be called and the metadata of the call. The connector creates the call object, e.g., after already having called the upstream gRPC service. All intermediaries can modify the returned call object. Multiple payload requests and responses can be sent in arbitrarily interleaved order and again all intermediaries can modify the objects. The pipeline also processes the final status object.

The diagram in Figure 5.3 shows a typical sequence of the Gateframe processing a call. But the intermediaries can modify this behavior in the following ways:

- The information about the request and response pipeline is contained in the parameters of the call creation and the returned call object. This allows intermediaries to process request and response payload, but also skip processing by the intermediary. This allows intermediaries to only work on the call metadata.

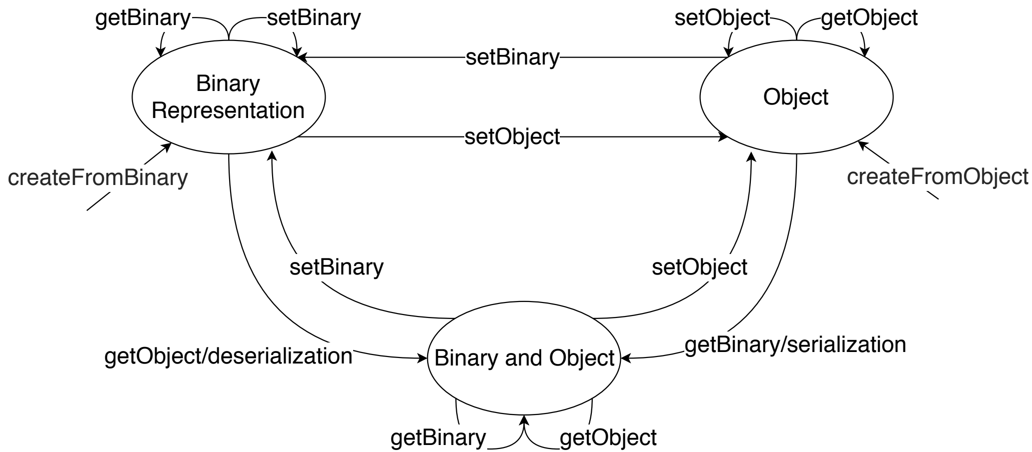

Figure 5.4: The state diagram of a lazy message format transformation with two representations, a binary serialization format and an in-memory object.

- Intermediaries can directly return a call object without calling the upstream brick. By this, they can process the request directly themselves. For example, an open circuit breaker directly returns a call object with an error status code.
- Intermediaries can create or discard request or response objects, for example for merging or splitting messages.

We emphasize that the communication model is that of a synchronous Remote Procedure Call (RPC). Through the usage of gRPC as a common protocol, the Gateframe inherits this model or more specifically the model of an RPC call as seen by gRPC. APIs based on asynchronous communication, like messaging, need API translations of the API remodeling kind to be processed by the Gateframe. The Gateframe only supports client-server based asynchronous approaches like polling or task resources.

5.2.3 Lazy Message Format Transformation

In case of connector and adapter using the same format (e.g., gRPC) and intermediaries do not or only seldom inspect the payload, (de)-serialization poses an unnecessary performance overhead. The intermediaries could themselves deserialize the Protocol Buffer, but that would introduce complexity into the intermediaries, especially if the original protocol payload is not a Protocol Buffer.

To solve the problem mentioned above, we wrap the access to the payload in a wrapper that manages access to the payload and lazily performs (de-)serialization or even conversion to other formats (e.g., into JSON). The wrapper minimizes the (de-)serialization or conversion operations.

Figure 5.4 shows a state diagram of such a wrapper object for two representations. A client of the wrapper can get either representation without having to know whether internally it exists. If the representation does not exist it will be created by conversation from the other representation. The wrapper then keeps both representations until a new value is set.

Adapters and connectors wrap request and responses in wrapper implementations for their specific protocols. All intermediaries can, if necessary, retrieve the object representation. If the object was not changed the corresponding adapter/connector can reuse the binary representation without needing to serialize the object after the element has gone through all intermediaries.

5.3 Plugin Concept

In the Gateframe plugins do not provide additional supporting features, but provide API bricks that make up the complete processing pipeline. The framework only defines interfaces and provides basic management functions like loading and instantiation of plugins.

5.3.1 Configuration

The behavior of the individual plugins should be customizable to enable adaptation to the needs of the developer. If a generic plugin is flexible in its operation, it can be used in different usage scenarios. This reduces the need for the development of custom plugins and fosters code re-usability.

Configuration from Upstream

We enable the customization through configuration of the plugin instances. There are two general channels over which a plugin receives configuration information. The first is the direct configuration of the plugin instance as depicted in Figure 5.2. The other configuration channel is from the upstream service definition, meaning from the API the plugin is calling. This configuration can originate from the definition of the original service or an upstream brick as it can change the API for the downstream. The upstream configuration is contained directly in the Protocol Buffers API definition. Protocol Buffers allow the configuration of any element of a service definition by a concept called options. An option is itself a Protocol Buffer message that is associated with a definition element.

The Gateframe currently only uses this approach of embedding the configuration into the service definition to retrieve the configuration from upstream. This approach also has the disadvantages that it clutters the proto definition and spreads the configuration to the individual service definition elements. If we want to apply a configuration to multiple elements, an option must be appended to each element individually. We could in the future extend the Gateframe to provide an additional configuration option from upstream. This could be implemented by not only providing the proto that contains the gRPC service definition but an object that contains more information about the service (Google defined a service definition for their uses [Goo17e]).

Configuration for Individual Protocol Buffer Elements

We need configuration at the level of individual elements to enable task like filtering methods, or requiring a role for a subset of methods. Another example is the mapping of the gRPC methods to a RESTful representation. One possible way to add information is adding options in Protocol Buffers [Goo17f]. Listing 5.1 shows an example of the HTTP rule for annotating the REST mapping

Listing 5.1 Example of Googles HTTP rule [Goo17c] in option of method

```
service Retrieval {
  rpc GetResource(GetResourceRequest) returns (Resource) {
        option (google.api.http).get = "/v1/resources/{resource_id}"
  }
}
message GetResourceRequest {
  string resource_id = 1;
}
message Resource {
  string some_data = 1;
}
```

Listing 5.2 Example of Googles HTTP rule [Goo17c] in gPRC API Configuration YAML

```
package resource;
service Retrieval {
  rpc GetResource(GetResourceRequest) returns (Resource);
}
message GetResourceRequest {
  string resource_id = 1;
}
message Resource {
  string some_data = 1;
}
```

Proto containing service definition.

```
http:
  rules:
    - selector: resource.Retrieval.GetResource
      get: /v1/resources/{resource_id}
```

External YAML containing the configuration.

information. The alternative is to provide the configuration not directly with the elements, but provide a means to select the elements that the configuration should be applied to. We reuse the notation of the selector defined by Google for their configurations: Each configuration rule (a rule is an element of the configuration) has a selector that defines the elements that the configuration is applied to. The selector consists of one or more patterns, separated by a comma. Each pattern is a fully qualified name for an element, or matches a sub-hierarchy (format definition: [Goo17b]). Listing 5.2 shows how a selector is used to apply the HTTP configuration to a service definition.

The Gateframe provides an implementation of the selector to the plugins, such that they can easily integrate it in their configurations.

Increase Customizability through Hooks

We can expand the ability to adapt to the user needs by implementing a default behavior of the plugin, that can be customized not only through static customization options but also by providing hooks for configuring custom implementations of specific tasks. This can be seen as an implementation of the GoF template method pattern [GHJV95].

If the configuration for a hook originates from within the runtime environment (contrary to originating from a serialized representation, e.g., from a JSON file or Protocol Buffer options), the implementation and usage of hooks can use internal references. But we can also implement hooks for serialized formats. For scripting languages like JavaScript, code from a string can be directly executed. But also for other platforms exist solutions to enable dynamic code execution, e.g., Groovy for the Java Virtual Machine.

5.3.2 Status Information

Plugins can generate information that is interesting for development, monitoring, and configuration. This information can be for example the ports an adapter is listening on or the number of requests an intermediary has seen. We want to provide a way for plugins to provide static and dynamic information and make it easily accessible through HTTP. We decided to provide this information over a REST API and not a gRPC API, as the retrieving of the information better matches this style (identification as resources) and through HTTP the information is accessible to generic HTTP clients, including browsers.

The Gateframe information interface works the following way: Each plugin can optionally provide a method that returns a dictionary with information. This method will be called each time the information is needed and thus plugins can either return a static object that is created after initialization, or they can dynamically create the information object to reflect changing values. The endpoint returns for each plugin instance the information that the instance provides as well as the configuration provided to the plugin.

The structure of the REST style API is the following: The base collections returns all pipeline configurations, with the attribute 'href' referencing the URL of the specific configuration. The specific URL is the base path concatenated with the configuration ID (which is currently an integer counter starting at 1). URLs are also available for the individual bricks to enable fine granular retrieval of information.

5.3.3 Sharing State

Some API bricks need to keep state for their operation. An example of such functionality is rate limiting. An intermediary implementing rate limiting has to remember the number of requests that have gone through the intermediary over a period. This state must be shared in the case of multiple instances, as in the example of rate limiting the number of requests over all instances must be known to all. This can be solved be externalizing the state, for example in an in-memory database. We decided not to implement state sharing into the Gateframe, and no bricks that share state are part of this work. Plugins have to implement sharing state themselves. As different bricks have varying requirements for keeping state like different consistency requirements, it is reasonable that they have

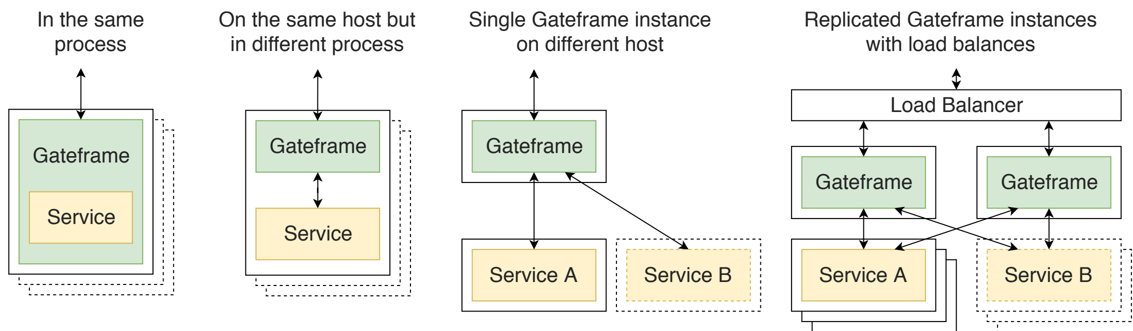

Figure 5.5: Technical deployment options

to provide the shared state themselves. This allows them to use a solution that best fits their needs. The disadvantage of this approach is the effort required from brick developers if they implement a brick that shares state between instances. The brick itself has to implements state synchronization or externalization of state. This can be an especially significant overhead for bricks that just need sharing of a single variable.

5.4 Deployment Approaches

The Gateframe is intended to be very flexible in its technical and logical deployment options.

5.4.1 Technical Deployment Options

We intend the Gateframe to be deployed in various settings from a process on the same host as the service to a replicated deployment for multiple services. Figure 5.5 illustrates the four described deployment options.

In the same process As the Gateframe is a framework for processing requests, it can also be directly used to implement a service. This can be suitable for a service that orchestrates others. Plugins provide common service functionality and the service orchestration can, for example, be implemented using a custom connector.

