University of Applied Sciences Würzburg-Schweinfurt
Faculty Of Computer Science and Business Information Systems

Bachelor Thesis

Design and Implementation of a Node.js Framework for the Development of RESTful APIs

submitted to the University of Applied Sciences Würzburg-Schweinfurt in the Faculty Computer Science and Business Information Systems to achieve the Bachelor of Engineering degree in "Computer Science"

N. N.

Submitted on: July 18, 2021

First Reader: Prof. Dr. Peter Braun
Second Reader: Prof. Dr. Rolf Schillinger

Zusammenfassung

In dieser Arbeit wird ein Framework-basierter Ansatz vorgeschlagen, um es Entwicklern zu ermöglichen, effizient REST APIs zu entwickeln. Vielmehr noch soll sichergestellt werden, dass die REST Anforderungen eingehalten werden, indem es der neue Ansatz schwieriger für die Entwickler macht die Anforderungen zu missachten als sie einzuhalten.
Aktuell liegt die Problematik in der Komplexität der REST-Thematik und der draus resultierenden Schwierigkeit das Thema vollumfänglich zu verstehen. Dies führt dazu, dass einigen Entwicklern entweder unmöglich ist, REST-konforme APIs zu implementieren oder dies mit einem großen Zeitaufwand und einer hohen Fehleranfälligkeit verbunden ist. Ursächlich dafür werden unter anderem sowohl der abstrakte Architekturstil REST, dessen Verständnis dem durchschnittlichen Entwickler schwer fällt [19, p. 343], als auch das Fehlen passender Tools, die die Entwickler bei der Umsetzung der APIs unterstützen [20], gesehen. Um diesen Umstand zu verdeutlichen, wird eine Analyse aktuell vorhandener Frameworks präsentiert, wobei diese auf deren Möglichkeiten untersucht werden, Entwickler bei der Implementierung von REST APIs zu unterstützen.
Das in dieser Arbeit entwickelte Framework wirkt der beschriebenen Problematik entgegen und unterstützt die Entwickler bei der effizienten Entwicklung, indem es Klassen Templates für die Verarbeitung von HTTP Anfragen der vier CRUD Methoden bietet. Diese abstrakten Basisklassen definieren einen festen Ablauf von Verarbeitungsschritten und stellen dabei auch Funktionalitäten zur Verfügung, die bei jeder Verarbeitung einer Anfrage benötigt werden. Dies ermöglicht eine effiziente Entwicklung, da sich wiederholende Aufgaben, wie beispielsweise die Link-Generierung bei der Paginierung, bereits von den abstrakten Klassen übernommen werden. Zudem wird durch den festgelegten Verarbeitungsprozess die REST-konforme Entwicklung gesichert. Dass dieser sehr strikte Ansatz tatsählich zu einer REST-konformeren Entwicklung führt, wird anschließend anhand einer empirischen Studie gezeigt. Deren Ergebnisse werden abschließend präsentiert und in diesem Zusammenhang neu aufgetretene Fragestellungen diskutiert.

Abstract

In this work, a framework-based approach is proposed to enable developers to efficiently develop REST APIs. Further, to ensure that REST requirements are followed, the new approach makes it more difficult for developers to violate the constraints than to comply with them.
Currently, the problem lies in the complexity of REST and the resulting difficulty to fully understand it. As a result, some developers are failing to implement REST-compliant APIs or consider it time-consuming and error-prone to do so. The reasons for this include both the abstract architectural style of REST, which is difficult for the average developer to understand [19, p. 343] and the lack of appropriate tools to support developers in implementing the APIs [20]. To demonstrate this fact, an analysis of currently available frameworks is presented, examining their capabilities to support developers in implementing REST APIs.
The framework developed in this thesis addresses the described issue and supports developers in efficient development by providing class templates for processing HTTP requests from the four CRUD methods. These abstract base classes define a fixed sequence of processing steps, while also providing functionality that is required each time a request is processed. This enables efficient development, since repetitive tasks, such as link generation during pagination, are already handled by the abstract classes. In addition, the defined process ensures REST compliant development. The fact that this very strict approach actually leads to REST-compliant development is then shown by an empirical study. Finally, the results of this study is presented and new questions that have arisen in this context will be discussed.

Acknowledgement

A special thanks goes to my supervisor Prof. Dr. Peter Braun, who guided me through this work and always took the time to answer all my questions about REST.
I would also like to thank Sophia Dürrnagel for the professional realization of my idea for the graphic in the introduction.
Finally, I would like to thank Moritz Reis for helping me with our discussions about the survey design.

Contents

1. Introduction

Like Meinel et al. [15, p. 22] describe it, whenever a client, for example with the help of a browser, requests a document from a server, some form of interaction must take place between the two. Thereby, this communication is based on the so-called **Hypertext Transport Protocol** (HTTP). This protocol has a rather long history, whose description in the following is based on Meinel et al. [15, p. 739 ff]. The very first version *0.9* was developed at CERN in 1989/1999. This first version initially had no version number. This was later changed to distinguish this first very rudimentary version from the much more mature version *1.0*. This advanced version was necessary because the original version *0.9* had a couple of problems. For example, it was not possible to transfer data from the client to the server. Already introduced in 1992, this version was, however, not specified until 1996 in the **Request for Comments** (RFC) 1945. Although this version helped the World Wide Web achieve a breakthrough, there was still a need for improvements in some aspects. These include inadequate caching strategies. For this reason, it was necessary to specify a new version, which addresses these issues. The result is version *1.1* of HTTP, which was specified in RFC 2616 in June 1999. Besides the improvement of the caching strategies, further functionalities, like content negotiation, are added to the protocol. As Tilkov et al. [22, p. 9-10] points out, Roy Fielding has been involved in the standardization of web protocols since 1994. He used his self-developed HTTP Object Model as a framework for the design of HTTP versions *1.0* and especially *1.1*. With this model, he wanted to define a unified concept for static content and dynamically computed information, which together forms a global information system. In his famous doctoral thesis [5], he abstracted the concrete HTTP architecture and developed the abstract architecture style **Representational State Transfer** (REST). Thereby, the **Hypertext Transport Protocol** is a concrete implementation of this abstract architecture style.

1.1. Motivation

A majority of the new publicly available APIs now follow REST principles [19, p. xvii]. Although these HTTP APIs often claim to be RESTful, it is often the case that they violate the REST constraint [22, p. 10]. This is problematic in the sense that one probably gets the most out of an architecture (REST) by using the concrete technology (HTTP) as envisioned by the architect, like Tilkov et al. [22, p. 10] describe it. For

example, REST can significantly reduce the coupling of two systems [22, p. 2-3]. Also, the server can change its underlying implementation without necessarily adapting the client as well [19, p. 350]. But not only APIs violate the REST constraints, but also the underlying frameworks with which the APIs were developed [22, p. 10-11]. Software developers would therefore benefit from a framework that complies with the REST constraints, as the API developed would thus take full advantage of the architecture. In [4, p. 71-99], Doglio gives several reasons why Node.js in particular is well suited for API development. These include the possibility for asynchronous Input/Output as well as the simplicity of Node.js. The fact that Node.js in particular lends itself so well to API development makes it even worse that there is no Node.js framework to help developers take full advantage of the REST architecture.

1.2. Goal of the Thesis

Thus, the goal of this work is to design and implement a Node.js framework for the development of RESTful APIs. By properly applying REST principles, the framework should enable developers to effectively build REST APIs. In addition, the framework should be designed to enforce developers to comply with REST constraints. Overall, this should make it harder for developers to violate the REST constraints than to comply with them. With this approach, even software engineers who have difficulties understanding REST should be able to develop REST-compliant APIs and take full advantage of them.

In summary, the goal is to develop a framework that allows the developed APIs to be REST compliant and thus take advantage of this architecture. Thus, the client and the server, as shown in Figure 1.1, are coupled much more loosely. Furthermore, the server can be modified without the client having to be adapted.

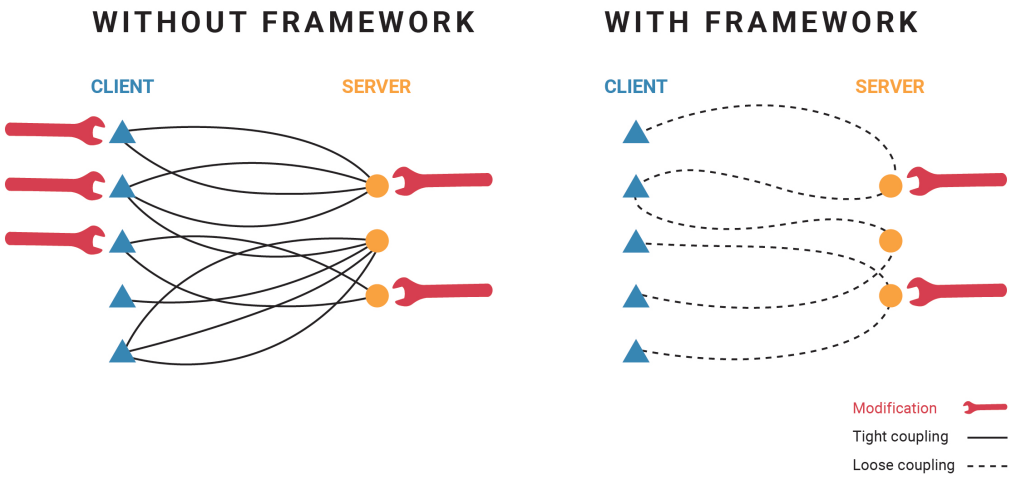


Figure 1.1.: Goal of the thesis interpreted graphically.

1.3. Organization of the Thesis

At the beginning of the thesis, Chapter 2 explains all the required foundations. There will first be a short summary of the five mandatory REST constraints. Thereby a special focus is on the hypermedia constraint. Afterwards the basics of the programming languages Node.js and TypeScript are presented. The TypeScript features that play a central role in this work are given special consideration. Chapter 3 first analyses the problems of REST and specifically of hypermedia. Thereafter, existing Node.js frameworks are analyzed to show the problems of developing REST APIS in Node.js. The solution to the problems, presented in the previous chapter, is covered in Chapter 4. The developed framework is evaluated in Chapter 5. For this purpose, a developer survey is conducted and the results are discussed. Finally, Chapter 6 summarises the results obtained in the previous chapters. In addition, there is an outlook on possible future features of the framework and further questions that could be investigated empirically.

2. Technical Background

We start this chapter with a summary of the REST architectural style, formulated by Roy Fielding in his doctoral thesis [5]. Since the developed framework is based on Node.js and TypeScript, these two technologies will be explained afterwards. In the TypeScript section, besides the general explanation, we will only focus on the points that are particularly important for this work.

2.1. Representational State Transfer (REST)

In the fifth chapter of his thesis [5], Fielding defines architectural constraints that must be followed to comply to the REST architectural style. Thereby, the **Code-On-Demand** constraint is optional and is therefore not considered. For an explanation of it see [5, p. 84-85] or [19, p. 354-355]. In addition, a special focus is on the "hypermedia as the engine of the application state (HATEOAS)" constraint.

2.1.1. Constraints

Before discussing the HATEOAS constraint in more detail, the other constraints are explained first.

Client-Server

As Fielding [5, p. 45] describes, a server provides an arbitrary number of services, that can be requested by clients. If a server now receives a request, it is able to decide to either reject the request or to process it and to send back a response to the client. This restriction leads to three advantages, which Fielding also explains in [5, p. 78]. First, the user interface, consuming the services provided by the server, can be made available on more than one platform. Second, scalability is increased because the server can be kept more simplified by concentrating on its services and not on the user interface. The last and most important benefit is the situation, that the client and the server can be developed independently. Overall this constraint should be very familiar since

the client-server network architecture is the most dominant one on the internet [19, p. 351].

Stateless

The Stateless constraint restricts the communication in the way, "[...] that each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server" [5, p. 78-79]. This does not mean that the server is stateless but that the state must be held by the client or transformed in a resource state by the server [22, p. 18]. Among other advantages, this constraint again enhances scalability since the server need not store any state information between requests [5, p. 79]. In combination with the self-descriptive message constraint the stateless constraint makes caching first ever possible [19, p. 352].

Cache

"Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable" [5, p. 79]. If a response has been set as cacheable, caches are now allowed to reuse this response to respond to any request, that is equivalent and would result in an identical response [5, p. 48]. To enable caches to decide if requests are identical, the two constraints "Stateless" and "self-descriptive message" are essential and make caching possible [19, p. 352]. Thereby, caching brings the advantage to "partially or completely eliminate some interactions, improving efficiency, scalability, and user-perceived performance [...]" [5, p. 80]. In practice, it is possible to delegate the caching to the infrastructure by using the Cache-Control header (see RFC 7234 [7]), instead of implementing a use-case specific cache [22, p. 138].

Uniform Interface

"The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components" [5, p. 81]. To obtain this uniform interface, Fielding defines the four constraints **identification of resources**, **manipulation of resources through representations**, **self-descriptive messages** and **hypermedia as the engine of application state (HATEOAS)** [5, p. 82].

Richardson et al. [19, p. 347-350] provide a short and good overview of these four constraints. The following explanation of constraints is based on this overview. They call the first constraint the "addressability" one. "A URI identifies a resource. The resource's state may change, but its URI stays the same" [19, p. 348]. The next constraint is **manipulation of resources through representations**. Since resources can not be

transferred over the network, the client has to interact with the server via representations of the resources. Hence, the clients and servers send representations of resources to manipulate (read, create, etc.) these. The **self-descriptive messages** one simply states, that each message must contain everything needed so that the recipient can understand it. The **HATEOAS** constraint is explained in more detail in subsection 2.1.2.

One advantage of the uniform interface is the independent evolvability since the implementations are decoupled from the provided services [5, p. 81-82]. However, through the uniform interface, "information is transferred in a standardized form rather that one which is specific to an application's needs" [5, p. 82]. This circumstance degrades the efficiency [5, p. 82].

Layered System

"The "layered system" constraint is less about proxies and gateways, and more about the fact that adding one between client and server is a nearly transparent operation. A client doesn't know whether it's talking directly to a server, or whether it's talking to a proxy that talks to a proxy that talks to a gateway" [19, p. 353]. So the main idea of this constraint is that proxies and gateways are invisible [19, p. 354]. An arbitrary number of proxies or gateways can be placed between the client and the server and the client nor the server would even notice [19, p. 354]. In practice, proxies can be used for load balancing and caching [19, p. 354].

2.1.2. Hypermedia

The "Hypermedia as the engine of application state" or according to Richardson et al. [19, p. 360] the "hypermedia constraint" is part of the uniform interface described in section 2.1.1. Richardson et al. [19, p. 360] defines hypermedia as data "sent from the server to the client, which explains what the client can do next" [19, p. 360]. But hypermedia has not only the function to define the next possible actions for the client but also to link different resources between each other, as Tilkov et al. [22, p. 13] mentions. Without hypermedia, so linking between resources and proposed actions, the client would not know to which URLs it can make requests and how they have to look like since there are infinitely many URLS [19, p. 45]. As Richardson et al. [19, p. 45] states, Hypermedia driven Web APIs should guide the client by providing a set of possible HTTP requests that the client could make next. Thereby, they note that while the server knows what might happen next, it is the client who finally decides what happens next.