On the same host but in different process By deploying the gateway on the same host and only exposing the service API through the proxy, the gateway can provide common functionality. Even functionality that requires an additional level of trust, like authentication can be provided in this setup. As a gateway instance is created for each service instance, this setup scales with the service and is therefore suitable for production.

Single Gateframe instance on different host A single gateway instance is deployed for one or multiple services. This setup aims at development environments as it creates a performance bottleneck at the single gateway instance.

Replicated Gateframe instances with load balancer Multiple gateway instances allow the gateway to scale depending on performance needs. An additional load balancer may be needed in front of the instances to spread the load and hide the distribution. This setup scales with increasing requests and is therefore suitable for production.

5.4.2 Logical Deployment Options

The gateway is intended to fulfill many uses in a microservice environment.

On the outer edge to the web to provide a client optimized API Deployment as API gateway: The Gateframe can be used to build gateway solutions. Its flexibility is is an appropriate approach for the backends-for-frontends pattern.

On the edge to other systems to connect them Enables deployment of flexible solutions to other systems. A gateway can hide the structure of the system. The API translation capabilities support the integration of heterogeneous systems.

As placeholder for a legacy service/system to support integration The Gateframe can be used as a proxy to legacy systems and expose their API with another protocol using protocol translation. This follows the style of integrating legacy systems into microservices.

As wrapper to hide the distribution of a subsystem The Gateframe can be deployed in front of a small number of microservices. The functionality of these can be combined into a reduced interface intended for external use.

In front of service to provide common functionality to polyglot services The common functionality that most services have to provide can only be extracted into libraries if the service uses that same platform. The Gateframe has first-class support for the development of common service functionality through intermediaries. By deploying a gateway directly in front of the service, common functionality can be reused in a platform-independent way.

5.5 Implementation

We implemented the Gateframe as part of the Any2API ecosystem[1], as we used the concept of API bricks from Any2API.

The Gateframe implementation is written in TypeScript [Mic18b] for the Node.js platform [Nod18b]. We chose the Node.js platform for the following reasons:

- The customization of plugins through hooks can be easily achieved. Even values from configuration files can be interpreted as JavaScript code.
- The dynamic type system allows the inspection of payloads without the need the needs of special access methods like reflection.

[1] https://github.com/any2api/any2api-gateframe

- The event-based and non blocking processing of Node.js supports scalability through resource efficient processing in a single thread [Cap].
- The NPM registry already provides libraries for many use cases [Npm18].

TypeScript is a language that is compiled into JavaScript and adds static type checking. We chose to use TypeScript, as it can provide interface definitions for plugins that can be checked statically. This allows a plugin developer to check his code and thus improves the developer experience.

The implementation mainly depends on the following three packages:

gRPC [Aut18a] Implementation of gRPC for Node.js. We use the package in the Gateframe for interface definitions of gRPC elements of the intermediary format. The gRPC adapter and connector use the package to convert the internal representation to the wire format.

protobuf.js [Wir18] Implementation of Protocol Buffers for JavaScript. It performs better than Googles implementation and is used by the gRPC package. Used for the API definition format and Protocol Buffer (de-)serialization.

RxJS [Aut18e] Reactive extensions for JavaScript. We use the 'Observable' interface for all asynchronous operations during request processing. The library provides a good experience for developers when operating on request and response streams through its many predefined operators.

5.5.1 Structure

The Gateframe implementation provides the following four modules that are available in individual NPM packages only to be used when needed:

Commons Package The package contains interface definitions and shared functionality that can be useful for plugin development. The interface defines amongst others the functions that adapters, connectors, and intermediaries have to implement. Listing 5.4 shows the interface of the object passed during call creation.

Gateframe Package The package contains the code to instantiate plugins and start a gateway.

gRPC Adapter and gRPC Connector Packages The packages contain the bricks to use the common protocol gRPC for upstream or downstream connections. They transform the wire format into the internal gRPC format and vice versa.

5.5.2 Packaging of plugins

The Gateframe implementation uses NPM packages [Npm18] for the distribution of plugins. The idea is to use a concept known to developers and a technology that can leverage existing infrastructure. Using NPM packages, we can use existing repositories (the global NPM repository or private local repositories). An NPM package can contain one or more plugins, allowing the distribution of collections of smaller bricks in one package.

Listing 5.3 Minimal example Gateframe configuration. The Gateframe loads the package defined in the gRPC adapter configuration. The default plugin for a connector is gRPC. Thus we only need to provide the service definition.

```
1  const { runConfigs } = require('@any2api/gateframe');
2  const configuredGrpcAdapter = {
3    packageName: '@any2api/grpc-adapter',
4    pluginConfig: {
5      port: '0.0.0.0:9000',
6      insecure: true
7    }
8  };
9  const upstreamService = {
10   protoUrl: './service.proto',
11   host: 'localhost',
12   port: 8000
13 };
14 runConfigs([{
15   adapter: configuredGrpcAdapter,
16   intermediaries: [],
17   protoService: upstreamService
18 }]).then(() => console.log('running...'));
```

5.5.3 Configuration and Instantiation

For the creation of a gateway, we have to load the packages for the Gateframe and all bricks from NPM registries. The declaration of dependencies is made by providing a file with the name ‘package.json‘ containing the needed packages. A tool like NPM can then fetch these dependencies during development and again during deployment of the gateway.

The configuration and startup of the Gateframe are done using a JavaScript file. In the script we load all needed packages (including the Gateframe itself), then we configure the bricks, and finally, we instruct the Gateframe to execute the configuration. Listing 5.3 shows an example of such a configuration script. By using a script as configuration, we can use code to configure plugins or define custom bricks (mainly short intermediaries) directly in the configuration. Also, this allows developers the flexibility in the loading and creation of bricks.

5.5.4 Manipulation of request or response streams using RxJS

We use RxJS [Aut18e] to provide developers a consistent way of manipulating the communication. Listing 5.4 shows the interface definition of the parameters that are passed from brick to brick during call creation. The ’requestObservable’ represents the stream of requests. RxJS provides operators for modifying the behavior of an observable as well as modification or inspection of the object in the stream.

Listing 5.4 Example interface definition for the parameters that are passed from bricks to brick during the call creation. Extracted from https://github.com/any2api/any2api-gateframe/blob/f8ce0405af260383e8b4fb1e6f7bee93fd1510fa/packages/common/lib/interfaces/request.ts

```
1 export interface RequestParameters {
2   method: { namespace: string, name: string };
3   type: GrpcMethodType;
4   requestObservable: Observable<MessageAccessor<{}>>;
5   responseType: ProtoBuf.Type;
6   metadata?: Metadata;
7   callOptions?: CallOptions;
8 }
```

The following code is an example extract of an intermediary that manipulates the ’requestObservable’ to write the request objects to console:

```
1 makeRequest: (requestParameters) => {
2   requestParameters.requestObservable =
3         requestParameters.requestObservable.do((r) => console.log(r.getMessage()));
4
5   return upstream.makeRequest(requestParameters);
6 }
```

The ’do’ operator of the reactive extensions [Rea18] allows executing an action for each object sent in a stream.

6 Framework Plugins for Common API Functionality: API Bricks

The Gateframe architecture is designed for the combination of API bricks to build up complete request processing pipelines. This chapter contains concepts of individual bricks that are referential for essential functionality needed in microservices. We based the relevance of this functionality from the existing microservice we examined in the domain of connected cars. We have implemented a selected number of these concepts as plugins for the Gateframe.

We separate the bricks into three categories: Adapters, connectors, and intermediaries. We introduced those terms in Section 5.1.3.

For the adapter and connector, we only consider protocol translation, as protocol modeling and especially API remodeling can introduce state into bricks. We solely look at bricks that are stateless or keep a small local state like a circuit breaker. We think that these bricks are much easier to use and therefore are a much better fit as generalizing bricks.

6.1 REST and OpenAPI Adapter

REST over HTTP is widely used in service computing, and most microservices use HTTP based APIs [FL14]. The usage of gRPC is restricted, as we currently call a gRPC API directly from a web browser. For this reason, there already exists a mapping definition from a gRPC service definition to REST over HTTP [Goo17c]. The REST adapter provides this mapping of gRPC to REST for the Gateframe. The support for REST over HTTP increases the number of devices that can use a service that makes use of the Gateframe. Thus this adapter is relevant for practical use cases.

OpenAPI [Ope18a] allows us to provide a specification of the structure of a service API. The gRCP API from upstream provides a service definition, that we can use to create an OpenAPI service definition. This allows easier consumption by other services, e.g., through the automatic generation of client code.

The REST adapter fulfills the following tasks:

1. The adapter consumes the gRPC service definition from the upstream bricks with annotated HTTP rules that influence the structure of the REST API. The mapping to the REST API is consistent with other gRPC to REST mapping implementations by being consistent with the mapping defined in [Goo17c].
2. The adapter uses the service definition and creates an OpenAPI 3 definition during the initialization.

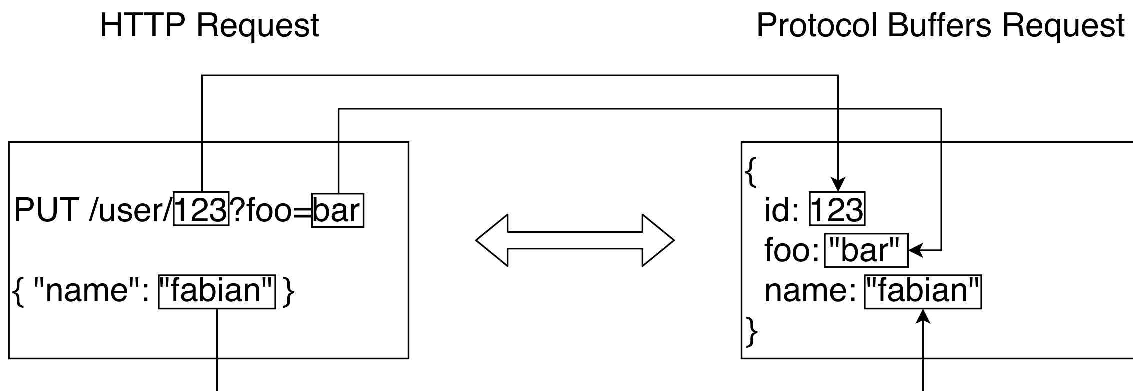

Figure 6.1: Illustration of the mapping from an HTTP request and a Protocol Buffers request message. The mapping can be done both ways. The elements from the Protocol Buffers message are mapped from path, query or body parameters of the HTTP request.

3. The adapter not only provides access to the OpenAPI definition but also provides a human-usable interface for testing the API (Swagger UI[1]).
4. The adapter correlates the incoming requests to gRPC methods and creates the request object from a combination of path, query and body parameters.
5. The adapter provides the ability to set gRPC metadata through HTTP headers (preferably through mapping of standardized HTTP headers with similar meaning).

6.1.1 Implementation Notes

The implementation uses the express.js framework [Nod18a] to serve the REST API, the OpenAPI definition, and the Swagger UI.

To support the path templates defined in [Goo17c] we used ohm [Ohm18] to specify a grammar that matches the template string. Regular expressions cannot be used for this tasks, as the path templates do not form a regular grammar. The following example template used by the mapping illustrates the nonregularity (the language is non regular as it contains nested pairs of brackets):

```
1  /foo/{a=*/bar/{b}}
```

The grammar allows correct and easy extraction of the parameters that are set using path parts and the creation of a regular expression needed to match the paths that match the path template.