We now consider a well-designed web application as a state-machine as proposed by Fielding [5, p. 109]. For this purpose we model a REST API as a deterministic finite state machine (DFA). A DFA consists of the 5-tuple [13, p. 204]:

- finite set of states S

- finite input alphabet $\sum$
- state transition function $\delta : S \times \sum \rightarrow S$
- set of final states $E \subseteq S$
- start state $s_0 \in S$

We apply this now to the elements of a REST API. Thereby a *State* "is a pair of one HTTP method and one resource and represents one valid REST request, which uses a URI to access one resource" [20]. The state transition function δ is modeled by the hyperlinks [20], included in the representation of the current resource. The initial state s_0 is represented by the initial URI, as this is one of the few details that the client needs to know from the beginning [6]. We equate the set of final states E with the set of state S so $E = S$, since the client can break the usage of the application at any time. The alphabet $\sum$ consists of events, representing the selection of a provided hyperlink by the client and thus triggering a state transition as proposed by Fielding [5, p. 109].

If an API does not take the hypermedia constraint into account this would mean, it would define a DFA without a state transition function. As a result, the alphabet could not be processed or rather no link selection could take place, because there are just no hyperlinks the client could select from. So there is no formal correct DFA. This also means that the API cannot be RESTful, or as Fielding points out "if the engine of application state [...] is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API" [6].

Tilkov et al. [22, p. 77] noted several advantages of following the hypermedia constraint. One of them is the looser coupling of client and server. As the server provides a lot of information at runtime, the client does not need to be updated even if the server has changed. Thereby one gets extensibility because the server can "change its underlying implementation without breaking all of its clients" [19, p. 350]. Another advantage is that the server completely controls the application flow by only providing links for actions that are currently possible. The server can thereby control what the client is or is not able to do next. The client can then adapt this and show the end-users only the actions they are allowed to do.

2.2. Node.js

Node.js[1] is a JavaScript runtime that executes JavaScript code with the V8[2] JavaScript engine. The V8 is developed by Google and written in C++. It is for example used to run JavaScript in the Google Chrome browser. In contrast to this, Node.js uses the V8 engine to execute JavaScript on the server-side. In the following, it will be explained

[1] https://nodejs.org/
[2] https://v8.dev/

what makes Node.js different from other environments and then the most important aspects of Node.js will be presented.

As Casciaro [1, p. 24] mentions, Input/Output (I/O) operations are very slow and therefore expensive. Tilkov et al. describe in [21] how Node.js handles these expensive I/O operations. This description serves as the basis for the following explanation of Node.js. While other environments often rely on multithreading, Node.js relies on a single-threaded asynchronous I/O eventing model. Thereby, the asynchronous I/O operation does not block the general execution flow while waiting for the result but notifies the Node.js environment via an event when the result is available. This event is registered by Node.js and the result of the asynchronous task can be processed as soon as the event occurs. In a multi-threading environment, the thread would be blocked while waiting for the result.

Tilkov et al. also mention that programming with Node.js is more efficient and scalable compared with programming in a multithreading environment. Multithreading can be very error-prone, as problems such as deadlocks or parallel access to the same resource can occur. See [21] for a detailed discussion. This asynchronous eventing model is also available in other languages like Python[3] for example. Unlike these, Node.js does not support this on a framework level but a language level. So the implementation details are hidden from the developers. See Pasquali [18] for more details about Node.js.

2.2.1. Asynchronous Programming

In Node.js, asynchronous programming is done with so-called promises. Thus, a function that executes an asynchronous task returns a promise. This promise can then be awaited to receive the result of the asynchronous task. In Listing 2.1 the result of the *fetchFromDatabase* method is awaited via the *await* keyword. Note that this keyword is only available in *async* functions. While the result of this function is awaited and the execution within the *readFromDatabase* method is blocked until the result is available, the execution of other tasks is not blocked. If the function would not be awaited, the execution within the *fetchFromDatabase* would continue and the result would be returned even it is still undefined.

```
1
2   async function readFromDatabase() {
3     const dbResult = await fetchFromDatabase()
4     return dbResult
5   }
6
```

Listing 2.1: Asynchronous functions in JavaScript.

[3] https://docs.python.org/3/library/asyncio.html

2.2.2. Object and Classes

In Node.js it is very simple to create objects. In contrast to very strict object-oriented languages, class definition beforehand. In Listing 2.2 a simple plain object that represents a user object can be seen. In addition, further properties can be added to an object at any time, but can also be deleted again.

```
1
2   const user = {
3     firstName: 'Matthias',
4     age: 23
5   }
6
```

Listing 2.2: Creation of a plain JavaScript object.

In addition to simple plain objects, there are also classes. For these, a class definition must be written beforehand and an instance must be created with the *new* keyword. There are differences not only in the creation of these two but also in their use. With a class, it can be checked at runtime whether an object is an instance of this class if it was created with the *new* keyword. If it is just a plain object, it only has the *object* type. This is especially relevant for the topic in subsection 2.3.4.

```
1
2   class User {
3
4     constructor(firstName, age) {
5       this.firstName = firstName
6       this.age = age
7     }
8   }
9
10  const user = new User('Matthias', 23)
11
```

Listing 2.3: Define a class and initialise an instance of it.

2.2.3. Functions

As Tilkov et al. [21] describes, JavaScript is a functional language and therefore supports higher-order functions. Compared with Java, for example, functions are very flexible in JavaScript.

```
1
2   function add(x) {
3     return function(y) {
4       return x + y
5     }
6   }
7
8   const add10 = add(10)
9
10  add10(20) // === 30
11
```

Listing 2.4: Higher-order functions in JavaScript.

In Listing 2.4 a function is defined that returns a function that adds the first specified number to any number. However, functions can not only be returned but also passed as function parameters. See Listing 2.5 for an example. There, the provided function is called and the returned string is changed to upper case.

```
1
2   function myFunction(fn) {
3     return fn().toUpperCase()
4   }
5
6   myFunction(function () {
7     return 'hello, world'
8   }) // === 'HELLO, WORLD'
9
```

Listing 2.5: Functions as parameters.

JavaScript has so-called *arrow functions*, that is an alternative function syntax. It is especially useful while providing functions as parameters. In Listing 2.6 the function from the example before is rewritten with arrow functions.

```
1
2   const myFunction = (fn) => fn().toUpperCase()
3
4   myFunction(() => 'hello, world')
5
```

Listing 2.6: Arrow functions in JavaScript.

2.2.4. HTTP Server

Node.js has a built-in HTTP server. See [16] for the API documentation. In fact, Node.js is developed with a focus on HTTp. The issues of streaming and low latency were also

considered in the design of Node.js. For this reason, HTTP frameworks do not have to implement this part themselves but can implement further functionalities based on it. So frameworks like Express[4] are often very lightweight and minimalist.

```
1
2   const http = require('http')
3
4   const server = http.\createServer(function (req, res) {
5     if (req.url === '/myResource' && req.method === 'GET') {
6       res.statusCode = 200
7       res.end()
8     } else {
9       res.statusCode = 404
10      res.end()
11    }
12  })
13
14  server.listen(8080, function () {
15    console.log('Server is listening on port 8080')
16  })
17
18
```

Listing 2.7: Example HTTP server in Node.js.

In Listing 2.7 an example HTTP server with a single endpoint is shown. One can see that only the built-in *http* module is required to create and start an HTTP server. The function in line **4** receives two arguments, the request and the response object and is called every time the server receives an HTTP request. The first argument is used to extract any information from the HTTP request and the response object is to define the HTTP response like status code or response payload. Within the function, the request URL and the request method is checked. If the URL does not match */myResource* and the method is not *GET* a *404 Not Found* response is sent, otherwise the client receives a *200 OK* response. Even though it would be possible to implement a complete application with this built-in HTTP server, a framework is usually used for this, since such frameworks provide the developers further features that ease the development.

In Listing 2.8, one can see the exact same server but implemented with the Express framework. While the implementation of the endpoint is much easier with the framework, it still uses the same method signature for the endpoint registration. This is very common for most HTTP Node.js frameworks. These usually simply extend the request and the response object from the Node.js core with further features. Additionally, the framework usually comes with an HTTP router, so the developers can define endpoints in a scaleable way. In the example above, the router is used in line **6** to register the endpoints with the path */myResource*.

[4]https://github.com/expressjs/express

```
1
2     const express = require('express')
3
4     const server = express()
5
6     server.get('/myResource', function (req, res) {
7       res.sendStatus(200)
8     })
9
10    server.listen(8080, function () {
11      console.log('Server is listening on port 8080')
12    })
13
14
```

Listing 2.8: Example HTTP server with Express.

2.3. TypeScript

TypeScript [23] is a language that is created and maintained by Microsoft and was released in 2012. It is a superset of JavaScript, so valid JavaScript is valid TypeScript [3, p. 1-2], and adds static typing to JavaScript. The TypeScript compiler (TSC) works a little differently compared with languages like Java. Instead of compiling the source code to byte or machine code, the source code is transformed to JavaScript. The generated code can then be run anywhere where normal JavaScript code could be run, no matter whether in a browser or on Node.js. Like Fenton [3, p. xix] mentions, TypeScript was built to provide a possibility to write large JavaScript applications in a scalable way by providing features like compile-time checking and better design-time tooling.

After introducing the basic idea of TypeScript, the most important features of the language for this work are presented below. Every feature introduced in the following is explained in detail in [24]. For even more details see [3] or [11].

2.3.1. Types

As already described above, TypeScript adds types to JavaScript. In contrast to Java, for example, these types are placed behind the variable name. The same applies to methods. See Listing 2.9 for an example. Even though we specify the type of the *myString* variable, this would not be necessary, since the TypeScript compiler can infer the type automatically. Generally, one can say, that the TypeScript compiler tries to infer the types wherever it is possible.

```
1
2   // type is not necassary here
3   const myString: string = 'Hello, World!'
4
5   function addTwo(n: number): number {
6     return n + 2
7   }
8
```

Listing 2.9: Types in TypeScript.

TypeScript comes with multiple built-in types which includes the types **string, number, boolean, Array, any, undefined** and **object**.

```
1
2   const s: string = 'Hello, World!'
3
4   const n: number = 1
5
6   const b: boolean = true
7
8   const a: string[] = ['a', 'b'] // is the same as Array<string>
9
10  const anything: any = 1 // but could be also '1' or true or {}
11
12  const u: undefined = undefined
13
14  const o: object = { a: 1 }
15
```

Listing 2.10: Built-in TypeScript types.

Any type, so built-in and custom ones, can be combined within so-called Union Types. With these multiple types can be combined to create a new type.

```
1
2   function acceptsStringOrNumber(x: number | string): void {
3   }
4
5   // No errors
6   acceptsStringOrNumber(2)
7   acceptsStringOrNumber("2")
8
9   // Type error
10  acceptsStringOrNumber(false)
11
```

Listing 2.11: Union Types in TypeScript.

When working with asynchronous functions, this affects the return type. The actual

type must then be wrapped in a promise. Such functions must be awaited to retrieve the actual return type otherwise Promise type is returned.

```
1
2   function syncFunction(): string {
3     return 'Hello!'
4   }
5
6   async function asyncFunction(): Promise<string> {
7     return 'Hello!'
8   }
9
10  const actualType: string = await asyncFunction()
11
12  const promiseType: Promise<string> = asyncFunction()
13
14
```

Listing 2.12: Promise type in TypeScript.

In TypeScript it is possible to declare method parameter or class attributes as optional via the "?" symbol. Parameters with a default value are automatically optional. See Listing 2.13 for an example.

```
1
2   function add(x: number, y?: number, z = 0): number {
3     if (typeof y === 'undefined') y = x
4
5     return x + y + z
6   }
7
8   // Error
9   add()
10
11  // No errors
12  add(1) // result = 2
13  add(1, 2) // result = 3
14  add(1, 2, 3) // result = 6
15
16
```

Listing 2.13: Optional parameters in TypeScript.

2.3.2. Classes, Interfaces and Enums

TypeScript has support for classes, interfaces and enums just like classic object-oriented programming languages. Just like the **final** keyword in Java the **readonly** keyword can

be used in TypeScript to make class fields only assignable in the constructor. With the keywords **public, protected** and **private** the visibility of class fields can be controlled. Thereby **public** is the default visibility.

```
1
2   enum NumberType {
3     integer,
4     double,
5     complex
6   }
7
8   interface Addable {
9     add(x: number, y: number): number
10  }
11
12  class Calculator implements Addable {
13    add(x: number, y: number): number {
14      return x + y
15    }
16  }
17
18
```

Listing 2.14: Classes, Interfaces and Enums in TypeScript.

These visibility modifiers can be used to shorten the initialization of class fields. In the example Listing 2.15 the initialization process of **s** and **n** are identical.

```
1
2   class MyClass {
3
4     protected readonly n: number
5
6     constructor(protected readonly s: string, n: number) {
7       this.n = n
8     }
9   }
10
11
```

Listing 2.15: Shorten property initialization via visibility modifiers within the constructor.

2.3.3. Generics

Generics work like in languages like Java. The **extends** keyword can be used to constraint the possible type of the generic. As seen in Listing 2.16, one can restrict a generic

parameter to be from type **string** or **number**. Besides classes or interfaces, functions can also be declared with a generic parameter.

```
1
2   class Model<T extends string | number> {
3     readonly id: T
4
5     constructor(id: T) {
6       this.id = id
7     }
8   }
9
10  // Valid
11  const m1 = new Model(1)
12  const m2 = new Model('1')
13
14  // Error
15  const m3 = new Model(false)
16
17  function justReturn<T>(x: T): T {
18    return x
19  }
20
```

Listing 2.16: Generics in TypeScript.

2.3.4. Decorators

Decorators, already known from other languages like Java or Python, are a JavaScript feature, that is still in the proposal stage[5]. TypeScript itself already supports this but as an experimental feature. See [11, p. 397 ff.] and [25, p. 132 ff.] for a detailed explanation. Decorators can be applied to class declarations, methods, accessors, properties and parameters. Thereby they can be used to modify or extend the behaviour of these. In Listing 2.17 the *logClassName* decorator is used to log the name of the annotated class and is executed at runtime. Instead of just logging the class name, one could also modify or even replace the class constructor for example. The possibility to change the behaviour also applies to all the other types of decorators. Decorators are often used to register HTTP endpoints in the application.

Another common use case is decorator factories. These are functions that return a decorator function. With this approach, it is possible to modify the behaviour of the decorator. In the example below, the developer can specify the maximum length of a string parameter. Without a factory, the developer could not specify the maximum length.

[5] https://tc39.es/proposal-decorators/

```
function logClassName(clazz) {
  console.log(clazz.name)
}

@logClassName
class MyClass {
}

new MyClass()
```

Listing 2.17: Example Decorator in TypeScript.

```
// Decorator factory
function maxStringLength(maxLength: number): ParameterDecorator {
  // Actual decorator function
  return function(
  target: object,
  propertyKey: string | symbol,
  parameterIndex: number
  ) {
    // validation logic
  }
}

class MyClass {

  public printString(@maxStringLength(10) s: string): void {
    console.log(s)
  }
}
```

Listing 2.18: Decorator factory.

The last use case of decorators presented is the access of metadata. Decorators are executed at runtime when there is no more type information available. However, TypeScript can be configured to emit metadata about the respective types. Note that this metadata is only available for elements that have been annotated with a decorator. To have access to this kind of metadata the *reflect-metadata* module has to be installed. It grants access to the constructor types, class attribute types, parameter types, accessor types and the return types of methods. In Listing 2.19 the types of the constructor parameter are retrieved. This is used for example to implement a dependency injection container.

```
1
2     import 'reflect-metadata'
3
4     @logClassName
5     class Spaceship {
6
7       constructor(protected s: string, protected n: number) {
8       }
9     }
10
11    // const constructorTypes = [ String, Number ]
12    const constructorTypes = Reflect.getMetadata('design:paramtypes', Spaceship)
13
14
```

Listing 2.19: Retrieve constructor parameter types via metadata.

Note that decorators are only available for classes and not for interfaces or plain objects.

3. Problem Definition

This chapter defines the problems that developers face when implementing a REST API in Node.js. Therefore, we first consider the problems of REST and hypermedia in particular. Afterwards, there is an analysis of currently available Node.js frameworks. Finally, the current situation for developers when implementing REST APIs in Node.js is analysed and summarised.

3.1. The Problems of REST and Hypermedia

Richardson et al. [19, p. xvii] outlines, that the majority of new released public Web APIs follows the REST principle. Nevertheless, according to Schreibmann et al. [20], there are huge differences between the ideas and principles of the REST architectural style and the concrete implementations. This problem is worsened by the fact that REST became a buzzword these days [19, p. xvii] and therefore a lot of APIs are called RESTful, even they are just HTTP based, as Fielding himself criticizes in his blog post [6]. One REST constraint, which is confusing and controversial, is the hypermedia one [19, p. 46]. While the concepts of resources or the naming of URLs are mostly understood, the hypermedia constraint is hard to understand for developers [19, p. xvii]. But as Fielding puts it in his blog entry [6] the hypermedia[1] constraint is mandatory and a violation of it results in a non-RESTful API. In the following paragraphs, the causes of the misunderstanding will be discussed and the consequences for developers will be explained.

One reason is the circumstance, that the thesis from Fielding is very hard to understand for the average developer due to its style and level of abstraction [19, p. 343]. Roy Fielding works in his thesis on a higher level of abstraction by concentrating on concepts instead of the concrete syntax [22, p. 10]. Even though HTTP is the only practical implementation of this architectural style, one could use any underlying protocol to implement the architecture [22, p. 10]. This not only leads, as mentioned above, to a more difficult understanding but also opens this topic for interpretations [20]. So it could easily happen, that software developers just select some principles [20].

While the hypermedia concept seems quite natural when using the World Wide Web, it is still a very difficult to understand when interacting with Web APIs, as Richardson et al. [19, p. xviii] and Tilkov et al. [22, p. 71] mention. There is a difference between a

[1] Fielding used originally the term hypertext. Hypermedia is just the extension of hypertext with graphics, videos and different multimedia formats [22, p. 71].

website and a Web API [19, p. 16]. In one case a human makes the decision which links to follow, in the other, a programmed client [19, p. 16].

Another issue, described by Schreibmann et al. [20], is the lack of tools, which enables the developers to test, design and implement RESTful APIs. Available frameworks allow the developer to interact and provide an HTTP interface, so for example to read or write the HTTP request/ response body. So theoretically they give the developer the ability to implement an API, which is REST compliant, but they do not enforce it.

3.2. Analysis of existing Node.js Frameworks

To illustrate the lack of tools, the following section presents an analysis of several existing Node.js frameworks. For this purpose, a template is created with which all frameworks are analyzed and evaluated. The section begins by explaining why each of the selected aspects of the template is considered.

- **Hypermedia**:

  As discussed in section 3.1, hypermedia is a confusing topic for developers and often the missing part of an API to be REST compliant. Hence the framework must enable the developers to implement a hypermedia strategy more easily and efficiently. Consideration is given to linking between resources, as well as providing possible state transitions in the form of links. However, it does not matter which hypermedia format is supported.

- **Pagination & Filtering**:

  Pagination and Filtering only concerns collection resources, which in turn are also resources [5, p. 88]. Since such a collection could contain millions of single resources, but the server does not want to serve the client a response, containing millions of resources, the server should implement pagination strategies [19, p. 101]. The server does not have to only implement these strategies on the server-side, but must also provide the client links, with the correct relations types, see RFC 5988 [17], to guide the client through the collection as described by Richardson et al. [19, p. 101]. Even though pagination is no REST constraint nor it is mandatory to support it [2, p. 51], it is required to guide the client with the provided links, to obey the HATEOAS principle, as soon as it is implemented.
  The same principle can be applied to filtering, too. If the API let the client filter through the collection to provide only resources, the client is searching for, the server must guide the client. Therefore the server has to provide the client search templates, which can be filled out and then sent back to the server. Richardson et al. [19, p. 99] describes this for the Collection+JSON hypermedia format, but

it can be also applied to any hypermedia strategy or format. For example, URI templates [10] can be used for this purpose.
Although theoretically both, pagination and filtering, are not necessary to build a RESTful API, practically speaking, both are relevant and therefore framework support is desirable.

- **HTTP Caching**:

  Caching must be considered since it is a mandatory constraint of the architectural style REST. See section 2.1.1 and Fielding [5, p. 79-81] for details. In the following, the situation is considered where no application-specific cache is implemented but is delegated to the infrastructure like Tilkov et al. [22, p. 138] describe it. Therefore the support for the HTTP mechanisms defined in RFC 7234 [7] and RFC 7232 [8] are evaluated. Conditional requests defined in RFC 7232 [8] is on the one hand an effective mechanism for cache updates and on the other hand, the lost update problem can be prevented, and are thus examined as well.

- **Content Negotiation**:

  According to Tilkov et al. [22, p. 88], the selection and definition of representations is a huge part of the development process of a REST API. Since one resource could have multiple representations, we need some mechanism for the client to select the desired representation [19, p. 32]. Thereby a different representation could mean a different data format, like JSON or XML, or a different view on the same resource [19, p. 239]. The HTTP protocol provides a mechanism for this, called Content Negotiation, which is defined in RFC 7231 [9]. This mechanism is based on specific HTTP headers, with which the clients can specify which formats they prefer. A framework should include utilities to use Content Negotiation RFC compliant and to provide the client with the requested format. Furthermore, the framework is also analyzed to see if it allows developers to implement different route handlers for the same route. The only difference between the different handlers is the media type they produce and/or consume. For example, creating an HTML representation might require more steps than creating a JSON representation, so two different route handler would be required.

- **Custom Media Types**:

  The support for custom media types is necessary for several parts. One use case is the versioning via different media type like described in [22, p. 189]. Thereby a media type for each version is defined, for example *application/vnd.user.v1+json* and *application/vnd.user.v2+json*. In the same way, this strategy can be used to implement different views on the same resource.
  As one can see, custom media types have multiple use cases and therefore the evaluated framework should support the processing of media types besides the standard ones like *application/json*.

- **Extensibility**:

  The last point considers the extensibility of the analyzed frameworks and how easily a framework can be extended to include the missing functionality.

The resulting template is shown in Table 3.1. In the sections below, we will analyze the four frameworks Express, NestJS, Fastify and restify. Therefore we will refer to the latest currently available version. As a basis for the analysis, we use the current state of the source code, the documentation and the website, if available. Thereby we do not consider any third party modules but only the built-in functionalities or modules that are part of the ecosystem or developed by the team of the framework.

For the rating of each analyzed aspect of the frameworks, a rating scale adapted from Fielding [5] is used. Thereby, the more plus an aspect has, the better the respective framework supports the developers in this area. The same only the other way around applies to the minus.

Hypermedia (HM)	Pagination & Filtering (P&F)	HTTP Caching (C)

Content Negotiation (CN)	Media Types (MT)	Extensibility (E)

Table 3.1.: Template to analyze the selected frameworks.

3.2.1. Express

Express[2] reached version 1.0.0 on 17 November 2010 and is thus the oldest analyzed framework. Currently, it is available in version 4.17.1. "Express.js is sometimes considered the de facto solution when it comes to building a web application in Node.js, much like Ruby on Rails was for Ruby for a long time" [4, p. 131]. For this reason, Express is the first framework we will analyze and evaluate.

Framework	HM	P&F	C	CN	MT	E
Express	-	+	+	+	+	+

Table 3.2.: Results of the template analysis of Express.

[2] https://github.com/expressjs/express

Hypermedia

Express comes with very basic hypermedia support. All it provides are the functionalities to set *Location* and *Link* headers.

Pagination

Pagination is not supported in any way. But unlike the others there is a module, created and maintained by the Express team, to implement pagination. However, this is limited to one pagination strategy and is only working with relative URIs. Filtering is only supported in the regard that query parameters can be extracted.

HTTP Caching

ETags are generated and set by default. This can be configured so that Express generates weak or strong ETags or generate none. With these ETags, there also comes built-in support for conditional GET requests. And unlike to restify[3], this is located in the core of Express, so the usage is much more efficient. Besides the *If-none-match* header, containing the ETag, even the *If-modified-since* header is respected. Methods to define the caching behaviour or functionalities to implement conditional PUT or DELETE requests are not available.

Content Negotiation

Express has built-in but very basic content negotiation support. The developer can implement different functions, that each format the resource into another representation, for each produced media type. Then, Express parsers the *Accept* header and based on that selects the appropriate format function. If the requested media type can not be provided, Express sends the client automatically the appropriate HTTP status. But it is not possible to implement a completely different route handlers for different media types.

Media Types

Out of the box the media types *application/json*, *application/x-www-form-urlencoded* and *text/plain* are supported. The JSON body parser can be extend so it also parsers

[3]See section 3.2.4

any media type with JSON syntax[4] . For further media types, such as XML, must be implemented with a middleware.

Extensibility

As Express is very minimalistic and unopinionated it is easy to extend. However, one must note that the middleware approach can be limited, just like with restify.

3.2.2. NestJS

In strong contrast to the other three analyzed frameworks, NestJS[5], currently available in version 7.6.17, is very opinionated and feature-rich framework. Even though, it is not required to use TypeScript, NestJS is still very TypeScript leaning, as it also uses TypeScript decorators heavily. NestJS internally uses Express or Fastify as HTTP server framework.

"Nest provides a level of abstraction above these common Node.js frameworks (Express/Fastify) but also exposes their APIs directly to the developer. This gives developers the freedom to use the myriad of third-party modules which are available for the underlying platform" as the NestJS website states. Therefore, in the following we will focus on the features offered by NestJS, since the features offered by Express and Fastify can also be used, but are also already analyzed here.

Framework	HM	P&F	C	CN	MT	E
NestJS	-	-	-	- -	/	- -

Table 3.3.: Results of the template analysis of Nestjs.

Hypermedia

NestJS does not add any hypermedia functionalities.

Pagination

NestJS does not come with pagination support.

[4]to be recognized if the media type ends with *+json*
[5]https://github.com/nestjs/nest

HTTP Caching

There is built-in caching support in NestJS. However, this focuses on server-side caching, which will not be considered here. Assistance for developers to define the caching behaviour or to utilize conditional requests is not given. But through the concept of Interceptors[6] in NestJS, it is possible to implement conditional requests more efficiently compared to Express for example.

Content Negotiation

NestJS do not have any content negotiation support.

Media Types

The support for custom media types depends on the underlying HTTP server framework. See section 3.2.1 and section 3.2.3.

Extensibility

With Providers, Middlewars, Interceptors, etc. NestJS have a lot of concepts for the developers to add functionalities easily to their application. The problem is thereby, that extensions, which cannot be developed within the scope of these concepts, are very hard to implement. An example of this is content negotiation. Thus it is very difficult to implement it in such a way that one can have different handlers for different produced media types.

3.2.3. Fastify

Fastify[7] reached version 1.0.0 on 6 March 2018 and is currently available in version 3.15.1. It is built with a focus on performance and developer experience. According to their benchmarks[8] Fastify is one of the fastest Node.js frameworks and up to 4.6 times faster than Express.

One key feature of Fastify is its plugin architecture and ecosystem[9]. Fastify itself has very few built-in features but can be extended with a variety of different plugins. For

[6] https://docs.nestjs.com/interceptors
[7] https://github.com/fastify/fastify
[8] https://www.fastify.io/benchmarks/
[9] https://www.fastify.io/ecosystem/

this purpose, the Fastify team maintains a set of core plugins based on their Long Term Support strategy. Thereby these plugins usually have a stable and high-quality API. Besides the core plugin, there are community maintained plugins, that usually follow the Fastify best practices but are not maintained by the Fastify team. All in all, there are currently 187 plugins listed in the official ecosystem. We consider these plugins less as external modules and more as a part of Fastify since the plugins are partly developed by the Fastify team or at least get a review from them. However, it is important to keep in mind not to consider them completely as part of Fastify, since they are not built-in and still need to be deliberately selected and installed by the developer.

Framework	HM	P&F	C	CN	MT	E
Fastify	- -	-	+	+	+	+++

Table 3.4.: Results of the template analysis of Fastify.

Hypermedia

Fastify does not provide any functionalities to work with hypermedia. For example, there is not even a method to set the Location Header to allow at least minimal hypermedia support, which is even defined in the HTTP specification (see section 4.3.3 in RFC 7231 [9]).

Pagination & Filtering

The same applies to the topics pagination and filtering. Currently, no listed plugin is maintained for either of them. Filtering is supported in the regard of extracting and reading the query parameters, that are often used to perform filtering on a collection resource. But what Fastify does not support is the creation of search templates.

HTTP Caching

Unlike the first two issues, caching has significantly better support via core plugins. There is support for the automatic generation of ETags based on the response payload and for the evaluation of conditional GET requests. However, only the If-none-match is taken into account and the If-modified-since is ignored. Methods for setting the Expires header and defining the caching behaviour via the Cache-Control header does also exist. A functionality that is also implemented, but not considered here, is server-side caching.