The support for streaming currently maps a stream of Protocol Buffers objects to a JSON array. This means a requestor has to send an array of JSON objects for a client streaming call, which each object being the JSON equivalent of the request type of the gRPC method. The mapping to HTTP does not allow interactive bidirectional streaming. The approach of using an array to map to a stream creates endpoints with regular JSON. The arrays are no problem with small amounts of data per request. For large amounts client or server have to receive the complete array before

[1] https://swagger.io/swagger-ui/

deserialization, that can lead to memory problems especially on the server. We can improve the problem by sending multiple JSON objects in a nonstandard, streaming, kind of way. For example, the gRPC gateway [Aut] maps streams to newline delimited JSON objects. The extensible service proxy [Goo18a] uses a correct JSON format, but essentially writes the array elements '[,' ',' and ']' manually. A client, to consume the elements in a streaming way, would have to manually split the array to access elements before the transmission is over.

We only implemented JSON as a data type, as it is the only mapping that is defined for Protocol Buffers and it is the most widely used serialization format for public Web APIs. As we use protobuf.js [Wir18] for the JSON conversion, we rely on their support for well-known types, which currently is very restricted. Thus currently we only support the well-known type 'Any' [Goo17f] completely. 'Timestamp' [Goo17f] is supported in the direction of conversion from Protocol Buffers to JSON. The missing support does not break the usage of well-known types, but exposes the Protocol Buffers structure of their values and instead of using the JSON representation specially defined for the types.

6.2 REST and OpenAPI Connector

As for the adapter, this connector is relevant for the usage of the Gateframe for services that provide a REST API. The REST connector allows for faster adaptation of the Gateframe in microservices architectures that heavily rely on REST/HTTP.

We require the upstream REST/HTTP service to be defined using an OpenAPI definition. From the OpenAPI definition, the connector creates the gRPC service definition that is needed by the downstream bricks.

The following details describe the behavior of the REST connector:

- The connector merges the HTTP requests parameter from path, query, and body into the definition of the request message format (see Figure 6.1). This is relevant for the creation of the gRPC service definition, as well as during runtime.
- The connector creates only unary gRPC methods from the REST endpoints.
- The created service definition contains the REST mapping annotation information. With this information, a REST adapter can map the API back to REST. The mapping from REST and back enables using the Gateframe for providing common service functionality for REST APIs.
- The connector can map only endpoints that support JSON as the content type for request and response, as JSON is the only mapping defined for Protocol Buffers.
- Details of the gRPC API, like the package name, that the connector cannot meaningfully infer from the OpenAPI definition, can be provided in the configuration.

We have not implemented the REST connector as part of this work.

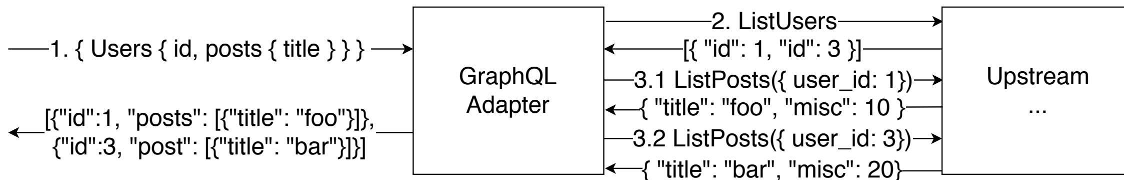

Figure 6.2: Example showing the basic idea of the GraphQL adapter: The adapter receives a request for a listing of messages titles. Through correlation of a method that can list the messages and another that can resolve the details, the adapter fulfills the requests.

6.3 GraphQL Adapter

The idea of the GraphQL adapter is to serve a GraphQL API for an existing service with minimal additional information and without any modification to the service. A client can request to read data from one or multiple methods with a single request, and the server returns only those parts of the information to the client that the client explicitly requested.

Figure 6.2 shows an example of request processing of a GraphQL adapter.

6.3.1 gRPC to GraphQL mapping

We think the GraphQL adapter should work with minimal configuration. Thus the GraphQL adapter only needs the proto definition with REST mapping annotations [Goo17c] and adapter-specific relation annotations.

The types of the GraphQL schema are created from the proto definition by using similar primitive data types. E.g., 'int32' is mapped to an 'Int' in GraphQL. As GraphQL is intended to have a type system similar to JSON, the JSON schema equivalent to the proto definition also provides a good starting point for the mapping.

We reuse the REST mapping annotations as the semantics of the HTTP verb 'GET' correlates to 'Queries' in GraphQL. Thus the adapter creates an entry in the 'Query' type for all methods annotated with 'Get.' Both describe reading operations that do not change the state. All methods with other verbs, or no HTTP annotation, are mapped to GraphQL 'Mutations', as they are potentially unsafe and may change the state of the system. The mapping service definition that contains no save operations (reading of some information) that are annotated as 'GET,' is not only not able to leverage a key feature of GraphQL, namely the spreading of reading request to multiple services. But the adapter cannot create a correct GraphQL schema from this service, as each schema needs at least one 'Query' entry point.

For the GraphQL adapter to be useful, the design of the service should be resource-oriented CRUD operations, or at least reading and listing of resources should be supported. If the resources have some relations between each other, e.g., users and their posts. Then the adapter can fetch the correlated resources. The relations between instances are typically only described in the verbal documentation of the service. Therefore we introduce an option to configure relations for the adapter.

We annotate the following elements for a message data type relation:

- The name of the property that should represent the relationship in the GraphQL schema, e.g. 'posts' on a message type 'Users.'
- The identifying name of the method that the adapter should call to resolve the relation, e.g., 'GetPosts.' The signature of the method defines the type of the relation in the GraphQL schema. A server streaming method indicates that the type should be a list.
- Mapping information of properties from the resource to the parameters of the method, that allow the automatic resolving of the relation. For example, the field 'id' of 'User' is used for the value of 'user_id' in the request of 'getPosts.'

The example in Listing 6.1 illustrates how the GraphQL adapter works.

We have not implemented the GraphQL adapter as part of this work.

6.4 Interface Filter Intermediary

Not all functionality of a service is meant to be exposed beyond the reach of the network or scope it resides. An API Gateway deployed at the edge of the network should not expose internal functionality to other untrusted networks.

6.4.1 Configuration

The idea behind the interface filter is to remove all internal elements from the definition, requests, and responses of the service. We can filter various elements of a service definition, like methods, fields of requests or responses, or even types. If a methods request or response message type is filtered out, the method itself will also be filtered out.

These elements can be directly annotated in the proto definition to be only accessible during specific conditions (whitelist). A simple example of this is the visibility configuration of the Google API [Goo18d]: An element specifies labels and access is granted to those consumers that have at least one of these labels associated. The intermediary can determine those labels by configuration, to implement a gateway for different environments, or by using for example metadata. The configuration alternative is to define selectors in the config file to hide certain elements (blacklist). The default visibility of this configuration, if no rule is set, is visible.

6.4.2 Behavior of intermediary

The filter intermediary should have the following behavior during initialization and request execution:

During initialization, the filter intermediary deletes all elements from the proto definition, that no consumer can use. This means deletion of methods, fields, etc. This behavior is not security relevant, but it minimizes the size of the service definition and thus optimizes the comprehensibility.

During execution the filter does the following for **hidden elements** (and only for those):

Listing 6.1 This example shows a service definition as a proto file and the GraphQL schema the GraphQL adapter creates from this service definition.

```
service UserService {
  rpc ListUsers(google.protobuf.Empty) returns (stream User) {
    option (google.api.http).get = "/users";
  }
  rpc ListPosts(ListPostsRequest) returns (stream Post) {
    option (google.api.http).get = "/users/{user_id}/posts";
 }
}
message User {
  int32 id = 1;
  option (any2api.graphql.reference) = {
    name: "posts"
    resolver:  "UserService.ListPosts"
    param: {
      from: 1
      to: 1
    }
  };
}
message ListPostsRequest {
  int32 user_id = 1;
}
message Post {
  string title = 1;
  int32 misc = 2;
}
```

```
type Query {
  Users: [User]
  Posts(userId: String): [Post]
}
type User {
  id: Int
  posts: [Post]
}
type Post {
  title: String
  misc: Int
}
```

Methods The intermediary resolves any request to hidden methods directly with a status code of '7' (permission denied). It does not forward the request upstream.

Types and Fields For all requests and responses, the intermediary removes hidden fields (or fields with a hidden type). Technically it sets them to default values in such a way that any upstream handler cannot distinguish this behavior to the element not being set at all.

6.4.3 Implementation Notes

As the performance of the filter is important, the filter is prepared during initialization to execute the request in a timely manner.

This implementation does not filter the Protocol Buffers 'Any' types. This means it does not determine the payload type of 'Any' at runtime to evaluate the selector.

Filtering of Methods

During the initialization of the intermediary, the selector is tested against all methods. The filter creates a set of all method names from the service definition that did not match the selector. During a request the filter tests the set for the existence of the called method name. This means a method not defined in the service definition will always be rejected.

On rejection of a request the filter directly returns a completed call observable with the status code set to '7' (permission denied) and an empty 'Observable' of responses.

Filtering of Fields

The plugin prepares the filtering of fields by creating handlers for each message type. It creates the handlers starting at the request and response types of the visible methods. The handler of a message types does the following: For each hidden type, it resets the value to the default. For each visible nested message, it recursively calls the filtering handler.

With the premise that it is likely for message types to contain no filtered elements and for nested types to occur frequently, the idea to optimize is the following: Remove all filtering calls on message types that contain no fields to be filtered. The following algorithm creates a set of handlers only for types that directly contain fields that should be filtered or contain nested messages of types that should be filtered. We emphasize on the recursive relation of this condition.

The algorithm works in four phases:

1. Create dependency graph of message types. An edge from type A to B indicates that A has a field of type B that is not hidden.
2. Mark all types that contain a hidden field.
3. Recursively mark all message types that reference a marked type.
4. Create filter handlers for all marked message types. With each filter calling the filter or nested messages which are marked.

Figure 6.3 shows an illustrating example. In step 1 the graph is created. Type A has the references as it contains fields with the types A, B, and C that are not hidden. These references indicate possible sub-elements from a data element of this type. In step 2 we mark B, as B contains a hidden field of which the value has to be deleted during request processing. In step 3 we also mark A as it references type B and transitively A has to be processed as well. In step 4 we create the handlers for the deletion of field values for type A and B. The handler of type A calls the handler of type B for all its fields of type B.

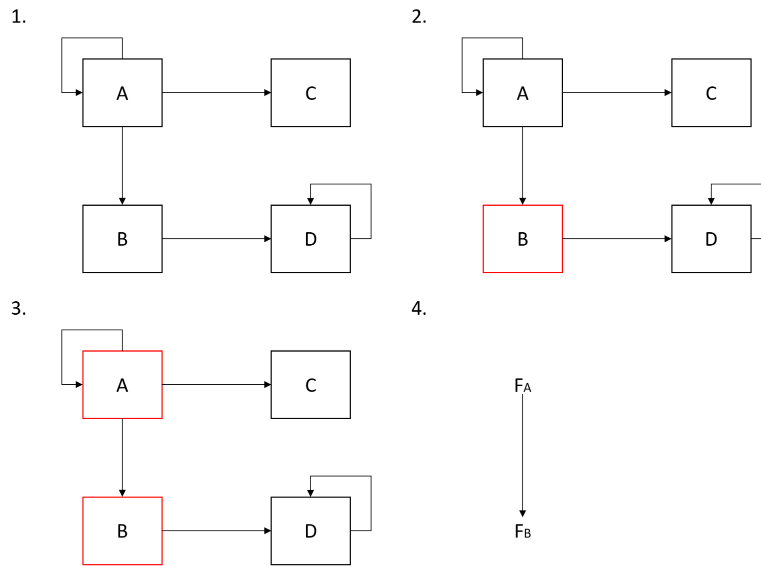

Figure 6.3: This example shows the preparation step of the filter optimization with four types. Only fields in B need filtering. If type A is used by a request or a response of a method, it only needs to call the filter of existing fields of type B.