Content Negotiation

Fastify has no built-in methods for Content Negotiation, but again there are plugins to extend Fastify. So it possible to parse the accept header and extract the desired representation format. It is also possible to format or rather serialize depending on the content type of the response payload. What in contrast is not possible out of the box is to implement different route handlers for different media types. But it is possible with so-called constraints to implement content negotiation at the HTTP router level. Of all the frameworks analyzed, Fastify is the only one where this is possible.

Custom Media Types

Fastify only supports the media types *application/json* and *text/plain* out of the box. But it also ships a powerful content type parser API. With this, one can register a content type parser for each different content type. This gives the developer the possibility to process each content type very specifically.

Extensibility

Extensibility is one of the core features that Fastify comes with. Through its plugin architecture, decorators[10] and plugins developers are able to extend Fastify with every required functionality. Hooks enable the developer to listen to every event occurring during the request/response lifecycle. This offer much more possibilities to intervene in the processing of a request compared to the middleware approach of Express. It simplifies the automatic generation of ETags for example.

3.2.4. Restify

Restify[11] reached version 1.0.0 on 13 February 2012[12]. It is currently available in version 8.5.1. Restify was built to provide a framework to build "semantically correct RESTful web services ready for production use at scale" as stated on its website. One goal of it is to be as compliant as possible with the RFCs. To achieve this goal, restify comes with a set of plugins. In contrast with Fastify, these plugins are built-in and do not have to be installed separately. In return, however, restify does not have as extensive an ecosystem as Fastify.

[10] Not to be confused with the decorators from TypeScript. See subsection 2.3.4
[11] https://github.com/restify/node-restify
[12] https://github.com/restify/node-restify/releases/tag/v1.0.0

Framework	HM	P&F	C	CN	MT	E
restify	++	+	-	++	++	+

Table 3.5.: Results of the template analysis of Restify.

Hypermedia

Restify has very basic hypermedia support but has already significantly better support than most of the others. Restify let the developer give routes a unique name. This name can then be used later to reference this route in any route handler. With this functionality, one can link to other resources in the response for example. Thereby path and query parameters can be rendered into the route. Although these are first relative and not absolute URIs, restify also provides a method to retrieve the absolute URI so it is easy to build absolute URIs. In addition, restify supports the developer to set the Link header correctly, but without considering the *type* parameter.

Pagination & Filtering

Like the other frameworks before it restify has no built-in support for pagination. Filtering is supported in the regard that query parameters can be extracted and that search templates can be created with the route render functionality described in the previous section.

HTTP Caching

Restify does support the developers to define the caching behaviour with several methods. Firstly, there is a method to set the *Cache-Control* header. There is also a method to prevent caching by the client by setting the appropriate headers (*"Cache-Control, Pragma, Expires"*).

In theory, there is also a plugin that implements conditional requests. This would even consider, in contrast to Fastify[13], the *If-modified-since* and *If-unmodified-since* headers and do not implement conditional requests only for GET but also for PUT and DELETE. But due to its middleware style, this plugin can not be used in an efficient way[14]. Thereby it is very confusing for the developers to use this plugin and so it is hard to implement these concepts RFC 7232 [8] compliant.

[13] See section 3.2.3

[14] Therefore, there was also a pull request (see https://github.com/restify/node-restify/pull/536) to move this to the restify core instead of a plugin as this would solve the problem. However, the pull request was never merged.

Content Negotiation

Restify has the best support for content negotiation of all frameworks. Restify allows the developer to implement a dedicated formatter for each media type produced, which then has access to the response payload and can format this one. This has two advantages. First, the logic to convert the resource into the different representations is separate from the business logic and is thus more maintainable. Second, restify can check directly at the beginning of the request processing whether the representation requested by the client can be provided by the server and if not, send the corresponding HTTP status.

In addition, restify is the only one to support the implementation of different route handlers for different media types. This makes it very easy to restrict access to specific representations of a resource.

Media Types

Any media type is supported by simply implementing a suitable middleware that handles the corresponding media type. For *application/json* as well as for any media type with JSON syntax[15] there is a built-in parser.

Extensibility

As restify is still very unopinionated, since the features to make the API semantically correct do not have to be used, it is open for extensions. However, it is limited through its middleware concepts if compared with the hook concept of Fastify.

3.2.5. Summary

Framework	HM	P&F	C	CN	MT	E
Express	-	+	+	+	+	+
NestJS	-	-	-	- -	/	- -
Fastify	- -	-	+	+	+	+++
restify	++	+	-	++	++	+

Table 3.6.: Summary of all analyzed frameworks.

We will conclude by summarising the results of the analysis of the frameworks. As one can see in Table 3.6, none of the frameworks analysed offer support for the developer

[15] to be recognized if the media type ends with *+json*

in every section of the template. Even restify, which presents itself as a framework for REST API development, has a lack of support for pagination and caching. Compared to the other frameworks it has good support for hypermedia, but it is still very rudimentary and not sufficiently good to enable the developers to build REST API effectively. This means that developers have to use a framework that does not support them in all aspects. Furthermore, none of the frameworks analyzed enforces REST-compliant development.

3.3. REST and Node.js

In the previous section, several existing Node.js frameworks were analyzed with the result that none of the evaluated frameworks provides sufficient support for the development of REST APIs. Therefore, this section assumes that there is no Node.js framework that provides sufficient support for developing REST APIs. If a developer wants to develop one in Node.js, two different scenarios can be considered.

In the first scenario, we assume that the software developers understand all the concepts of REST, including hypermedia. But although they understood the concepts, the design and implementation is still a very difficult and time-consuming task [20]. This is worsened by the fact that there are not any tools, that support the developers to create RESTful APIs efficiently. Due to this the developers have to build the application from scratch, only using a framework for the basic interaction with HTTP, like routing. Since the developers have understood the ideas behind REST, the final API will be still RESTful. However, this once again significantly increases the development time.

In the second scenario, the developers have not fully understood the concepts behind REST. So they would rely on having a tool, that enforces the REST compliant development. Since this does not exist, the final API can not be RESTful in any way. To summarize, one can say, that in both scenarios the developers would profit from a framework, that on the one hand enables the developers to efficiently build REST APIs and that enforce it in a very strict way at the same time.

4. Solution

As shown in the previous chapter, it is currently very challenging to implement a Web API, that complies with the REST constraints. Therefore, in the following chapter, we would like to present our solution to this issue.

Schreibmann et al. [20] proposed a model-driven approach to overcome the problems of implementing a REST API. With their approach they can on the one hand ensure that the resulting API is REST compliant but they also lose flexibility and the ability to model any kind of REST API. In contrast to this approach, we propose a Node.js framework that, first, helps developers efficiently build a REST API and second, forces developers to follow the constraints. The framework should make it overall harder for the developer to harm the constraint than to follow. Of course, this can not be as strict as with the model-based approach, but therefore flexibility is gained. See Figure 4.1 for a graphical interpretation. The framework presented in this work can be used to implement any API that can be modelled using the model-driven approach, and at least a large subset of the possible REST APIs. However, our approach cannot prevent the development of non-REST compliant APIs. So all in all, it is a tradeoff of REST compliance for flexibility.

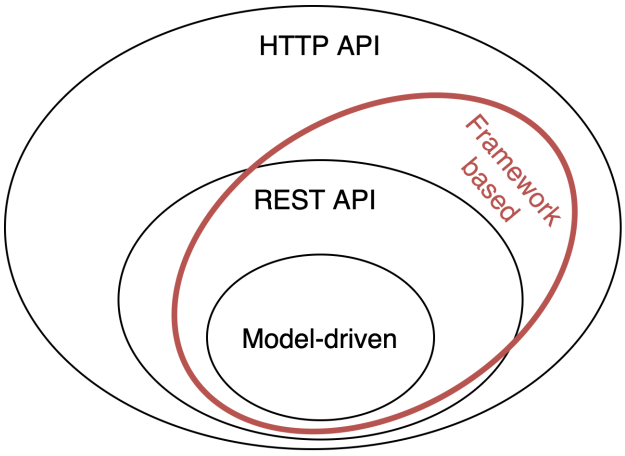


Figure 4.1.: Model-driven and framework-based approach for the development of REST APIs graphically illustrated. Adapted from [20]

4.1. Framework Design

As described above, the goal is to design a framework that enables developers to build RESTful APIs efficiently and thereby also enforces strict compliance with the REST constraints. Therefore we will first have a look at the framework design that enforces the REST compliant development.

4.1.1. HTTP Framework

Just like NestJS[1] we use an existing HTTP server framework and build a wrapper around it. This wrapper should then ensure REST compliant development. Before discussing the structure of this wrapper, the reasoning behind the decision to use Fastify is explained. The main reason for choosing it is its extensibility. As already analyzed in the previous chapter, Fastify comes with a small set of built-in features but makes it very easy to add some via its plugin architecture and hook system. Since among the analyzed frameworks, there is no, that provides everything needed to develop a REST API, either way, a lot of development has to be done from scratch. And this is significantly simplified if the underlying framework is easy to extend. Besides this, Fastify offers further advantages. The first to mention should be that Fastify gives the developers the ability to influence the HTTP routing process. This is made possible by the router[2] that Fastify uses internally. This allows, as already mentioned in the previous chapter, to implement content negotiation at the router level. A further decision criterion is the performance of Fastify.

4.1.2. State Templates

While developing a Web API, that is REST and HTTP compliant, one will recognize two circumstances. The first one is the repetitive tasks that must be implemented at each endpoint. This includes the link generation for the pagination or the implementation of conditional requests for example. The second one is that the sequence in which the tasks are processed is usually the same. To see this again with an example, we consider a client's request to get the representation of the resource. Thereby the server will first check if the client is authorized to request this resource before the server checks if this resource does even exist and then convert the resource to its representation.

To overcome the issue of the repetitive tasks, each endpoint is implemented as a whole class instead of a single method. This allows the common tasks to be refactored into an abstract base class. To tackle the second issue we apply the template pattern [12, p. 325 ff.] to the abstract class. This skeleton, specified in the template method,

[1] See subsection 3.2.2

[2] See https://github.com/delvedor/find-my-way for details

consists of concrete methods, implementing the repetitive tasks, and abstract methods. These abstract methods must then be implemented by the developers to do the request specific tasks, such as the database access.

If this approach is applied to each HTTP CRUD method[3], since each method requires different sequence flow, five[4] different classes are obtained. When these five classes are analyzed, one can see that many functionalities are implemented in all classes. Since there is no need for the duplicated code, further abstract classes are created, which are the base for every other class. These classes contain all common functionalities that are used by all subclasses such as authentication and methods that enable hypermedia. These steps result in the class hierarchy seen in Figure 4.2. In this graphic, the classes are named like *States*. The reason for this, as already described in subsection 2.1.2, is that we are defining an endpoint as an application state.

To sum it up, the framework provides an abstract base class for each endpoint type. The developers must then implement a concrete class, which inherits from the correct base class, for each endpoint. Due to this, the developers are supported to develop RESTful APIs efficiently, since the most required functionalities for this are already implemented in the base classes. Further, the development is also less error-prone because the framework implements the functionalities and not the developers. Additionally, they are not only supported but even enforced to consider the most important aspects, like hypermedia, through the need of implementing the abstract methods defined in the base classes. Thereby, the developers can use hook methods to extend and change the behaviour of the states. This is a common approach for the template pattern and is described in [12, p. 326 ff.]. The different states are explained in detail in section 4.2. These strict requirements are continued within the Router, which also uses the states to prevent incorrect development.

4.1.3. Router

The next part that should enforce the correct development is the so-called Router. Router refers to the component that allows developers to register endpoints that listen to specific paths.

The router does not enforce any structure regarding the registered paths, but it does enforce how an HTTP method-specific request gets processed in a very strict way. This is achieved by forcing the registered method to return the corresponding state type. To provide an example, an endpoint, that is listening for *GET* requests, would have to return an instance of a class that inherits from *AbstractGetState*. In the actually registered method, the developers just have to give the state access to the request and response object[5]. The further processing happens in the state itself. This has the advantage, that the developers can be guided more easily within the state by forcing them

[3] GET, POST, DELETE, PUT

[4] For GET there are two different cases: GET single resource and GET collection resource

[5] See subsection 2.2.4 for an explanation of the two objects

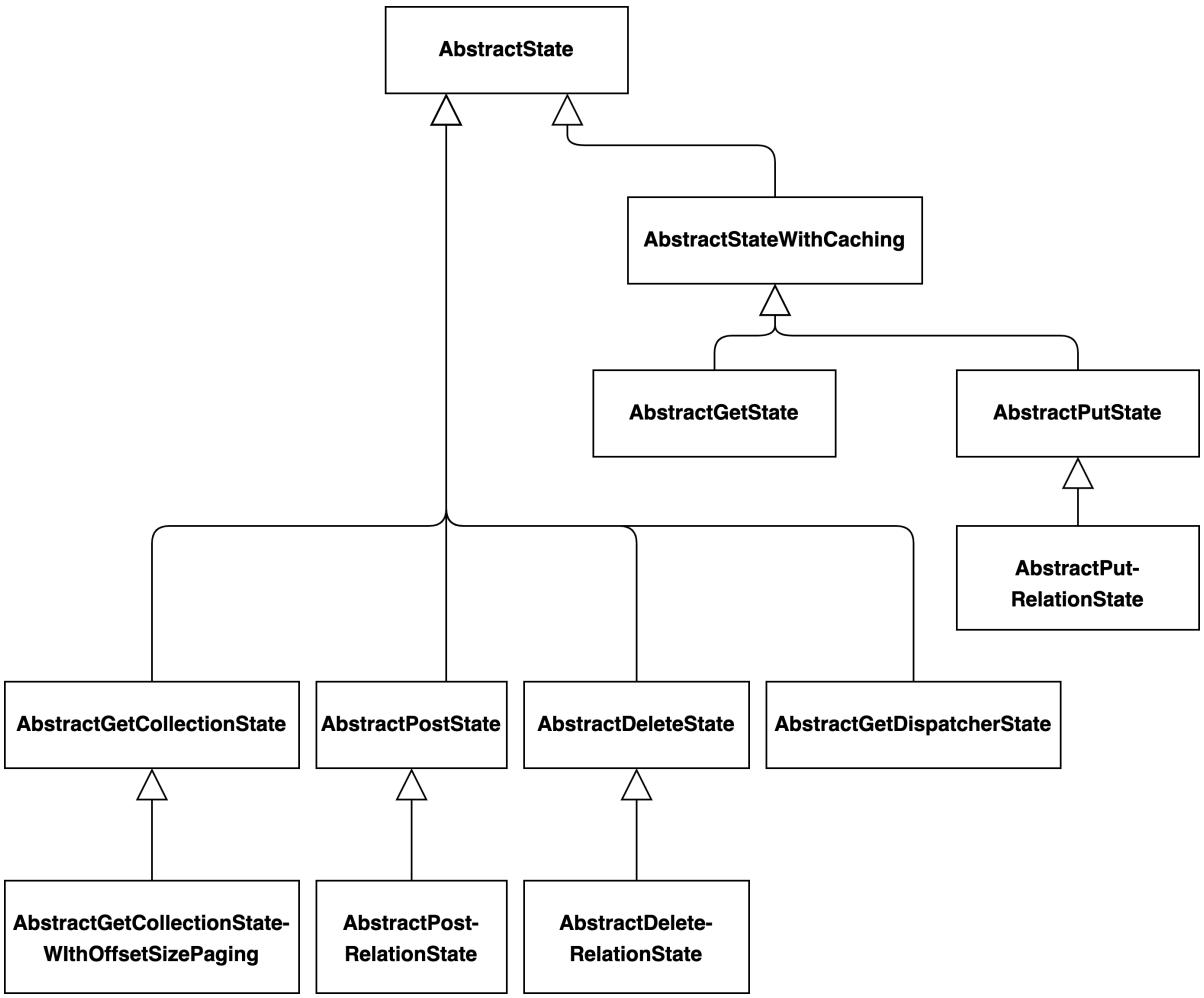


Figure 4.2.: The resulting class hierarchy when refactoring repetitive taks and common functionalities into abstract base classes for the CRUD HTTP methods.

to implement specific abstract methods for example. Compared to other frameworks like NestJS[6] this is very uncommon but it fits well with the overall strict way the framework suggests. How this is realized with TypeScript is shown in section 4.3.

4.1.4. Models

Furthermore, the framework provides base models for different use cases. These include models that serve as a basis for the classes that define the resources and the views. Views are the models with which the client interacts. So a *CreateUserView* would be defined, which the client then has to send to the server to create the resource *User*. Furthermore, the views can also define different views of the same resource, which differ, for example,

[6]See subsection 3.2.2

in the attributes that they make available to the client. of resources. By forcing the developers to use these models, the framework can implement functionalities just like conditional requests. To provide this as built-in functionality is first made possible due to the *lastModifiedAt* attribute that is guaranteed to be defined through the abstract base model.

In addition to the models for resources and views, the framework also provides models for the results returned by the database layer, which can be seen in Figure 4.3. The standardized interface between the database and the business logic layer, which is created by these models, allows again to implement functionalities that depend on the provided information. An example for this is the *CollectionModelDatabaseResult*. This model enforces the developers to provide the total count of results of the current collection query to the business logic layer. The information is then used within the state to generate the links for pagination. In section 4.4 the specific models are explained in detail.

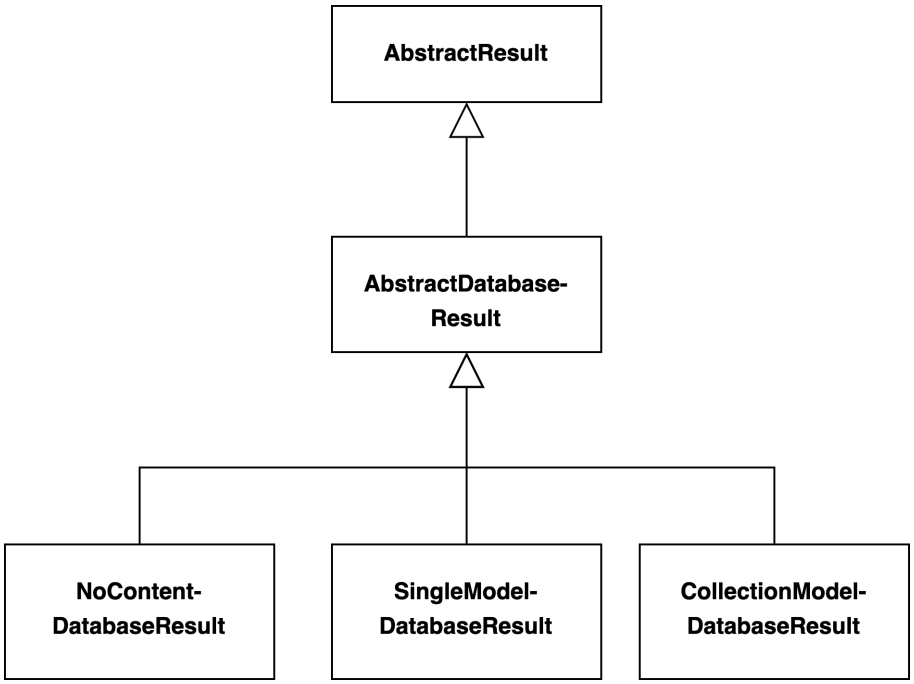


Figure 4.3.: Class hierarchy of the result model classes.

4.2. State Templates in Detail

After introducing the two basic states *AbstractState* and *AbstractStateWithCaching* with their common functionalities, the HTTP method-specific states will be described. Thereby the focus will be on the sequence of tasks they define in their template method. To give a graphical overview of this, we define a graphical model to represent it. This model is shown in Figure 4.4. The class diagrams of the classes presented here can be found in section C of the appendix.

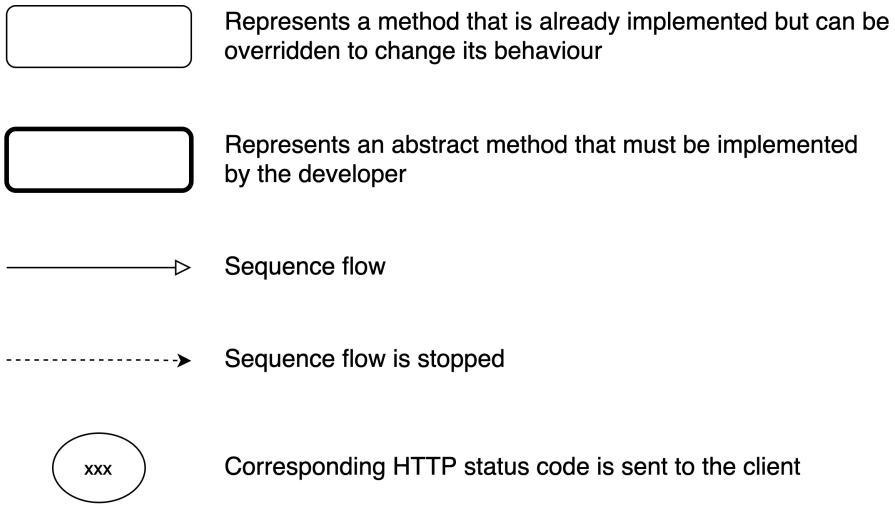


Figure 4.4.: Definition of the graphical model for the visualization of the sequence flows of the state classes.

4.2.1. AbstractState

AbstractState is the base class of any other state class in the framework. The key aspect of it is that it has only the two public methods *configure* and *build*. With the *configure* method, the state gets access to the request and response objects. Through these, all other methods within the state can access the information and data of the request. This includes the request payload, query parameters, path parameters and headers. With the response object, the state can be set among other things the response payload, the status code and response headers. However, no direct access is given to the Fastify response object but through a wrapper, namely the *HttpResponse* class. Using this wrapper, the non-REST and non-HTTP compliant development can be prevented, like setting a wrong status code. The second public method is *build*. Calling this method starts the processing of the request. Internally only the abstract template method *buildInternal* is called and thereby any uncaught error will be caught here and transformed into an HTTP response.

The reason for this restricted public interface is to give developers as few opportu-

nities as possible to make mistakes. With the current implementation, they only have to call the *configure* method and then the *build* method to start the processing of the request. Everything else is done in the state classes, in which the developers are guided by the abstract methods, that have to be implemented.

Besides the two public ones, there are other methods, that for example support the developer to implement role-based access control. The most important methods within the class will be briefly presented here.

configureState

Within this method, the developer configures several settings of the state. This includes the activation of the API Key verification and the authorization check, setting required user roles, defining state entry constraints and defining the caching behaviour.

```
1
2     protected configureState(): void {
3       this.activateApiKeyVerification()
4     }
5
```

Listing 4.1: Exemplary use of the configureState method.

extractFromRequest

As described above, the state has access to the request object. In the method *extractFromRequest* the developers extract any information that is required to process the request. This could be the identifier of the requested resource. See below for an example.

```
1
2     // using method ‘extractFrom‘ to extract ‘id‘ from request parameters
3     protected extractFromRequest(): void {
4       this.resourceId = this.extractFrom('params', 'id')
5     }
6
```

Listing 4.2: Extract resource identifier from request within the extractFromRequest method.

addLink/addConstraintedLink

These two methods are one of the main functionalities that support the developers to follow the hypermedia constraint. For both, a URI, the relation type and optionally a media type must be specified. If the URI is a URI template and contains placeholder parameters, the developer can provide values and the template is filled with these. The resulting link is then added to the HTTP response as a link header together with the relation type and type values, as specified in RFC 5988 [17].

```
1
2     this.addLink(
3     '/users/{}',
4     'getSingleUser',
5     'application/vnd.user+json',
6     [this.userId]
7     )
8
```

Listing 4.3: Add link header to response via the addLink method.

```
1
2     this.addConstraintedLink(
3     () => this.userId < 20,
4     '/users/{}',
5     'getSingleUser',
6     'application/vnd.user+json',
7     [this.userId]
8     )
9
```

Listing 4.4: Add contrainted link header to response via the addConstraintedLink method.

As one can see in Listing 4.4 the *addConstraintedLink* method also accepts a function. This function is evaluated and the corresponding link is only added, if *true* is returned from it. Assuming the host of the server is *http://localhost:8080* and *this.userId = 23*, the method from Listing 4.3 would generate the following link header:

```
<http://localhost:8080/users/23>;rel="getSingleUser";type="application/vnd.user+json"
```

Listing 4.5: Example link header.

addStateEntryConstraint

A state entry constraint is a condition, that has to be fulfilled for the state to be executed. Via the *addStateEntryConstraint* method such constraints can be added to the particular state. The simplest example is shown in listing 4.10. There, access to the state is granted only when the user retrieves his own profile.

```
1
2     this.addStateEntryConstraint(function () {
3       return this.requstedId === this.authenticationInfo.userModel.id
4     })
5
```

Listing 4.6: Exemplary adding of a state entry constraint.

With the method *verifyAllStateEntryConstraints* every added constraint will be evaluted and *true* is returned only if every constraint is fulfilled.

convertLinks

Unlike the *addLink* method, the *convertLinks* method does not support proposing the next possible actions via link headers, but linking resources to each other. See Listing 4.7 for an example. There is a user modeled that has an *id*, a *name* and a list of friends. Since the list of friends could have thousands of entries, these should not be sent, although the client wants to have only the user. Therefore the *convertLinks* method in combination with the *link* annotation injects a link object, that points to the friend's collection resource. In Listing 4.8 an exemplary user object can be seen, in which the link is already injected. The link object contains the same values as the Link header described above.

```
1
2     class User {
3       public id: number
4
5       public name: string
6
7       @link('/users/{id}', 'getFriends', 'application/vnd.users+json')
8       public friends: Link
9     }
10
```

Listing 4.7: User model with a link decorator.

```
User {
  id: 23
  name: 'userName'
  friends: {
    href: 'http://localhost:8080/users/23/friends',
    rel: 'getFriends',
    type: 'application/vnd.users+json'
  }
}
```

Listing 4.8: Injected link in a user object.

buildInternal

The *buildInternal* method is the template method called inside the *build* method defines the HTTP method-specific sequence. This method is abstract and implemented in the HTTP method specific classes.

verifyRolesOfClient

This method checks, if this has been enabled in the *configureState* method if the client has all the required roles to execute this state. Internally, it calls the authorizeUser method. Thus, although a default implementation for role verification is given, the developer can easily override verifyRolesOfClient in subclasses.

authorizeUser

AuthorizeUser first checks if the client has sent *authenticationHeader* with. If this is not the case, it is checked whether the state may be executed without prior authentication. If it is set, the *isAccessAllowedForThisUser* method is used to first fetch the current user and its roles with the *authenticationProvider* and then the roles of the client are matched with the required roles for the execution of the state.

verifyApiKey/verifyNecessaryApiKey

If the API key verification was activated via *activateApiKeyCheck*, the *verifyApiKey* method verifies the API key sent by the client. Just as with the *verifyRolesOfClient* method, it again internally calls a method that contains the actual logic. This is the

verifyNecessaryApiKey one. With the usage of the *apiKeyInfoProvider*, it verifies the validity of the API key.

4.2.2. AbstractStateWithCaching

The *AbstractStateWithCaching* extends the *AbstractState* by caching utilities. Hence, the class provides therefore four different caching types:

- deactivate caching by appropriate *Cache-Control* header (no-cache, no-store, no-transform)
- define caching behaviour via *Cache-Control* header
- define caching behaviour via *Cache-Control* and *ETag* header
- define caching behaviour via *Cache-Control* and *Last-Modified* header

The caching behaviour should be set within the *configureState* method like in Listing 4.9.

```
1
2     protected configureState(): void {
3       this.setHttpCachingType(
4         CachingType.VALIDATION_ETAG,
5         CacheControlConfiguration.PUBLIC
6       )
7       this.maxAgeInSeconds = 3600
8     }
9
```

Listing 4.9: Define caching behaviour within the configureState method.

The configuration in Listing 4.9 would produce the *Cache-Control* header "public, max-age=3600". Additionally the *ETag* is set automatically. The enumeration *CacheControl-Configuration* gives the developer a utility to set only valid *Cache-Control* directives.

4.2.3. AbstractGetState

The *AbstractGetState* is the base class for every state, that represents *GET* endpoints. Since such a request always corresponds to a resource, the developer has to provide the class of the resource as a generic parameter. The class diagram is shown in Figure C.3 and the sequence flow in Figure 4.5. As one can see in the sequence flow, this state does support conditional requests and sends automatically a *304 Not Modified* response if the

client still knows the current state of the resource. Additionally, the state checks if the resource with the request id exists, and if not it sends a *404 Not Found* response.

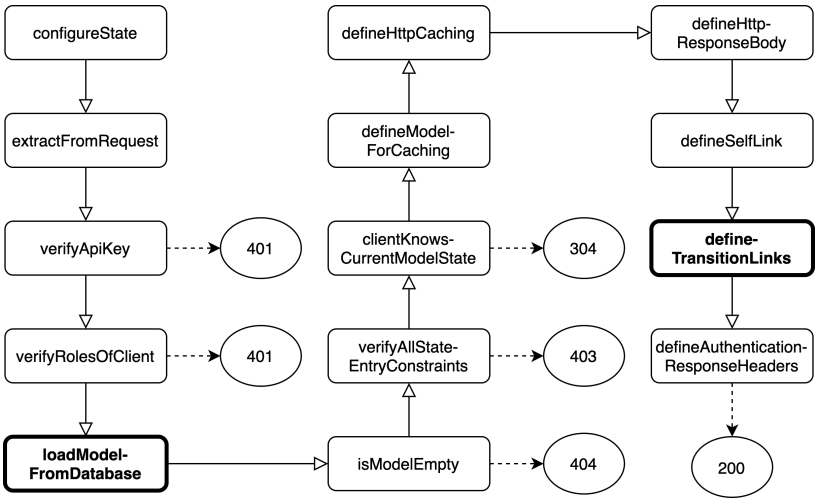


Figure 4.5.: Sequence flow AbstractGetState.