6.5 Authentication Intermediary

Authentication means identifying and validating the subject that is associated with a request. This task has to be repeated for every service in need of authentication, and an expert should do the job of writing critical security code. Thus authentication is a suitable task to to be implemented as a separate intermediary.

The general task of an authentication intermediary is identifying the subject of the requests, potentially with additional claims, and validation of the identity.

6.5.1 JWT validation intermediary

We use a JSON Web Token (JWT) validation intermediary as a reference for an authentication intermediary. Companies like Google and Microsoft use OpenID Connect for authentication solutions [Ope18b] and thus also JWT. JWT provide the ability to validate authentication with reduced communication to an identity service.

The intermediary fulfills the following tasks:

Validation Check that the token is in the correct format.

Verification Check that the token is in the correct format. Ensure that a trusted identity service administered the JWT by validating the signature of the token and is of valid format.

Enforce Neccessary Information Ensure the existence of necessary claims in the JWT.

The assumption is that only JWTs are used for authentication and that they are signed, but not encrypted. Additionally, this intermediary does not provide other claims from an identity service but assumes that all necessary claims are already available in the token. Thus the only task that is done by this intermediary is the validation of the JWT as described and ensuring some additional configured constraints. This means it checks whether the signature of the request is from an identity provider (issuer) that is allowed to authenticate the requestor for the method that is called and that the JWT audiences intersect with those defined for the method.

After the filter, the request can be seen as preauthenticated. While the filter does not manipulate the request, it ensures the authenticity of the JWT. Authorization, meaning determining whether the authenticated subject is allowed to access the requested resources, is done by the application itself or by other intermediaries (e.g., the authorization intermediary described in Section 6.6).

6.5.2 Configuration

Google already defines a set of proto definitions that directly match this use case which we will reuse [Goo17a].

Authentication Configuration container for rules and providers.

AuthProvider Defines an identity provider including issuer, JSON Web Key (JWK) URL to retrieve the public signing keys, and a list of audiences that are accepted for requests.

AuthRequirement Configuration that references a required 'AuthProvider'.

AuthenticationRule Contains a selector to determine the methods to apply a rule to and a list of 'AuthRequirements'.

These configuration entities are used to define the needed requirements for a method that the filter will validate.

6.5.3 Implementation Notes

The public keys needed for validation of the JWT are lazily retrieved from a URL. The location where the keys can be retrieved should conform to 'jwks_uri' as defined in the OpenId Connect Discovery specification [SBJJ14].

The intermediary executes the following for each request if an 'AuthenticationRule' with at least one 'AuthRequirement' matches the called method:
The intermediary checks if

- a JWT is present in the authentication header.
- the JWT is not expired.
- the JWT was signed by one the identity providers defined in the 'AuthenticationRule' that matches the called method.
- the intersection of of the audiences defined in the 'AuthProvider' with those in the JWT is not empty.

If a check evaluates to false the intermediary will immediately return a call with status '16' (Unauthenticated).

6.6 Authorization Intermediary

If the gateway is deployed as a sidecar, it can provide features for a common task like authorization. While it can be problematic to authorize a request at the edge directly, a request can be securely authorized by a sidecar process on the same host. Some restriction still applies to the kinds of authorization that can be provided by a generic authorization plugin: Simple scenarios like authorization based on claims of the subject (e.g., required role) can be easily implemented. But the plugin cannot implement authorization based on the business objects. A common example is checking whether the requestor is the owner of the accessed object. The authorization plugin should only be used if the service only needs basic authorization like requiring a specific role from a requestor. If the service validates authorization based on business objects, then the service already contains logic for authorization. We think that splitting authorization into service and a service proxy in front of the service creates unnecessary complexity.

The authorization plugin supports the scenario mentioned above for the simple case of requiring certain scopes to be present in a JWT. Additionally, through hooks, we can add additional requirements to the claims and request. The configuration contains a list of rules with a selector of which the intermediary uses the rule with the last matching selector.

6.6.1 Implementation Notes

The use of JavaScript allows the configuration of hooks to retrieve the claims from different properties of the request. This property can, for example, be a custom header that contains a JWT or another representation of claims. The flexibility to define code allows us to also provide for checking the claims and deciding on the authorization.

If authorization for a call is not granted the intermediary directly returns the call with a status code of 7 (Permission Denied).

6.7 API-Level Monitoring Intermediary

Monitoring is an essential part of microservices as in case of failures engineers can easier find the services to look at [New15]. The gateway is not able to retrieve direct metrics about the service and only about the host if running on the same host. But a gateway can extract basic API information like request duration and status codes from inspecting the communication with the service.

We chose to use the gRPC status codes and request durations as metrics for the monitoring intermediary. The metrics are collected for each method individually, as in gRPC long request duration not necessarily means a problem, it could be a streaming request.

We chose to use Prometheus [Pro18] as the monitoring solution our intermediary is connected to. Prometheus is open source and a member project of the Cloud Native Computation Foundation and it is a highly scalable monitoring solution for microservices.

The monitoring intermediary provides the collected metrics of all request sent through the gateway by providing a Prometheus pull interface. The interface an HTTP endpoint that returns the current values of the metrics.

6.7.1 Implementation notes

The monitoring intermediary captures the request durations using a histogram. Each request increases the counter of a bucket. This does not allow to see exact durations but only a rough distribution of request durations. A counter records the status codes for each method and status code combination. This inspection is on the level of methods and actual error status codes.

The implementation uses express.js [Nod18a] to serve the pull interface and a npm package called 'prom-client' [Nyb18] to create the metric representation to serve at the endpoint.

In the current implementation, each instance of the intermediary serves the pull interface. In case of multiple service pipelines on the same Gateframe instance, this means that a Prometheus service has to retrieve metrics from multiple ports on the same host.

6.8 Circuit Breaker Intermediary

The circuit breaker intermediary helps to increase the robustness of the service by failing fast when the upstream service experiences issues. Not all gRPC status codes indicate a problem with the service; some indicate a problem with the request. Thus the circuit breaker does not count codes like '14' (Unauthenticated). As the exact usage of status codes can differ, the intermediary allows configuring the codes that it sees as a service overload indicator.

The circuit breaker uses the same states and state changes as the Hystrix commands by Netflix [Net17]. It uses the same bucket schema to determine the error rates, but only success and errors are counted. The circuit breaker allows defining the circuit breaker on the complete service, groups of methods, or even individual methods. This defines how the errors for the services are counted.

6.8.1 Implementation notes

The circuit breaker intermediary has a state, that counts the number of errors and successes of requests. This state is not shared by a replicated gateway. As the goal of the state is not to be accurate but to detect the overload of an upstream service, it is reasonable that each instance can have a local state and their view on the health of the upstream service.

For the practical implementation of the circuit breaker pattern, it is important to specify a volume threshold, which defines some request per time under which the circuit breaker is inactive. Few requests do not provide a correct view on the health of the upstream service and bear the risk that outliers trigger the circuit breaker. Getting the threshold parameter right can be important tasks in ensuring the benefit of a circuit breaker.

7 Evaluation

In the last chapter, we introduced the API bricks necessary to add common service functionality to the Gateframe to build gateways. In this chapter, we evaluate the concepts and implementation of the Gateframe together with API bricks. The primary focus lies on the use of the framework as a service proxy to provide common service functionality for one service.

7.1 Preliminary Work

In this section we introduce some practical work we did, that we categorize as exemplary applications of the Gateframe, allowing to estimate benefits and introduced complexity by using the Gateframe. This work consists of the development of an example application that makes extensive use of the Gateframe, refactoring of existing service from REST to gRPC, the integration of a custom authentication library for the re-factored service, and the deployment of the Gateframe as a service proxy on Cloud Foundry.

7.1.1 Example Service: API-Talk

We developed an example application called 'API-Talk.' A very simple social network that is about talking about APIs through APIs. The idea is that the service provides many different APIs through which developers can interact using their own clients. The data model depicted in Figure 7.1 provides a simplified view on the resources managed by the service. The design goal of the service is to cover many features of gRPC and Protocol Buffers. The service is then deployed with all implemented API bricks to test the individual bricks as well as the integration of all components.

The service uses the following aspects of gRPC:

- All four request types: Unary, client-streaming, server-streaming, bidirectional-streaming.
- gRPC metadata.
- Single and repeated fields.
- All scalar value types.

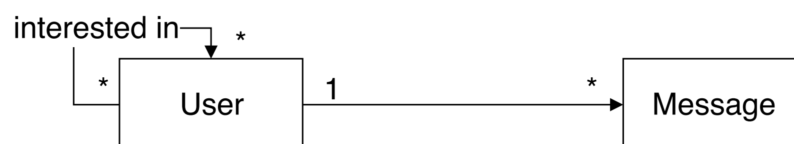

Figure 7.1: Data model of gRPC example service API Talk.

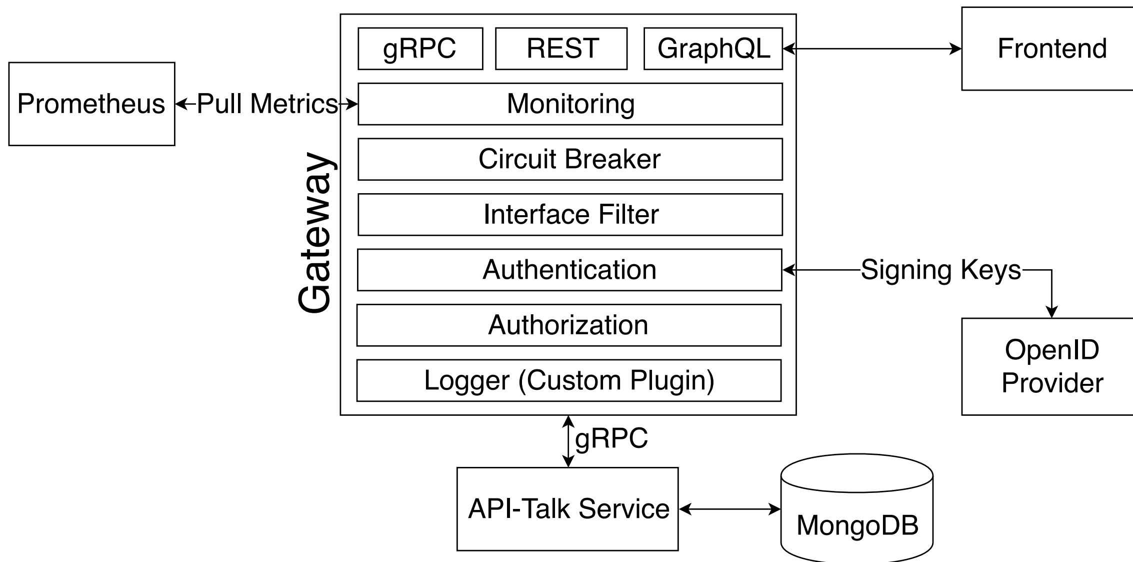

Figure 7.2: Overview of the setup used to test the API bricks. The service itself provides the gRPC API. Through the gateway an additional REST/HTTP and GraphQL API are provided. Intermediaries add different common service functionality. A logger that prints request and responses is provided as a custom plugin. It is directly defined in the JavaScript file that configures the Gateframe.

- Well known types ’Any’ and ’Empty’.
- Enumeration type.
- Map type.

We know the following features that the service does not implement:

- Explicit usage of deadlines or timeouts.
- Well known types other than ’Any’ or ’Empty.’
- Encryption of connection with TLS.

The service was developed in Java using the Spring Boot framework [Piv18a] using a gRPC starter [LOG18]. The service uses an instance of MongoDB [Mon18] to persist data. The technology was chosen to be similar to an existing service that we changed over from REST to gRPC.

Figure 7.2 shows the setup used to test the Gateframe and bricks with the example service. The elements in the gateway illustrate the configured pipelines of the gateway. The gateway provides three pipelines with a gRPC, a REST and a GraphQL API that have the same intermediaries configured. The setup provides a simple frontend to display some data to demonstrate the usage of the GraphQL API. The monitoring intermediary offers metrics to a Prometheus. The authentication intermediary retrieves signing key for the JSON Web Token (JWT) validation from a simple implementation of an OpenID provider.

The example setup is deployed using Docker. Each block and the database is an individual container. Therefore this setup is a deployment of the service proxy and the service itself on different hosts. Through the orchestration tool of docker 'docker-compose,' we define the complete setup configuration in one file and, can then deploy the whole setup with a single command on a developer machine.

The implementation of the example service shows us that infrastructure as code, the description of deployment setup in a file format which can be instantiated, is a good fit for the deployment of a service proxy on a different host than the service itself. The Gateframe is to some degree coupled to the service, e.g., changes in the API may need configuration changes in the gateway. Infrastructure as code like docker-compose provides a reprehensible way to develop services that consist of more than one process and platform technology.

7.1.2 Refactoring of a REST Service to gRPC

To evaluate the steps needed we re-factored a REST API of an existing service to a gRPC API. This existing service is located in the domain of connected cars and provides a REST API for the management of tax-conform logbooks. The primary resources managed by the API are logbooks and their corresponding trip entries. Logbooks can be created through the API; trips are then automatically added through the connection with the car. The API provides the retrieval of the information and some modification of the trips. These are small corrections like the correction of start/end address or the indication whether the trip was personal or business.

The service uses the Spring Boot framework [Piv18a]. Thus we decided to use a Spring Boot starter for gRPC [LOG18] to leverage the framework for a smooth restructuring to a gRPC service.

We did the following steps:

- We re-factored the code to separate all REST/HTTP depending code from the business logic. In our case the separation was already nearly completely existing. The complexity of these steps greatly varies on the structure of existing code.
- We created a gRPC service definition in a proto file. The service definition we created not only provides the same functionality as the old service, but we designed the gRPC API and the corresponding HTTP/JSON mapping to be compatible. This compatibility of the mapping means that the REST adapter creates an API from the gRPC service definition that is compatible with the original API. This allows us to change the API from REST to gRPC while still providing the original REST API through the REST adapter. The restriction of the approach is that the original API must be based on JSON.
- We added the gRPC dependencies to the project and generated server stubs.
- We implemented the generated stub. The implementation generally only converts the requests from the generated Protocol Buffers representation to the internal domain representation and then calls the corresponding business function.
- We reimplemented the used common service functionality to be compatible with gRPC. In the context of Java EE applications server and the servlet API, common service functionality is often implemented on top of servlet filers [Ora11]. The possibility to process incoming or outgoing messages exists in many service frameworks. The gRPC library uses a similar

concept called interceptors [Aut18c], but those have a different interface from servlet filters. Thus we have to create interceptors with the same functionality for filters that provide common service functionality. We can simplify this tasks by refactoring the code of existing filters to separate the servlet filter specific code as we did for the REST API itself. How we did this for a custom Spring security [Piv18b] library, is described below in Section 7.1.3.

If the existing service uses a blocking implementation, i.e., by executing each request in a distinct thread, we can gradually change the complete service to a reactive implementation. A reaction implementation allows to leverage the asynchronism of gRPC and enables easier scaling. We have not done this for the existing service, but this optimization should be done for a production-ready service.

We conclude that services that structure their business logic from their API logic can be refactored to other API technologies with reasonable effort. The more the code depends on API technology-specific interfaces, the more refactoring has to be done, or even elements have to be implemented again. For common service functionality, instead of reimplementing them for the gRPC framework, we can directly extract the common service functionality into an intermediary of a Gateframe instance that we deploy as service proxy.

7.1.3 Integration of a Custom Spring Security Library into gRPC

The only custom service functionality the existing REST service uses is a library for integrating custom pre-authenticated requests into Spring security [Piv18b]. A pre-authenticated request is a request that originates from a trusted party that already authenticated the request and added the authentication information (e.g., a gateway), the service blindly trusts the information provided.

The existing authentication library reads information from (custom) HTTP headers, retrieves additional information from an identity service and adds the information to the security context. From this information authorization then can be evaluated, like checking that its owner can only access some business object. This functionality cannot be extracted into an intermediary, as we can only set the security context from within the same process. Thus we implemented gRPC interceptors that reimplement the servlet filters of an existing library. We made use of existing functionality that the library provides and is independent of servlet filters. The reimplementation contains a lot of duplicated code from servlet filters, as circumstances prevented refactoring of the existing code. We think that in general, if refactoring of existing code is an option, gRPC interceptors equivalent to existing servlet filters can be implemented easily.

As the authentication information in the request headers is already 'plaintext,' the only thing we did on the side of the gateway in front of the service was to ensure backward compatibility for the REST API by configuring the REST adapter to transfer the information in the HTTP headers into the gRPC metadata.

We used a tutorial as inspiration and base on how to integrate gRPC with Spring security [Lei17].

7.1.4 Packaging Gateframe with Service on Cloud Foundry

We looked at how to deploy the Gateframe on Cloud Foundry. We think that, at least at the time of the writing of this work, it was not feasible to deploy a service and a Gateframe in different apps in Cloud Foundry, meaning we would deploy them on a different host and scale them independently. The unfeasibility of this approach originates in the lack of support of HTTP/2 and gRPC in the Cloud Foundry load balancer [17], and that would be needed to support the required distribution of the requests.

Because of the restriction mentioned above, we deploy the service and the Gateframe as a service proxy as two processes on the same host. The setup is a Spring boot application with a Gateframe instance in front that provides translation for the provided gRPC API to REST/HTTP (we do not expose the gRPC API due to the lack of HTTP/2 support). Thus we need a Java and a Node.js buildpack to deploy the setup. As Cloud Foundry buildpacks are used to support only applications on one platform, we need to use the multi buildpack mechanism that is provided by a 'multi-buildpack' that executes other buildpacks [Clo18a]. The standard execution of Cloud Foundry only supports the start of a single process. Therefore we also modified the multi-buildpack to allow additional start commands to start the service proxy and service. We implemented the starting of multiple processes in such way that exiting of one process leads to the termination of all others. We think this is a reasonable default behavior as it prevents running instances that have partially failed.

With the prepared buildpack we create a package (a directory with a manifest) consisting of the following elements:

- We create a Cloud Foundry manifest file ('manifest.yml') that describes the requirements and dependencies, i.e. how much RAM the app needs or to which database service to connect. In this file we configure our modified multi-buildpack to be used.
- Additionally we create an additional configuration file for the multi-builpack that configures buildpacks to be used, in our case the Java buildpack and the Node.js buildpack, and the additional start commands. In our case the Java application is automatically started and we add a start command for the gateway.
- We add the Java application to the package by extracting the Java package into the folder at root level. It is necessary to extract the files instead of adding only the Java package to conform to the file structure expected by the Java buildpack.
- We add all source files of the Gateframe project (Node.js application) to the folder. We configure the Node.js application to package itself into a self contained binary. As the Java buildpack is used to execute the application, the binaries from the Node.js buildpack are not available during execution.

While this is a possible solution for deploying a service proxy to the Cloud Foundry platform, we think that due to the introduction of many custom steps such deployments can be hard to maintain. In conclusion, we think the deployment of a sidecar in Cloud Foundry is hard to achieve while not interfering with the convenience of Platform as a Service (PaaS).

7.2 Gateframe as Service Proxy: API Translation and Common Service Functionality

We evaluate the Gateframe as a service proxy and the impacts of providing API translation and common service functionality in a service proxy that is deployed as a sidecar to the service. We compare two setups. The first setup is a service that provides a gRPC and a REST API for the same business functionality independently and the second setup consists of a service with a gRPC API that provides an additional REST API through a service proxy.

We measure the performance of the setups to estimate the performance impact of Gateframe as a service proxy. Additionally, we list and discuss some aspects that differentiate the two setups. From this, we conclude on some benefits and challenges that arise from sidecar service proxies.

7.2.1 Setup

For the evaluation, we created a simple service that returns a single object that it retrieves from a MongoDB [Mon18]. Figure 7.3 shows the structure of the two setups we compare. We think this setup is a reasonable approximation of how a real-world service using any of the two approaches looks. Setup A is a Java application based on an app server that provides a gRPC and a REST API by using shared business functionality. The other elements for providing the two different APIs are not shared and are independent of each other. Setup B provides only a gRPC API without the logging interceptor for its service. The business logic is the same as A. The service proxy in front of the service offers the logging functionality and the REST API that is translated from the gRPC API. We implement the service proxy using our Gateframe implementation. The logging components are a placeholder for all common service functionality in general. The implementation of this evaluation writes the API requests to the console.

7.2.2 Performance Impact of Gateway (Additional Hop)

We evaluate the impact of an additional hop in the API communication. Our goal is to get a rough estimate of the magnitude of the performance impact of our gateway in front of a service. We do this by measuring the time needed to fulfill a fixed number of requests. We also look at different degrees of parallelism. We do not intend to achieve the minimal possible values or compare the performance difference between gRPC and HTTP. This setup contains too many unknown factors for that. We only compare the times of the same API technology in setup A and setup B.

For the performance evaluation, we deployed the two setups using Docker for Windows [Doc18]. For measuring the performance, we used a program we have written in Go [Go 18]. We chose Go because of its concurrency model to be able to send requests in parallel easily. We used three configurations with 100, 1000, and 2000 requests and 1, 10, and 20 requests in parallel respectively. We run each setup ten times to make the results more robust, as through the dependencies on many factors of the system the values can fluctuate a lot. The computer used is a laptop running Windows 10, with an Intel Core i7-6820HQ @ 2.7 MHz and 16GB of RAM.