4.2.4. AbstractPostState

The *AbstractPostState* is used to create resources. Since the client does not send the resource directly but a representation of the resource, this class requires two generic parameters, namely the resource class and the representation/view class. The process of creating the resource model from the view model is also supported by this state. The logic for this is placed in the *mergeViewModelToDatabaseModel* method. Therefore, the method first receives an instance of the resource class from the abstract method *createDatabaseModel*. In this method, the developers can set values, that are required to create the resource, but which are not allowed to be set by the client, like user roles for example. The next step is to merge the view class with the resource class by the *merge* method. As shown in Figure C.4, the method signature matches exactly this behaviour. An instance of the view class can be taken and merged with an instance of the resource class and retrieve an instance of the resource class as result. This result, saved in the class attribute *modelToStoreInDatabase*, is then used within the *createModelInDatabase* method to save it in the database layer. Besides this, the state also sets automatically the *Location* header using the identifier created by the database layer.

4.2.5. AbstractPutState

This state is used when the client updates a resource. For this purpose, it is required to first fetch the current state of the resource from the database. This model is then

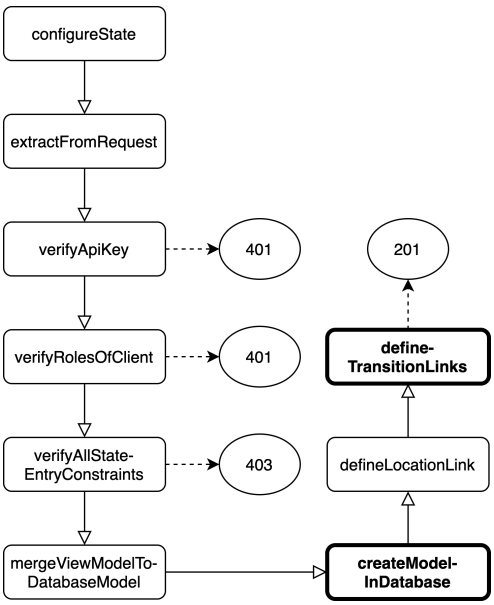


Figure 4.6.: Sequence flow AbstractPostState.

merged with the representation sent by the client. The merge method used for this takes each property of the view and sets its value in the resource to be updated. If a property was intentionally not set by the client, the corresponding property is set to *undefined* in the resource. An example usage of the merge method can be found in Listing A.1 in the appendix. The result of the merge process should be afterwards saved within the *updateModelInDatabase* method. The response to such a request is either empty or contains the updated resource. This behaviour can be configured by the developer via the *responseStatus200* attribute and the *defineHttpResponseBody* method.

4.2.6. AbstractDeleteState

Just like the *AbstractGetState* the *AbstractDeleteState* refers to a resource and therefore a generic parameter is required. Before the resource with the specified identifier is deleted, the state completes, besides the authentication and authorization, two checks. It verifies whether or not the resource exists. In case the resource is missing, a *404 Not Found* response is sent After that, it checks if the client knows the current state of the resource. If it does not, it will receive a *412 Precondition Failed* response. For this purpose, the method *createEntityTagOfResult* is used to generate an ETag of the resource, fetched from the database layer. This ETag is then compared with the ETag sent by the client. In Figure C.5 the attribute *responseStatus200* can be seen. If it is set to *true* the response payload of the respective request will contain the current state of the resource before it was deleted. If it is set to *false* the response payload will be empty and the status code will be *204 No Content*. By default the response payload is empty.

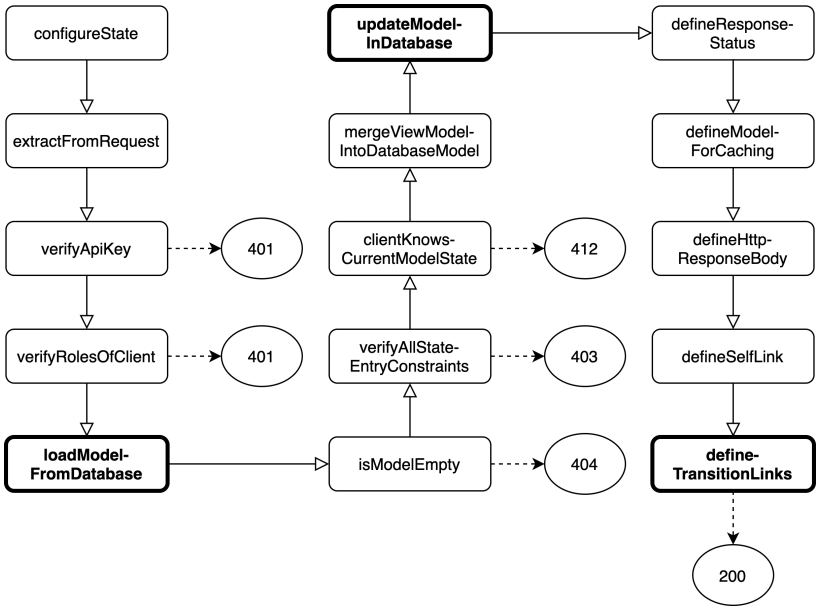


Figure 4.7.: Sequence flow AbstractPutState.

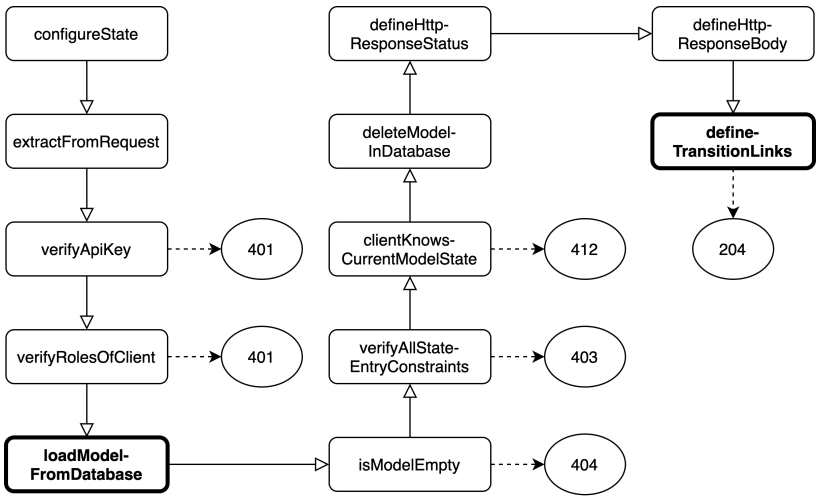


Figure 4.8.: Sequence flow AbstractDeleteState.

4.2.7. AbstractGetCollectionState

The *AbstractGetCollectionState* represents a request to get a collection of resources. Since such a collection resource could contain millions of entries, but the server does not want to serve these in a single response, the server should implement pagination strategies [19, p. 101]. Hence, the developer has to specify a pagination strategy via the *definePagingBehaviour* method. With the returned class, which must be a subclass of the *AbstractPagingBehaviour* (see Figure C.9), the links to navigate through the collection are added within the *definePagingLinks* method as HTTP Link headers. Next

to the links, HTTP headers are also set in the response, to provide the client with the information about the number of results and the total number of results. The framework provides the developers the *PagingBehaviourUsingOffsetSize* (see Figure C.11) as built-in pagination strategy. The *AbstractGetCollectionStateWithOffsetSizePaging* (see Figure C.10) uses this strategy and can be used as base class by the developers. The caching behaviour is set by default to *"no-store, no-cache"* since the frequency of changes is very high for collection resources.

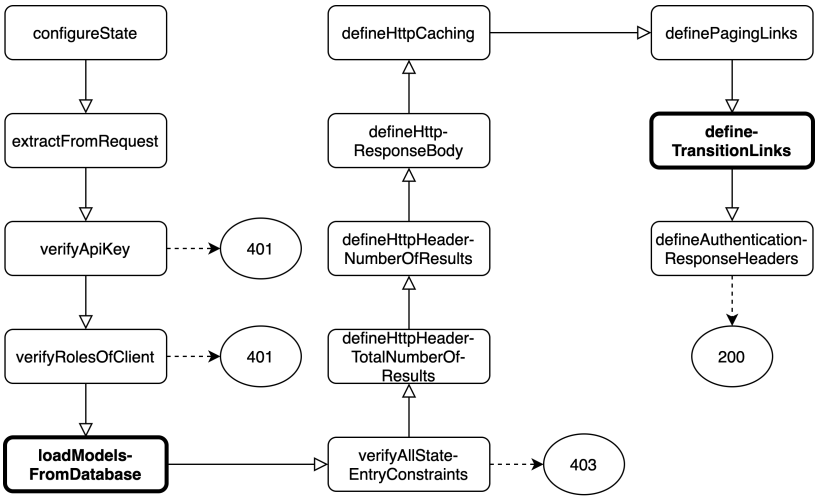


Figure 4.9.: Sequence flow AbstractGetCollectionState.

4.2.8. AbstractGetDispatcherState

The *AbstractGetDispatcherState* acts as entry point to the API. The main responsibility of it is to provide the client with all possible state transitions to start interacting with the API. This could be the link to the login page for example.

4.2.9. RelationStates

As one can see in Figure 4.2 there are several *RelationStates*. These are used when sub-resources are processed, so when the client wants to create a resource that relates to another resource for example.

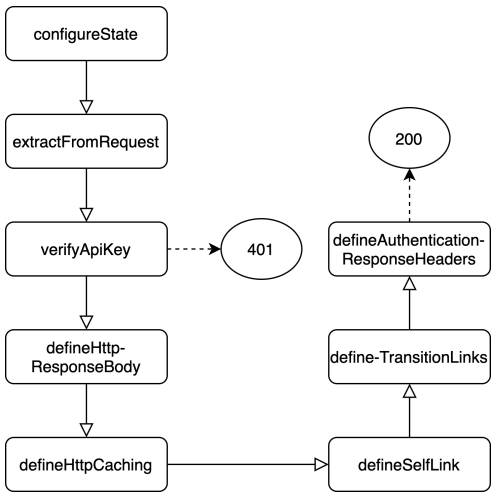


Figure 4.10.: Sequence flow AbstractGetDispatcherState.

4.3. Router

The Router is the next component with which the framework tries to enforce the developers to develop REST and HTTP compliantly. Thus the framework lets the developer register endpoints by annotating the respective method with a decorator. This concept is also used by NestJS like shown in Listing 4.11. In contrast to NestJS, where the decorator does not make any restrictions on the annotated method, the framework presented here is very strict in several places. Currently only the HTTP methods **GET**, **POST**, **PUT** and **DELETE** are supported.

The first point to mention is the required generic parameter of the decorator, which must be specified by the developer. This generic parameter must be the state class that represents the corresponding endpoint. Thereby the given class must be a subclass of the abstract one that represents the HTTP method. If we apply this to the example in Listing 4.12, the class *GetSingleUserState* must be a subclass of *AbstractGetState* since it is an endpoint that receives a *GET* request.

```
@Controller('/users')
export class UserController {

  constructor(private userRepository: UserRepository) {}

  // registers endpoint with path /users/:id
  @Get('/:id')
  getSingleUser(@Param('id') id: string): string {
    return this.userRepository.readById(id)
  }
}
```

Listing 4.10: Endpoint registration with NestJS.

```
@Controller('/users')
export class UserController {

  @Get<GetSingleUserState>({
    path: '/:id',
    produces: 'application/vnd.user+json'
  })
  getSingleUser(
  req: HttpRequest,
  res: HttpResponse
  ): Configured<GetSingleUserState> {
    return new GetSingleUserState().configure(req, res)
  }

  @Put<PutUserState>({
    path: '/:id',
    produces: 'application/vnd.user+xml'
    consumes: 'application/vnd.user+json'
  })
  updateUser(
  req: HttpRequest,
  res: HttpResponse
  ): Configured<PutUserState> {
    return new PutUserState().configure(req, res)
  }
}
```

Listing 4.11: Endpoint registration with the framework presented in this work.

Furthermore, the decorator also dictates the method's signature. Any method registered as an endpoint must have *HttpRequest* and *HttpResponse* as its only method parameters. Additionally, the return type must be the class already given as a generic parameter. However, as one can see in Listing 4.12, it is not the state that is returned

but the interface *Configured*, which is returned by the *configure* method. So the developer is forced to call the method and provide the request and response object to the state. Together with the very clear API of the states, these two restrictions ensure that on the one hand, the developer can hardly make any mistakes in the controller method, and on the other hand, most of the processing steps happen in the state. This has the advantage, that within the state, the developer can be guided better by forcing him to implement the abstract methods.

In Listing 4.12 is shown, that the developer must specify not only the path but also the media type that will be produced and/or consumed. Since it is HTTP method-specific what the developer can or must specify, the necessary definition for each HTTP method will be briefly explained here. Thereby the framework tries to force the developers to define the endpoints according to their definition in RFC 7231 [9, p. 24 ff].

- GET:

  Since the payload of a *GET* request has no defined semantic, the developer is only allowed to specify the produced media type. The *ViewConverter* is responsible to convert the resource to the requested representation.

```
1
2       interface GetRouteDefinition {
3         path?: string
4         viewConverter: ViewConverter
5         produces: string
6       }
7
```

Listing 4.12: Interface of the route definition for GET endpoints.

- PUT:

  The developers have to specify a media type that is consumed since the client has to send a representation of the resource to be updated in the request payload. As already mentioned in subsection 4.2.5, the developer can define whether or not the response payload contains the updated resource. Because of this, the produced media type and the ViewConverter are optional.

```
1
2       interface PutRouteDefinition {
3         path?: string
4         viewConverter?: ViewConverter
5         produces?: string
6         consumes: string
7         schema: Schemas
8       }
9
```

Listing 4.13: Interface of the route definition for PUT endpoints.

The **Schemas** type groups the types of the four request components **body**, **query**, **params** and **headers**.

- DELETE:

  Just like for *GET* the request payload has no defined semantic and hence the request body is ignored for *DELETE* requests. The response payload is again optional.

```
1
2       interface DeleteRouteDefinition {
3         path?: string
4         viewConverter?: ViewConverter
5         produces:? string
6       }
7
```

Listing 4.14: Interface of the route definition for DELETE endpoints.

- POST:

  Since the purpose of a *POST* is to create a resource, the client must send a request payload and also specify the media type that will be consumed. According to RFC 7231 [9] the response of such a request can contain the created resource. To force the client to make use of the location link, we decided to not follow this proposition. The request on the resource just created fills the cache entry with the corresponding value, from which not only the individual client can benefit but also other clients.

```
1
2       interface PostRouteDefinition {
3         path?: string
4         consumes: string
5         schema: Schemas
6       }
7
```

Listing 4.15: Interface of the route definition for POST endpoints.