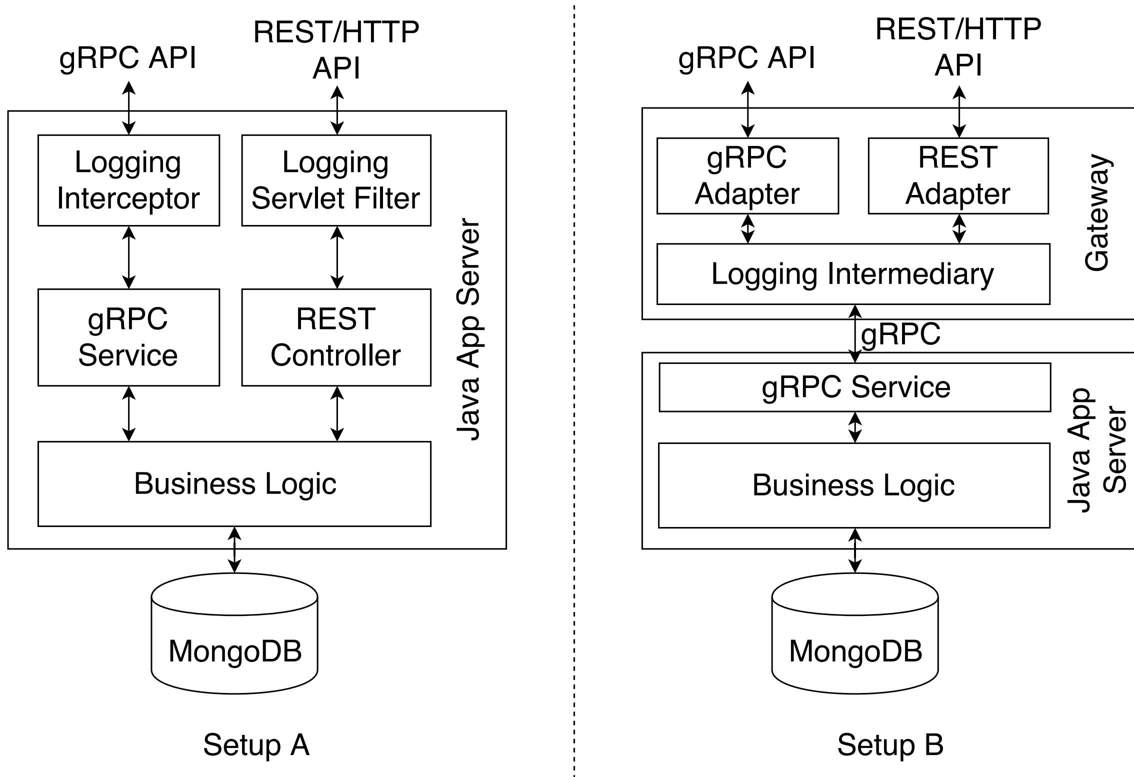

Figure 7.3: Two setups of services that use common service functionality and provide a gRPC API as well as a REST API. The first setup only uses a single platform while the second setup consists of a Java app server and the gateway that uses the Gateframe based on Node.js.

	Setup A			Setup B		
	100 request	1000 request	2000 request	100 request	1000 request	2000 request
	1 parallel	10 parallel	20 parallel	1 parallel	10 parallel	20 parallel
Round 1	0.69 s	3.29 s	6.22 s	0.73 s	4.41 s	7.26 s
Round 2	0.55 s	3.16 s	6.05 s	0.82 s	3.92 s	6.40 s
Round 3	0.53 s	2.88 s	5.88 s	0.85 s	3.79 s	6.95 s
Round 4	0.57 s	3.54 s	5.78 s	0.62 s	3.93 s	7.83 s
Round 5	0.53 s	3.21 s	5.34 s	0.73 s	3.92 s	7.68 s
Round 6	0.68 s	2.83 s	5.87 s	0.64 s	3.78 s	7.60 s
Round 7	0.51 s	3.17 s	5.70 s	0.64 s	3.91 s	7.08 s
Round 8	0.54 s	3.04 s	5.68 s	0.69 s	3.75 s	6.73 s
Round 9	0.50 s	2.96 s	6.19 s	0.65 s	3.73 s	6.68 s
Round 10	0.52 s	3.09 s	5.64 s	0.67 s	3.63 s	6.17 s

Table 7.1: Measured data from HTTP performance evaluation

	Setup A			Setup B		
	100 request	1000 request	2000 request	100 request	1000 request	2000 request
	1 parallel	10 parallel	20 parallel	1 parallel	10 parallel	20 parallel
Round 1	0.17 s	0.61 s	0.85 s	0.29 s	0.87 s	1.25 s
Round 2	0.17 s	0.59 s	0.87 s	0.26 s	0.91 s	1.23 s
Round 3	0.16 s	0.55 s	0.91 s	0.22 s	0.69 s	1.27 s
Round 4	0.16 s	0.57 s	0.92 s	0.27 s	0.85 s	1.23 s
Round 5	0.15 s	0.54 s	0.89 s	0.21 s	0.82 s	1.24 s
Round 6	0.15 s	0.56 s	0.89 s	0.22 s	0.84 s	1.23 s
Round 7	0.14 s	0.59 s	1.13 s	0.21 s	0.74 s	1.25 s
Round 8	0.14 s	0.59 s	0.91 s	0.21 s	0.78 s	1.23 s
Round 9	0.14 s	0.48 s	0.89 s	0.23 s	0.68 s	1.57 s
Round 10	0.14 s	0.54 s	0.93 s	0.21 s	0.76 s	1.87 s

Table 7.2: Measured data from gRPC performance evaluation

			Median			Average		
	Request	Parallel	Setup A	Setup B	Ratio	Setup A	Setup B	Ratio
HTTP	100	1	0.53 s	0.68 s	128%	0.56 s	0.70 s	126%
	1000	10	3.13 s	3.85 s	123%	3.12 s	3.88 s	124%
	2000	20	5.83 s	7.02 s	120%	5.83 s	7.04 s	121%
gRPC	100	1	0.15 s	0.22 s	149%	0.15 s	0.23 s	153%
	1000	10	0.57 s	0.80 s	141%	0.56 s	0.79 s	142%
	2000	20	0.90 s	1.24 s	138%	0.92 s	1.34 s	145%

Table 7.3: Aggregated data from performance evaluation

Table 7.1 and Table 7.2 show the measured times of all 10 runs of the performance test. The aggregated values (median and average) are listed in Table 7.3. For the REST/HTTP API the data shows an increase of processing time of about 25% and for gRPC an increase of about 50%.

7.2.3 Maintainability

The first setup has the risk of duplicated code, as functionality that exists in both stacks before the business logic often has to be implemented differently, e.g., code that accesses HTTP headers and gRPC metadata. The second setup reduces that risk by differentiating between the different APIs as late as possible at the adapters.

7.2.4 Operational Complexity

The first setup is easier to deploy as the application logic (i.e., everything without the database) only consists of a self-contained Java application. Setup A is ubiquitous, thus not only understood by most developers, but also supported by many deployment solutions (e.g., PaaS like Cloud Foundry [Clo18a]). The second setup adds another component to deploy, a Node.js application. The additional component is an extra process on a different platform. The effort of deploying setup

B using docker is manageable, as described in Section 7.1.1. The deployment using Cloud Foundry, as described in Section 7.1.4, needs some additional effort as it contains customization to the unique needs of the setup.

7.2.5 Understandability

Through distribution of a system, it becomes harder to understand [New15]. Setup B adds another level of distribution that adds to this complexity. The impact of this complexity can be handled by only using generalizing and stable code in the gateway, meaning if the code in the gateway is used by many services it can be expected to have a generalizing behavior and can be expected to work as expected.

7.2.6 Consistency and Customization

Setup A has independent stacks of functionality for the APIs while Setup B has the same, but translates the protocol at the end. A change in the gRPC API would not directly influence the REST API in setup A, but in setup B it would. If the REST and gRPC API are different technical representations of the same API, then they should behave as similar as possible, and the consistency behavior of setup B may be preferred as it easily keeps the two consistent. However if the APIs are not intended to be consistent, but provide two customized APIs for the different technologies, then in setup A the APIs can be customized indivually, but in setup B it depends on the degree of customization possible in the adapters. The intermediaries in setup A generally can not differentiate between the two APIs and thus can not provide a customized behavior. Through the independence in setup A the two APIs developed independently from each other.

7.2.7 Platform Independence

If the logging components of setup A were to be provided as a library, they could be reused on all services that use the same platform. The intermediary in setup B can also only be used with the Gateframe, but the Gateframe itself can be deployed in front of any service that provides a gRPC API, independently of their used platform.

7.2.8 Integration with Platform

In setup B the common service functionality resides in another process than the service itself. The only means of communication is through the gRPC API itself, e.g., through the use of metadata. This separation restricts the functionality that can be integrated into intermediaries. For example a servlet filter [Ora11] can set the context for Spring security [Piv18b], to allow complex authorization scenarios like checking that the authenticated subject is the owner of a business object. This integration is possible through common service functionality in setup A.

7.2.9 Reusability of API Bricks

In both setups, the common service functionality can be shared and reused. Setup B contains a greater potential for actual reuse of API bricks as it provides a dedicated platform for API bricks. Also, it is platform independent and API bricks can be used by services from different platforms increasing the potentials for reuse.

7.2.10 Separation of Concerns

As setup B consists of two separate parts, it divides the business logic and the technical API logic. This technical separation provides a clearly defined barrier for separation of concerns.

7.2.11 Individual Scaling

A potential advantage of setup B is its ability to scale API and business logic individually. The benefits from this advantage depend on the resource footprint of both parts. If both parts use a significant amount of resources, individual scaling can significantly increase resource utilization.

7.2.12 Summary

We summarize the previously discussed findings by listing advantages and disadvantages: A gateway in front of the service providing API translation and common service functionality has the following advantages:

- Easier maintenance, as common service functionality is shared for API technologies.
- APIs of different technologies are kept consistent.
- The functionality in the gateway is platform independent.
- The API bricks can be reused for various types of services.
- Clear separation of concerns.
- Individual scaling of business logic and API logic.

The service proxy approach has the following disadvantages:

- The service proxy adds a performance overhead.
- Additional component to deploy.
- Increased complexity through distribution.
- Different APIs are hard to customize.
- Functionality in gateway cannot integrate with the platform of the service.

In general, it can be said that while the approach promises some advantages, it also comes at the cost of splitting one component into two. In conclusion, it can be said, that the decision for such an approach depends on the relevance of said advantages and disadvantages in the concrete context.

8 Conclusion and Future Work

In this work, we looked at how the implementation of microservices, especially in the domain of connected vehicles, can benefit from the automatic translation of the service Application Programming Interfaces (APIs) by a service proxy that also can provide common service functionality. To answer this question we designed and implemented a framework that allows the development of gateways using API bricks that can be composed flexibly. We then implemented some of those API bricks necessary to evaluate the approach compared to a classical approach of a microservice that implements the functionality itself.

In Chapter 3 we looked at different kinds of API operations, like request-response, distribution of events, or asynchronous requests. Through examining how these kinds of operations can be implemented using various styles and technologies. For example, event distribution is convenient in messaging through the concept of publish-subscribe, while client-server styles like REST have to rely on polling. We concluded that the similarity of the hard translation between different approaches of the same kind is the introduction of a persistent state into the translation component.

In Chapter 4 we defined three types of API translations to structure API translation in general and bring the challenges from the API operations chapter into a more comprehensible form. Protocol mapping describes the simplest form of API translation to another API which uses a similar protocol and allows to pass the structure of the API. An example of this type of translation is an HTTP/2 to HTTP/1.1 proxy. The translation type called protocol mapping describes the possibility to provide access to an API through modeling the protocol used by the API. A good example is a Representational State Transfer (REST) API that provides access to Advanced Message Queuing Protocol (AMQP) by modeling its elements like messages, queues, topics, and subscriptions. The last kind that we call API remodeling puts only the restriction on a translator that the translated API must provide the same business functionality as the original. Other than that the translator can completely restructure the API. The components providing this kind of translation are hard to automate and typically introduce state for the component to persist. Therefore this type while providing flexibility also introduces complexity.