4.4. Provided Base Models

In the following, the two abstract base classes for all resources and views, that are defined, when working with the framework, are presented. Thereafter, the classes for modelling results, in particular database results, are explained.

4.4.1. Abstract- and AbstractViewModel

The framework provides two different base models. One for the definition of so-called views and one for the definition of models respectively resources. With views, the developers can define a restricted viewing of resources. This is used while creating or receiving a resource. Its usage can be compared with the representation, defined in the formal context of REST. Both the *AbstractViewModel* and the *AbstractModel* define the two attributes *id* and *lastModifiedAt*. See Listing 4.17 for their definition as TypeScript classes. As already mentioned the *lastModifiedAt* attribute is for example used to implement conditional requests.

```
1
2     class AbstractModel {
3       public id: string | number
4
5       public lastModifiedAt: number
6     }
7
8     class AbstractViewModel {
9       public id: string | number
10
11      public lastModifiedAt: number
12    }
13
```

Listing 4.16: Base models AbstractModel and AbstractViewmodel provided by the framework.

4.4.2. Results

The framework also provides an abstract model for any kind of results. See Figure 4.11 for details. However, the focus here should be on the classes that model the results from the database layer, namely *NoContenDatabaseResult*, *SingleModelDatabaseResult* and *CollectionModelDatabaseResult*. These classes create a uniform interface between the database layer and the state (business logic). If an error occurs during the database querying, the result classes wrap the raw error and provide an error message and code to the state. Additionally, the results give the developers the possibility to track the duration of the database query execution, which can be very useful while dealing with time-consuming queries. Besides these common functionalities, each concrete class has also specific functionalities. The *SingleModelDatabaseResult* checks if the requested resource was found and the *CollectionModelDatabaseResult* provides the total number of results for example.

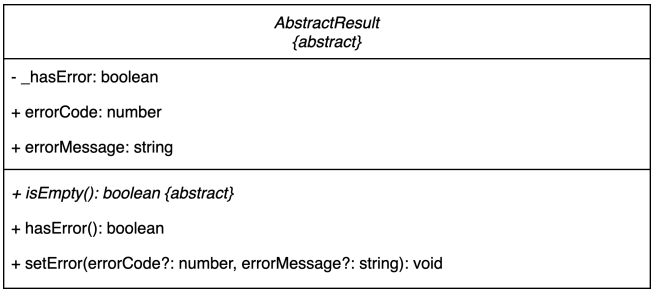


Figure 4.11.: Class diagram AbstractResult.

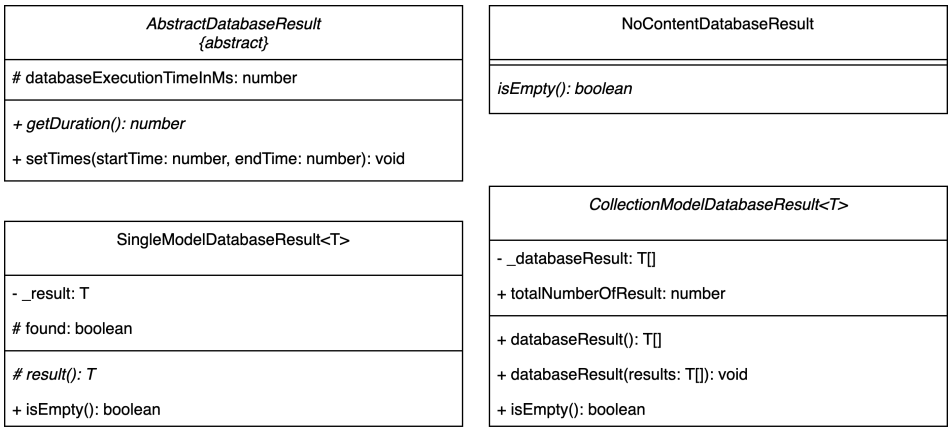


Figure 4.12.: Class diagrams of the four different database result classes.

4.5. Further Functionalities

In addition to the functionalities provided to support and enforce REST-compliant development, there are other functionalities required to efficiently create APIs. These include **Dependency Injection**, **Serialization**, **Validation** and **Security**.

4.5.1. Security

The framework supports the developers to implement a role-based authentication concept. Thereby it provides the classes shown in Figure C.12. With these classes, developers can define different roles and role concepts, as illustrated in Listing 4.18.

```
1
2   const adminRole = new Role('admin')
3   const userRole = new Role('user')
4   const guestRole = new Role('guest', true)
5
6   const roles1 = new Roles(Logical.OR, adminRole, userRole)
7
8   const roles2 = new Roles(guestRole)
9
10  roles1.matches( [ "guest" ]) // false
11  roles1.matches( [ "user" ]) // true
12  roles1.matchesWithoutAuthentication() // false
13  roles2.matchesWithoutAuthentication() // true
14
```

Listing 4.17: Exemplary use of the Role and the Roles classes.

This role concept can then be used within the states to allow access to states only for certain roles. In the example in Listing 4.19, the current user must have the role **admin** to be allowed to access the state.

```
1
2   protected configureState(): void {
3     this.allowedRoles = new Roles(new Role('admin'))
4   }
5
```

Listing 4.18: Exemplary definition of the necessary client roles to be allowed to execute a state.

Besides the role concept, the framework provides the two interfaces *AuthenticationInfoProvider* and *ApiKeyInfoProvider*. The developers must implement concrete classes for the two interfaces. While the only responsibility of *ApiKeyInfoProvider* is to verify

incoming API keys, the *AuthenticationInfoProvider* must authenticate and authorize the client. For example, if the provider implements a JWT strategy it is responsible to sign and verify the tokens. Additionally, if authentication is activated for the specific state, it provides the current state information about the current user, like user id and roles. This information is grouped within the *AuthenticationInfo* class which is returned from the *get* method (see **??**). The *AuthenticationHeader* class (Figure C.15) used by the *AuthenticationInfoProvider* acts as a wrapper around the request object. For example, this wrapper extracts the credentials from the Authorization header. Thereby, the two types Basic and Bearer are supported. Even though the framework has a default implementation of such a class, the developers can easily extend this class with their own implementation. This class would then contain all information the developers want to provide the states.

These two provider classes can then be registered in the application and thus automatically registered in the dependency injection container as well. They are then also automatically injected into each state. Listing 4.20 shows the registration of the two providers.

```
1
2     app.registerAuthenticationInfoProvider(AuthenticationInfoProvider)
3
4     app.registerApiKeyInfoProvider(ApiKeyInfoProvider)
5
```

Listing 4.19: Registration of AuthenticationInfoProvider and ApiKeyInfoProvider.

4.5.2. Dependency Injection Container

The framework internally uses the *TSyringe*[7] dependency injection (DI) container. The main use case of it is to load dependencies dynamically in the states. With the *@State* decorator, which is used in Listing 4.29 for example, the state is registered within the DI container. Through its registration, the developer can inject any dependency that was also registered before.

By default the previously registered *authenticationInfoProvider* and *apiKeyInfoProvider* are injected. This is done within the *AbstractState*, so both are available within all other states. Since these two are registered in the DI container, the developers can also inject other dependencies into these providers. A common use case is, for example, to inject the user's data storage into the *AuthenticationInfoProvider* for login purposes.

[7] https://github.com/microsoft/tsyringe

4.5.3. Serialization and Validation

The last topics considered are serialization and validation. Validation in this context means validating the incoming data from the client. For this process, the Ajv[8] JSON schema validator is used. To ensure that the developers do not have to write the JSON schemas by themselves, they can annotate the attributes of their view classes to generate the JSON schema automatically. An example class with its corresponding JSON schema can be seen in Listing 4.21 and Listing 4.22. "JSON Schema[9] is a vocabulary that allows you to annotate and validate JSON documents", according to their website. For more details see [14].

The automatic generation is only implemented for the validation of the request body. For the validation of request headers, parameters and query parameters the developers have to write the JSON schema by themselves. The request body payload needs to be both validated and transformed from a plain JavaScript object to an instance of the corresponding class. Referring to the example from Listing 4.21, the validated plain object has to be transformed to an instance of the *CreateUserView* class. The framework comes up with the *buildValidatorAndTransformer* method for this. This method, which receives the respective classes, then returns a method that validates and transforms the request payload. It is required to have an instance of the class, so it is possible to use reflection since the metadata is only available for classes and not for plain objects.

```
1
2   export class CreateUserView extends AbstractViewModel {
3
4     @viewProp()
5     public name: string
6
7     @viewProp({
8       type: 'string',
9       maxLength: 25,
10      minLength: 10
11    })
12    public password: string
13  }
14
```

Listing 4.20: Example view class to generate JSON schema.

Serialization is here on the one hand the process of transforming an object to a string to send it via the network and on the other hand to transform the current resource model to a view. To illustrate this with an example, see Listing 4.23 and Listing 4.24. In this example, the *User* resource is to be converted to the view *Userview*, which is missing the *password* attribute. For this purpose, just like validation, a JSON schema is generated for

[8] https://github.com/ajv-validator/ajv
[9] https://json-schema.org/

the view. This schema is then passed to *fast-json-stringify*[10], which generates a function based on this schema. The function accepts the model and returns the stringified view. So this function meets both requirements. First, it considers only the desired attributes and then converts the view into a string. Another advantage of this concept is, that the generated function from *fast-json-stringify* converts the view object up to three times faster to a string than the built-in *JSON.stringfy* method as the documentation of this modules states.

```
{
  type: 'object',
  properties: {
    id: {
      anyOf: [ { type: 'string' }, { type: 'integer' } ]
    },
    name: {
      type: 'string'
    },
    password: {
      type: 'string',
      maxLength: 25,
      minLength: 10
    }
  },
  required: [ 'name', 'password' ],
  additionalProperties: false
}
```

Listing 4.21: Generated JSON schema.

```
class User extends AbstractModel {

    @modelProp()
    public name: string

    @modelProp()
    public password: string
  }
```

Listing 4.22: Examplary user resource.

```
class UserView extends AbstractViewModel {

    @viewProp()
    public name: string
  }
```

Listing 4.23: Examplary view of the user resource

[10]https://github.com/fastify/fast-json-stringify

4.6. Using the Framework

After explaining the design of the framework in the previous sections, the following one presents the exemplary development process with this framework step by step. Therefore, a use case with the *User* resource is considered. The class diagram of this resource can be seen in Figure 4.13. For this resource, we want to implement three endpoints to create, update and fetch the resource. For sake of simplicity, it is assumed that the database layer is already implemented. The class *UserRepository* will be used for this purpose.

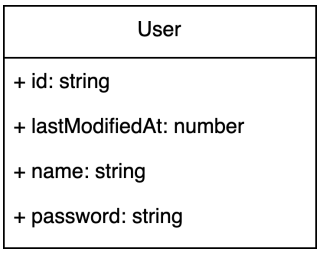


Figure 4.13.: Class diagram of the user resource class, used in the exemplary development with the framework presented in this work.

The **first step** (Listing 4.25) is to define the resource with a TypeScript class. Note that this class must be a subclass of AbstractModel. The *@modelProp* decorator is used to give the framework access to class property metadata. It is comparable to the *@viewProp* decorator.

```
1
2   export class User extends AbstractModel {
3
4     @modelProp()
5     public name: string
6
7     @modelProp()
8     public password: string
9   }
10
```

Listing 4.24: User resource class, used in the exemplary development with the framework presented in this work.

The **second step** is to define the views with which the client can create, update and read the resource. Since the client should not be able to fetch the user resource containing the password, we just define the *name* property within the *UserView*. Just like in the first step, the *@viewProp* decorators are used to give the framework access to metadata. This is then used to generate the JSON schemas for these views, for example.

```
1
2     export class UserView extends AbstractViewModel {
3
4       @viewProp()
5       public name: string
6     }
7
8     export class CreateUserView extends AbstractViewModel {
9
10      @viewProp()
11      public name: string
12
13      @viewProp()
14      public password: string
15    }
16
17    class UpdateUserView extends CreateUserView {}
18
19
```

Listing 4.25: Defining different views of the user resource.

The **third step** is to implement the three necessary states. The first to be implemented can be seen in Listing 4.27. This is used to create a *User* resource. Thereby, the *UserRepository* class is used within the *createModelInDatabase* method to save the user in the database. No links are defined in the *defineTransitionLinks* method because the client should follow the automatically generated *Location* link to get the resource that was just created. The *@State* decorator is used to register this class in the dependency injection container. If the UserRepository is registered, too, it is injected automatically in the state.

```
1     @State()
2     export class PostUserState extends AbstractPostState<User, CreateUserView> {
3
4       constructor(private readonly userRepository?: UserRepository) {
5         super()
6       }
7
8       protected createDatabaseModel(): User {
9         return new User()
10      }
11
12      protected createModelInDatabase(): Promise<NoContentDatabaseResult> {
13        return this.userRepository.create(this.modelToStoreInDatabase)
14      }
15
16      protected defineTransitionLinks(): Promise<void> | void {}
17    }
18
```

Listing 4.26: PostUserState.

The next state (Listing 4.28) is used to receive a single user resource. The *UserRepository* is used to fetch the resource from the database. In the *configureState* the *Cache-Control* is set to *"public, max-age=86400"* and the automatic *ETag* is activated. In the *defineTransitionLinks* method, the server defines three next possible state transitions that the client receives as HTTP link headers.

```
1
2   @State()
3   export class GetSingleUserState extends AbstractGetState<User> {
4
5     constructor(private readonly userRepository?: UserRepository) {
6       super()
7     }
8
9     protected defineTransitionLinks(): Promise<void> | void {
10      this.addLink('/users', 'getAllUsers', 'application/vnd.user+json')
11      this.addLink('/users/{}', 'deleteUser', [this.requestedId])
12      this.addLink('/users/{}', 'updateUser', 'application/vnd.user+json', [
13      this.requestedId,
14      ])
15    }
16
17    protected loadModelFromDatabase(): Promise<SingleModelDatabaseResult<User>> {
18      return this.userRepository.readById(this.requestedId)
19    }
20
21    protected configureState(): void {
22      // sets Cache-Control header to 'public, max-age=86400'
23      this.setHttpCachingType(
24      CachingType.VALIDATION_ETAG,
25      CacheControlConfiguration.PUBLIC
26      )
27      this.maxAgeInSeconds = 86400 // 1 day
28    }
29  }
30
```

Listing 4.27: GetSingleUserState.