In Chapter 5 we presented the concept and the implementation of our framework for building gateway components that provide API translation and additional common service functionality. The framework, which we call the Gateframe, uses API bricks as reusable building blocks to use translation components together with components that provide additional functionality. These API bricks can be combined flexibly like Lego bricks by using gRPC as a common communication protocol which leads to a generic interface that also contains detailed information through the interface definition language of gRPC. We defined three categories of API bricks: Adapters that provide an external API, Intermediaries that provide common service functionality and Connectors that consume external APIs. The Gateframe is intended to be very flexible; it is intended to be deployed as an API gateway at the edge to hide distribution of a microservice architecture as well as a service proxy that is deployed directly in front of a service to provide supporting features to

ease development of individual services. Our implementation of the Gateframe uses the Node.js platform [Nod18b]. The API bricks are provided as plugins contained in NPM packages [Npm18], to allow easy sharing of using public or private registries.

In Chapter 6 we took a closer look at a number of selected API bricks that provide functionalities that are good references for building microservices using the Gateframe. The REST adapter adds the ability to provide REST APIs and allows the integration REST based microservice environments. The GraphQL adapter provides the ability to add a declarative API to a service with minimal configuration. The intermediaries offer common service functionality like monitoring, resiliency (Circuit Breaker), authentication or authorization. By extracting these functionalities, the services can provide these functionalities, but the developers of the services do not have to implement them.

In Chapter 5 we looked at the benefits and the challenges of using the Gateframe as a service proxy to provide multiple APIs for a service and to extract common service functionality into reusable API bricks. We created an example application that provides a referential setup of how a service with a service proxy can look. We evaluated the effort of integrating this setup into an existing system by refactoring an existing microservice, which resides in the domain of connected vehicles, to gRPC and evaluated a possible deployment option in Cloud Foundry [Clo18b], the platform where microservices of the existing setup are deployed. Finally, we compared two setups, with and without the Gateframe, by looking at the performance and properties that they hold. From this, we conclude that the Gateframe as a service proxy provides a platform to share common service functionality between services to allow reuse of code even for polyglot service. But this setup also adds a significant cost to the complexity in development and operations, and therefore this cost has to compensated by the benefits of reuse. We think that the benefits of the Gateframe as service proxy are relevant in environments with many microservices using different platforms.

The challenges for microservices in the domain of connected vehicles originate from a great amount of data are produced and the number of use cases that are envisioned for transportation in the Internet of Things. This can result in systems with numerous microservices in which the Gateframe as a service proxy can provide real benefits by reducing the development effort of individual services.

What is still missing in the evaluation of this approach is its use in real scenarios. Only by using it in development and operation of real-world applications the impact of using a gateway component for API translation and common service functionality can be shown.

Future Work

The architecture of the Gateframe is very unbiased in its intended use, leaving room for many applications scenarios and extensions that could be examined in future work. Future work could look at different deployment scenarios, e.g., using the Gateframe as a platform to build services themselves, or deeper examine API remodeling using the Gateframe, which is out of the scope of this work.

Gateframe as Service Mesh

The task of service meshes is to hide the network complexity from services while calling network dependencies [Mor17]. Typically they operate on a network layer but add additional functionality for higher-level protocols like HTTP. The supporting functionality of the service mesh is similar to the API bricks of the Gateframe. Future work could examine what needs to be done to deploy the Gateframe as a service mesh. Potentially such work could add lower level network support to the Gateframe and try creating API bricks on lower levels.

Open Issues on API Translation

In this work, we looked at the challenges of API translation and how kinds of operations can be translated. Our framework is intended to support various translations, but the API bricks we looked at fall in the category of protocol mapping. The translation type of API remodeling could be examined in future work. The challenges here that should be looked at are how the remodeling of APIs can be automated and how the complex behavior of the translation component can be kept understandable to service developers.

A topic that gets increasingly popular is Function as a Service (FaaS) [Rob16]. This paradigm is often called 'serverless' as the notion of a server is hidden from a developer. Instead, a developer defines an event in which a program is executed. As the underlying infrastructure is completely automated, these programs than to be very simple. The architecture of the Gateframe seems to fit for FaaS as it can help to write applications with very few lines of code. For example, the Gateframe itself could be directly used for writing these functions, as Node.js is a popular choice for FaaS (Google Cloud Functions only provides support for Node.js [Goo18e]).

Improvement of the Gateframe

There is a list of improvements, additional features, or alternatives for the Gateframe:

- For the Gateframe to be more suitable as an API gateway, it should be integrated with service discovery. While basic discovery is already integrated through the use of gRPC and gRPC used DNS based discovery, more sophisticated service registries could be integrated into the Gateframe. This could be done by each service implementing it itself or by the Gateframe providing service discovery.
- The Gateframe currently is intended to be used for incoming connections, but the feasibility of the approach for calling other services seems also very promising. The architecture of the framework is unbiased and can be used to build a service mesh solution. As service meshes also provide common functionality it seems a good fit.
- The performance impact of the Gateframe deployed in front of a service that we measured was not negligible. To further improve the performance the Gateframe could be designed and implemented for a statically typed language that compiles to machine code like Go or C++.
- In this work our focus was on the adapter and intermediaries and less on the connectors of the Gateframe. So the REST connector described in this work could be implemented or a connector for a messaging solution.

- The implementation of the Gateframe in this work uses (multiple) linear pipelines to process data. This is simple and robust model, but for some use cases this is not enough. For example for merging the services provided by two upstream bricks. To support this the structure of the brick instances could be extended from linear pipelines to directed acyclic graphs. This would generally mean that some bricks could have more that one upstream and it has to decide which upstream to call. The configuration of the Gateframe would have to support the configuration of the directed acyclic graphs.
- The current implementation lets bricks implement shared state themselves. While this approach lets each bricks choose the best solution for their needs, implementing state sharing themselves increases the effort for creating bricks. The Gateframe could be extended to provide a simple solution for sharing state between different instances, that allow bricks to implement share variables like counter with reduced effort.

Bibliography

[17] *Issue on cloudfoundry/gorouter: Support HTTP/2*. 2017. URL: https://github.com/cloudfoundry/gorouter/issues/195 (cit. on p. 79).

[All10] S. Allamaraju. *RESTful Web Services Cookbook: solutions for improving scalability and simplicity*. O'Reilly Media, Inc., 2010. ISBN: 978-0596801687 (cit. on pp. 34, 37, 38).

[Ama18] Amazon Web Services Inc. *AWS Elastic Beanstalk*. 2018. URL: https://aws.amazon.com/elasticbeanstalk/ (cit. on p. 29).

[AMQ11] AMQP Working Group. *AMQP v1.0*. 2011. URL: http://www.amqp.org/sites/amqp.org/files/amqp.pdf (cit. on pp. 24, 43).

[Aut] grpc-gateway Authors. *grpc-gateway*. URL: https://github.com/grpc-ecosystem/grpc-gateway (cit. on pp. 28, 65).

[Aut18a] gRPC Authors. *gRPC for Node.js*. 2018. URL: https://github.com/grpc/grpc-node (cit. on p. 59).

[Aut18b] gRPC Authors. *gRPC over HTTP2*. 2018. URL: https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-HTTP2.md (cit. on pp. 35, 49).

[Aut18c] gRPC Authors. *Interface ServerInterceptor*. 2018. URL: https://grpc.io/grpc-java/javadoc/io/grpc/ServerInterceptor.html (cit. on p. 78).

[Aut18d] grpc-gateway Authors. *errors.go*. 2018. URL: https://github.com/grpc-ecosystem/grpc-gateway/blob/master/runtime/errors.go (cit. on p. 42).

[Aut18e] rxjs Authors. *RxJS: Reactive Extensions For JavaScript*. 2018. URL: https://github.com/reactivex/rxjs (cit. on pp. 59, 60).

[BFM05] T. Berners-Lee, R. Fielding, L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. 2005. URL: https://tools.ietf.org/html/rfc3986 (cit. on p. 18).

[BN84] A. D. Birrell, B. J. A. Y. Nelson. "Implementing Remote Procedure Calls." In: *ACM Transactions on Computer Systems (TOCS)* 2.1 (1984), pp. 39–59 (cit. on p. 19).

[BPT15] M. Belshe, R. Peon, M. Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. 2015. URL: https://tools.ietf.org/html/rfc7540 (cit. on pp. 19, 42).

[Bra17] J. Brandhorst. *Chunking large messages with gRPC*. 2017. URL: https://jbrandhorst.com/post/grpc-binary-blob-stream/ (cit. on p. 38).

[Bry17] D. Bryant. *The "Paved Road" PaaS for Microservices at Netflix: Yunong Xiao at QCon NY*. 2017. URL: https://www.infoq.com/news/2017/06/paved-paas-netflix (cit. on pp. 26, 47).

[Cap] T. Capan. *Why The Hell Would I Use Node.js? A Case-by-Case Tutorial*. URL: http://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js (cit. on p. 59).

[Clo18a] Cloud Foundry Inc. *Cloud Foundry buildpack for running multiple buildpacks*. 2018. URL: https://github.com/cloudfoundry/multi-buildpack (cit. on pp. 79, 82).

[Clo18b] Cloud Foundry Inc. *CloudFoundry*. 2018. URL: https://www.cloudfoundry.org/ (cit. on pp. 29, 86).

[D H12] E. D. Hardt. *The OAuth 2.0 Authorization Framework*. 2012. URL: https://tools.ietf.org/html/rfc6749 (cit. on pp. 30, 31).

[Doc18] Docker Inc. *Docker*. 2018. URL: https://www.docker.com/ (cit. on pp. 29, 80).

[Ecm17] Ecma International. *The JSON Data Interchange Syntax*. 2017. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (cit. on p. 20).

[Env18] Envoy Project Authors. *Envoy documentation*. 2018. URL: https://www.envoyproxy.io/docs/envoy/v1.6.0/ (cit. on p. 27).

[Erl08] T. Erl. *Service oriented architecture: principles of service design*. Vol. 1. 2008, p. 573 (cit. on p. 24).

[Eur15] European Commission. *eCall in all new cars from April 2018*. 2015. URL: https://ec.europa.eu/digital-single-market/en/news/ecall-all-new-cars-april-2018 (cit. on p. 17).

[Fac16] Facebook Inc. *GraphQL*. 2016. URL: http://facebook.github.io/graphql/October2016/ (cit. on p. 22).

[Fac17] Facebook Inc. *RFC: GraphQL Subscriptions*. 2017. URL: https://github.com/facebook/graphql/blob/master/rfcs/Subscriptions.md (cit. on p. 37).

[Fie00] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures." In: *Building* 54 (2000), p. 162. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (cit. on p. 18).

[FL14] M. Folwer, J. Lewis. *Microservices*. 2014. URL: https://martinfowler.com/articles/microservices.html (cit. on pp. 25, 26, 28, 47, 63).

[FLR14] C. Fehling, F. Leymann, R. Retter. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer, 2014 (cit. on p. 29).

[Fow10] M. Fowler. *Richardson Maturity Model*. 2010. URL: https://martinfowler.com/articles/richardsonMaturityModel.html (cit. on p. 19).

[FR14a] R. Fielding, J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. 2014. URL: https://tools.ietf.org/html/rfc7231 (cit. on p. 42).

[FR14b] R. Fielding, J. Reschke. *RFC 7230 - 7235*. URL: https://tools.ietf.org/html/rfc723{0-5}. 2014 (cit. on pp. 18, 20).