Finally, the *PutUserState* in Listing 4.29 is used to update a user resource. In addition to the definition of possible state transitions, the loading of the current resource and the saving of the updated resources, other tasks are performed by the base class *AbstractPutState*. The most important one is merging the current resource state with the one sent by the client. The result of it is assigned to the *modelInDatabase* property, which is then used to update the resource in the database.

```
1
2     @State()
3     export class PutUserState extends AbstractPutState<User, UpdateUserView> {
4
5       constructor(private readonly userRepository?: UserRepository) {
6         super()
7       }
8
9       protected defineTransitionLinks(): Promise<void> | void {
10        this.addLink('/users/{}', 'self', 'application/vnd.user+json', [
11        this.updatedId,
12        ])
13      }
14
15      protected loadModelFromDatabase(): Promise<SingleModelDatabaseResult<User>> {
16        return this.userRepository.readById(this.updatedId)
17      }
18
19      protected updateModelInDatabase(): Promise<NoContentDatabaseResult> {
20        return this.userRepository.update(this.modelInDatabase)
21      }
22    }
23
24
```

Listing 4.28: PutUserState.

Before the controller is implemented, the view converters and schemas must be defined as the **forth step**. Thereby, the createUserViewSchema and the upateUserViewSchema validates the incoming data based on the schema generated with the provided metadata via the @viewProp decorators. Afterwards they transform the plain JavaScript object the corresponding class defined in Listing 4.26 (*CreateUserView* and *UpdateUserView*). The *userViewConverter* converts the user resource to the user view defined in Listing 4.26. See Listing 4.30 for details.

```
1
2     export const createUserViewSchema = buildValidatorAndTransformer(CreateUserView)
3
4     export const updateUserViewSchema = buildValidatorAndTransformer(UpdateUserView)
5
6     export const userViewConverter = buildViewConverter(User, UserView)
7
```

Listing 4.29: Definition of the validation, transforming converting functions.

The **fifth step**, shown in Listing 4.31, is to implement the controller. Within this controller, the three different endpoints are registered. All endpoints have the same base path **/users** defined in the *@Controller* decorator. For each endpoint, the correct configuration must be given. So the *POST* endpoint must specify the consumed media type and the validation and transforming function for the request body. The *GET*

endpoint have to specify the produced media type and the view converter. Finally, the *PUT* endpoint must define the consumed media type, the viewConverter and also the validation and transforming function. In all methods, only the corresponding class is initialized and with the configure method the request and the response object are passed.

```
1
2   @Controller('/users')
3   export class UserController {
4
5     @Post<PostUserState>({
6       consumes: 'application/vnd.user+json',
7       schema: {
8         body: createUserViewSchema,
9       },
10    })
11    public postUser(req: HttpRequest, res: HttpResponse): Configured<PostUser> {
12      return new PostUserState().configure(req, res)
13    }
14
15    @Get<GetSingleUserState>({
16      produces: 'application/vnd.user+json',
17      viewConverter: userViewConverter,
18    })
19    public getSingleUser(
20    req: HttpRequest,
21    res: HttpResponse
22    ): Configured<GetSingleUser> {
23      return new GetSingleUserState().configure(req, res)
24    }
25
26    @Put<PutUserState>({
27      consumes: 'application/vnd.user+json',
28      schema: {
29        body: updateUserViewSchema,
30      },
31      viewConverter: userViewConverter,
32    })
33    public putUser(req: HttpRequest, res: HttpResponse): Configured<PutUser> {
34      return new PutUserState().configure(req, res)
35    }
36  }
37
```

Listing 4.30: UserController.

The **last step** is to register the controller and to start the application. Additionally, a base path can be defined, which is **/api** in the concrete example here.

```
1
2   async function main(): Promise<void> {
3     const app = new RestApplication({
4       prefix: '/api',
5     })
6
7     app.registerController(UserController)
8
9     await app.start()
10  }
11
12  main()
13
```

Listing 4.31: Exemplary RestApplication.

5. Evaluation

In this chapter we will evaluate the designed and implemented framework. Therefore, we will first conduct a survey among software developers and afterwards the framework is analyzed with the template created in section 3.2. Thereafter, the results of the survey and the template analysis are presetend together. Finally, we will discuss new research questions that come up during the analysis of the results.

5.1. Developer survey

The goal of the empirical investigation is to find out whether the developed framework actually supports the development of REST APIs. In addition, it will also be evaluated whether the developed APIs are more REST-compliant due to the very strict specifications of the framework, compared to APIs developed with other frameworks.

5.1.1. Setup

The survey consisted of several steps, which will be explained in the following. The first step was to have the **8** study participants fill out a questionnaire. They were asked about their knowledge of the technologies required for the study. One can see the questions and the results of the questionnaire below.

- How many years have you been working as a software developer?
- How many years of experience do you have in Node.js?
- How many years of experience do you have in TypeScript?
- How would you rate your knowledge of REST from 1 (very good) to 7 (no knowledge at all)?
- How often do you come across the buzzword REST in your daily work from 1 (every day) to 7 (never)?

Questionnaire 1 - Results					
	Years as developer	Experience Node.js	Experience TypeScript	Knowledge REST	REST as buzzword
$\bar{x}$	3.39	2.78	1.56	3.06	3.43
σ	1.25	1.10	1.15	1.66	2.57

Table 5.1.: Results of the initial questionnaire as mean value with corresponding standard deviation.

After this initial questionnaire, all participants attended a one hour workshop. In this workshop, the most important aspects of REST were presented to create a basic understanding of this topic among every participant. The slides of this workshop can be found in the appendix (section B). Especially hypermedia was one aspect of REST that was focused on within this workshop since it is the most confusing topic for developers.

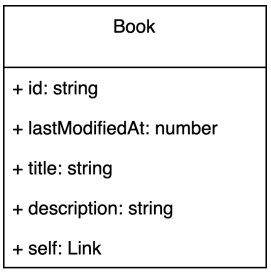


Figure 5.1.: Class diagram of the book resource class, used in the developer survey conducted in this work.

In the next step, the participants should work on a coding challenge. The attendees had to implement three different endpoints for the resource *Book* (see Figure 5.1). These endpoints enable a client to create, to get a single and to get a list of books. The different endpoints with the respective path and HTTP method and their requirements are briefly summarized below.

- **POST /books**:

  The consumed media type is *application/vnd.book+json* and if the client does not set the correct *Content-Type* header, the server should respond with the *415 Unsupported Media Type* status. Additionally, the server have to validate the incoming data. The incoming JSON object from the client should have a *title* and a *description* property. It should only be possible to create a *Book* resource, if the client sends a valid API key within the *x-api-key* header. The response from the server must contain the location header, which provides the client the URL to request the resource just created.

- **GET /book/:id**:

This endpoint should allow the client to request two different views on the *Book* resource with the corresponding media types *application/vnd.book+json* and *application/vnd.book-admin+json*. The more detailed view *AdminBookView* can be only received if the client sends a valid API key. If none or an invalid one is provided, the server should respond with *401 Unauthorized*. To obey the hypermedia constraint the server response should always contain a link to fetch the list of books. If the client's request contains a valid API key, the server should also send the links to update and delete the current book.

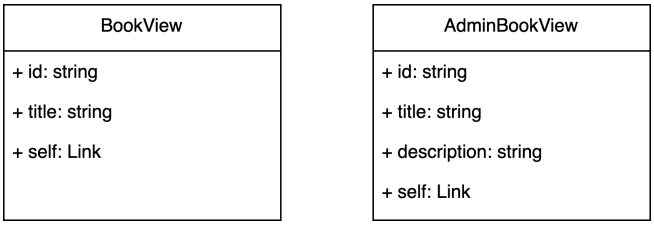


Figure 5.2.: The two different views on the books resource, used in the developer survey conducted in this work.

- **GET /books**:

  With this endpoint, the client can request a list of books. Thereby the endpoint produces the media type *application/vnd.user+json*. The returned view of the book has the same attributes as the *BookView* in Figure 5.2. The list of books has to be filtered by *title*, so the server should respond with *400 Bad Request* if the client does not send the corresponding query parameter. Additionally, an offset-size pagination strategy should be implemented. This also includes the generation of the links required to guide the client through the collection (see section 3.2 for a detailed explanation). The default *offset* should be zero and the default *size* ten. The server should inform the client about the number of results and the total number of results via the headers *x-totalnumberofresults* and *x-numberofresults*. The last requirement is that the server response must contain a link to create a new book resource.

This coding task was completed by the participants in two different groups. One group solved it with Express and the other one with the framework designed and implemented within the scope of this work. To reduce the time to solve this task, both groups were provided with a template project, that already contains the book model and the book data storage class. In addition, the HTTP server used in each case has already been configured for the participants. To give both groups a brief introduction to their framework, there was a video tutorial for both frameworks. There was also rudimentary documentation for the framework developed in this work.

To make the two groups comparable, they are divided so that there is an equal overall level of experience with the technologies used. Therefore an individual score was calculated for each participant. This score is simply an addition of the values of the first

three questions (*Years as developer*, *Experience Node.js* and *Experience TypeScript*) in the initial questionnaire. Participants were then divided so that both groups had a similar total value of this score. So the first group, using Express (**Group A**), had a score of **32** and the second one (**Group B**) a score of **29.8**.

After the developers completed the coding task, they had to fill out another final questionnaire. The corresponding questions can be seen below.

- How much time did it take you to complete the task?
- How many additional dependencies (npm modules) did you use?
- Looking back now after completing the task, how would you rate your REST knowledge before completing the task and workshop from 1 (very good) to 7 (no knowledge at all)?
- From 1 (very easy) to 7 (very hard), how easy did you find it to develop an API using REST principles?
- How REST compliant from 1 (everything is compliant) to 7 (not at all) do you rate your developed API?
- How would you rate your REST knowledge now (after the workshop and the task) from 1 (very good) to 7 (no knowledge at all)?
- How much did the framework you used from 1 (very supported) to 7 (not supported at all) help you develop REST compliantly?
- What was the most challenging problem during the development of the API?
- From 1 (very easy) to 7 (very hard) how challenging was it to implement the hypermedia principles?

The group that used the framework developed here got additional framework-specific questions.

- How did you feel about the very strict way the framework dictates?
- From 1 (not at all) to 7 (extremely time-consuming), how time-consuming do you estimate the familiarization with the framework and its documentation?
- Assume you are familiar with the framework and understand all the concepts: How effective from 1 (much more effective) to 7 (much slower) can you develop REST APIS with this framework compared to Express?

5.1.2. Results

Two different types of results are obtained. On the one hand, the data from the final questionnaire and on the other hand the data that can be retrieved from the analysis of the code submitted from the participants. First, the data from the questionnaire is presented and then an analysis of the submitted coding challenge results is given.

In the beginning, the survey had **8** participants, so **4** participants in each of the two groups. However, **3** participants did not finish the coding task after completing the initial questionnaire. So there were only **5** participants in total, **2** in group **A** and **3** in group **B**. Furthermore, 3 of the 5 participants stopped working on the task, before they finished it. Only one participant in each group has finished the task. Nevertheless, data from participants who did not complete the task were also considered.

The numerical results are presented as mean with corresponding standard deviation and the results of the open-ended questions are presented as the original text. It should be noted that the answers were translated from German to English. For the analysis of the submitted code, the focus is not on general coding errors, but on errors that lead to a non-REST and HTTP compliant API.

Group A

Questionnaire 2: Results Group **A** - Part 1				
	Required time	Additional dependencies	Retrospective knowledge	Ease of developing RESTful
$\bar{x}$	2.925	0.5	4.5	4
σ	1.29	0.71	2.12	1.41

Table 5.2.: Results of group A of the second questionnaire as mean with corresponding standard deviation - Part 1.

Questionnaire 2: Results Group **A** - Part 2				
	REST compliance	REST knowledge	Support from the framework to develop RESTful	Ease of hypermedia
$\bar{x}$	3	5	6.5	4
σ	0	0	0.71	0

Table 5.3.: Results of group A of the second questionnaire as mean with corresponding standard deviation - Part 2.

What was the most challenging problem during the development of the API?

– A clean division and separation of the different media types. So how do I distinguish between admin and normal users and how can I implement this most effectively and then really treat the data separately from each other.

– Thinking of everything and still passing on all the information needed in the right places.

The tasks like the validation of the incoming data or the extraction of the query parameters are done correctly. The biggest issue the participants had was the hypermedia aspect. For example, the formally correct use of the link header was a problem. Furthermore, although absolute URIs were used, the host part was hardcoded. So all links would have to be changed if the host changed. Most problems regarding hypermedia were encountered in the implementation of the collection resource. Thereby the creation of the links to guide the client through the collection and injection of the *self* link in every single resource are the major problems. Besides hypermedia, content negotiation was also a problem. The correct headers and status codes were not used.

Group B

Questionnaire 2: Results Group **B** - Part 1				
	Required time	Additional dependencies	Retrospective knowledge	Ease of developing RESTful
$\bar{x}$	3.82	0.33	4.67	3.33
σ	1.29	0.58	0.58	0.58

Table 5.4.: Results of group B of the second questionnaire as value with corresponding standard deviation - Part 1.

Questionnaire 2: Results Group **B** - Part 2				
	REST compliance	REST knowledge	Support from the framework to develop RESTful	Ease of hypermedia
$\bar{x}$	2.33	2.5	1.67	2.67
σ	0.58	0.5	0.58	2.08

Table 5.5.: Results of group B of the second questionnaire as mean with corresponding standard deviation - Part 2.

What was the most challenging problem during the development of the API?

– Lack of understanding - documentation is not perfect.

– Unfortunately, the documentation of the framework was not sufficient for the task. Although I was able to implement the task using the examples, unfortunately, I was not able to understand the execution flow of the state methods at any point in the documentation. Furthermore, the overriding of the endpoints based on the media type was not sufficiently documented, which is why I needed a further introduction.

– My biggest problem was the distinction between admin users and non-admin users, as I couldn't get the framework to check the API key. I got the admin user in both cases (even though I didn't send an API key) - but I think that's just a

configuration issue. Furthermore, I couldn't get the framework to send a 400 status code if I didn't include the "title" param in the GET Book param.

Questionnaire 2: Results Group **B** - Framework specific questions		
	Required familiarisation time	Effectiveness of supporting to develop RESTful compared to Express
$\bar{x}$	4.00	1.67
σ	1.00	0.58

Table 5.6.: Results of framework-specific questions from Group B as mean with corresponding standard deviation.

How did you feel about the very strict way the framework dictates?

- Personally, I like the fact that the framework provides a strict structure. Unfortunately, I missed a bit of flexibility or a description of the execution flow so that I could understand how my request was processed.
- Very good, the States approach is very easy to understand and I would like to use it in the future.
- It took some getting used to at first, because sometimes you didn't know directly why what you had just developed didn't work. I think with a longer period of use, however, you would understand this better (as with every tool). Otherwise, it makes sense that it is so strict - you want to develop a REST-compliant API.

The most frequent error made by the participants in group B is the incorrect use of the framework. Topics like content negotiation, HTTP compliant status codes or hypermedia are implemented correctly.

5.2. Template Analysis

In this section, the developed framework will be analyzed