[GBMP13] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami. "Internet of Things (IoT): A vision, architectural elements, and future directions." In: 1 (2013), p. 1660 (cit. on p. 17).

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "Template Method." In: *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995, pp. 325–330. ISBN: 0-201-63361-2 (cit. on p. 56).

[Go 18] Go Authors. *The Go Programming Language*. 2018. URL: https://golang.org/ (cit. on p. 80).

[Goo17a] Google Inc. *auth.proto*. 2017. URL: https://github.com/googleapis/googleapis/blob/75c3a512bce1ac7c2e9a1dd8b2c38ac3f1f5697c/google/api/auth.proto (cit. on p. 71).

[Goo17b] Google Inc. *DocumentationRule*. 2017. URL: https://cloud.google.com/service-management/reference/rpc/google.api#documentationrule (cit. on p. 55).

[Goo17c] Google Inc. *HttpRule*. 2017. URL: https://cloud.google.com/service-management/reference/rpc/google.api#google.api.Http (cit. on pp. 55, 63, 64, 66).

[Goo17d] Google Inc. *Operations*. 2017. URL: https://github.com/googleapis/googleapis/blob/master/google/longrunning/operations.proto (cit. on p. 34).

[Goo17e] Google Inc. *Proto definition of Service from Google API*. 2017. URL: https://github.com/googleapis/googleapis/blob/9f5a5eca1e482b8cc4f534e7c2e4ff27634cc6ee/google/api/service.proto (cit. on p. 54).

[Goo17f] Google Inc. *Protocol Buffers Version 3 Language Specification*. 2017. URL: https://developers.google.com/protocol-buffers/docs/reference/proto3-spec (cit. on pp. 20, 21, 27, 54, 65).

[Goo18a] Google Inc. *Extensible Service Proxy*. 2018. URL: https://github.com/cloudendpoints/esp (cit. on pp. 27, 65).

[Goo18b] Google Inc. *Google App Engine*. 2018. URL: https://cloud.google.com/appengine (cit. on p. 29).

[Goo18c] Google Inc. *Transcoding HTTP/JSON to gRPC*. 2018. URL: https://cloud.google.com/endpoints/docs/grpc/transcoding (cit. on p. 21).

[Goo18d] Google Inc. *Visibility*. 2018. URL: https://cloud.google.com/service-management/reference/rpc/google.api#visibility (cit. on p. 67).

[Goo18e] Google Inc. "Writing Cloud Functions." In: (2018). URL: https://cloud.google.com/functions/docs/writing/ (cit. on p. 87).

[Han14] J. Hanson. *What is HTTP Long Polling?* 2014. URL: https://www.pubnub.com/blog/2014-12-01-http-long-polling/ (cit. on p. 36).

[HW04] G. Hohpe, B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004. ISBN: 9780133065107 (cit. on pp. 23–25, 33, 34, 37, 39, 41).

[Inc16] G. Inc. *Wrappers for primitive (non-message) types*. 2016. URL: https://github.com/google/protobuf/blob/master/src/google/protobuf/wrappers.proto (cit. on p. 21).

[Ind17] K. Indrasiri. *Service Mesh for Microservices*. 2017. URL: https://medium.com/microservices-in-practice/service-mesh-for-microservices-2953109a3c9a (cit. on pp. 27, 47).

[JBS15] M. Jones, J. Bradley, N. Sakimura. *JSON Web Token (JWT)*. 2015. URL: https://tools.ietf.org/html/rfc7519 (cit. on p. 31).

[LCZ+14] N. Lu, N. Cheng, N. Zhang, X. Shen, J. W. Mark. "Connected vehicles: Solutions and challenges." In: *IEEE Internet of Things Journal* 1.4 (2014), pp. 289–299 (cit. on p. 17).

[Lei17] A. Leigh. *Securing Java gRPC services with Spring Security*. 2017. URL: https://eng.revinate.com/2017/11/07/grpc-spring-security.html (cit. on p. 78).

[LOG18] LOG-NET Inc. *Spring boot starter for gRPC framework*. 2018. URL: https://github.com/LogNet/grpc-spring-boot-starter (cit. on pp. 76, 77).

[Men07] F. Menge. “Enterprise service bus.” In: *Free and Open Source Software Conference* (2007), pp. 1–6. DOI: 10.1109/ICMSE.2006.313995 (cit. on p. 25).

[MG11] P. Mell, T. Grance. “The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology.” In: *National Institute of Standards and Technology, Information Technology Laboratory* 145 (2011), p. 7. URL: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf (cit. on p. 29).

[Mic15] Microsoft Inc. *Service Bus Runtime REST*. 2015. URL: https://docs.microsoft.com/en-us/rest/api/servicebus/service-bus-runtime-rest (cit. on p. 43).

[Mic17] Microsoft Inc. *Sidecar pattern*. 2017. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar (cit. on p. 26).

[Mic18a] Microsoft Inc. *Azure Service Bus*. 2018. URL: https://azure.microsoft.com/en-us/services/service-bus/ (cit. on p. 43).

[Mic18b] Microsoft Inc. *TypeScript*. 2018. URL: https://www.typescriptlang.org/ (cit. on p. 58).

[Mon18] MongoDB Inc. *MongoDB*. 2018. URL: https://www.mongodb.com/ (cit. on pp. 76, 80).

[Mor17] W. Morgan. *What’s a service mesh? And why do I need one?* 2017. URL: https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/ (cit. on pp. 27, 87).

[Mul18] MuleSoft Inc. *What is an API? (Application Programming Interface)*. 2018. URL: https://www.mulesoft.com/resources/api/what-is-an-api (cit. on p. 17).

[Net17] Netflix Inc. *Circuit Breaker*. 2017. URL: https://github.com/Netflix/Hystrix/wiki/How-it-Works#circuit-breaker (cit. on p. 73).

[New15] S. Newman. *Building Microservices*. O’Reilly Media, Inc., 2015. ISBN: 978-1-4919-5035-7 (cit. on pp. 25, 35, 37, 72, 83).

[NGI15] NGINX Inc. *HTTP/2 for Web Application Developers*. 2015. URL: https://cdn.wp.nginx.com/wp-content/uploads/2015/09/NGINX_HTTP2_White_Paper_v4.pdf (cit. on p. 41).

[Nod18a] Node.js Foundation. *Express*. 2018. URL: https://expressjs.com/ (cit. on pp. 64, 73).

[Nod18b] Node.js Foundation. *Node.js*. 2018. URL: https://nodejs.org (cit. on pp. 26, 58, 86).

[Npm18] Npm Inc. *NPM*. 2018. URL: https://www.npmjs.com/ (cit. on pp. 59, 86).

[Nyb18] S. Nyberg. *Prometheus client for node.js*. 2018. URL: https://github.com/siimon/prom-client (cit. on p. 73).

[Ohm18] Ohm Authors. *Ohm*. 2018. URL: https://github.com/harc/ohm (cit. on p. 64).

[Ope18a] OpenAPI Initiative. *The OpenAPI Specification*. 2018. URL: https://github.com/OAI/OpenAPI-Specification (cit. on pp. 19, 63).

[Ope18b] OpenID Foundation. *OpenID Connect FAQ and Q&As*. 2018. URL: http://openid.net/connect/faq/ (cit. on p. 70).

[Ora11] Oracle Corporation. *Interface Filter*. 2011. URL: https://docs.oracle.com/javaee/6/api/javax/servlet/Filter.html (cit. on pp. 77, 83).

[Par17] E. Paraschiv. *REST Pagination in Spring*. 2017. URL: http://www.baeldung.com/rest-api-pagination-in-spring (cit. on p. 37).

[Piv18a] Pivotal Software. *Spring Boot*. 2018. URL: https://projects.spring.io/spring-boot/ (cit. on pp. 76, 77).

[Piv18b] Pivotal Software. *Spring Security*. 2018. URL: https://projects.spring.io/spring-security/ (cit. on pp. 78, 83).

[Pro18] Prometheus Authors. *Prometheus*. 2018. URL: https://prometheus.io/ (cit. on p. 73).

[Rea18] ReactiveX Authors. *Do*. 2018. URL: http://reactivex.io/documentation/operators/do.html (cit. on p. 61).

[Ric17] C. Richardson. *Pattern: API Gateway / Backend for Front-End*. 2017. URL: http://microservices.io/patterns/apigateway.html (cit. on pp. 26, 27).

[Rob16] M. Roberts. *Serverless Architectures*. 2016. URL: https://martinfowler.com/articles/serverless.html (cit. on p. 87).

[Sal18] Saleforce.com Inc. *Heroku*. 2018. URL: https://www.heroku.com/ (cit. on p. 29).

[SBJ+14] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, C. Mortimore. *OpenID Connect Core 1.0*. 2014. URL: http://openid.net/specs/openid-connect-core-1_0.html (cit. on pp. 30, 31).

[SBJJ14] N. Sakimura, J. Bradley, M. Jones, E. Jay. *OpenID Connect Discovery 1.0*. 2014. URL: https://openid.net/specs/openid-connect-discovery-1_0.html (cit. on p. 71).

[SHLP05] M.-T. Schmidt, B. Hutchison, P. Lambros, R. Phippen. “The Enterprise Service Bus: Making service-oriented architecture real.” In: *IBM Systems Journal* 44.4 (2005), pp. 781–797. URL: http://ieeexplore.ieee.org/document/5386706/ (cit. on pp. 24, 25).

[Sma18] SmartBear Software. *Callbacks*. 2018. URL: https://swagger.io/docs/specification/callbacks/ (cit. on pp. 35, 36).

[Spe17] S. Speth. “Entwicklung von Microservices mit zusammensetzbaren API-Bausteinen.” In: 358 (2017) (cit. on p. 32).

[TD15] H. L. Truong, S. Dustdar. “Principles for engineering IoT cloud systems.” In: *IEEE Cloud Computing* 2.2 (2015), pp. 68–76 (cit. on p. 17).

[The18] The gRPC Authors. *gRPC*. 2018. URL: https://grpc.io (cit. on pp. 19, 35, 49).

[Tho18] ThoughtWorks Inc. *Overambitious API gateways*. 2018. URL: https://www.thoughtworks.com/radar/platforms/overambitious-api-gateways (cit. on pp. 47, 48).

[Vir17] H. Virdó. *What Is Immutable Infrastructure?* 2017. URL: https://www.digitalocean.com/community/tutorials/what-is-immutable-infrastructure (cit. on p. 51).

[Wal07] C. Waldspurger. *MIT IAP Course Lecture# 1: Virtualization 101*. 2007 (cit. on p. 29).

[Wet17] J. Wettinger. "any2api: the better way to create awesome APIs." In: (2017). URL: https://github.com/any2api/any2api/blob/master/README.md (cit. on p. 32).

[Wir18] D. Wirtz. *protobuf.js*. 2018. URL: https://github.com/dcodeIO/ProtoBuf.js/ (cit. on pp. 59, 65).

[WWWK96] J. Waldo, G. Wyant, A. Wollrath, S. Kendall. "A note on distributed computing." In: *International Workshop on Mobile Object Systems*. Springer. 1996, pp. 49–64 (cit. on p. 20).

[Xia17] Y. Xiao. *Paved PaaS to Microservices*. 2017. URL: https://www.slideshare.net/yunongx/paved-paas-to-microservices (cit. on pp. 26, 47).

All links were last followed on May 1, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature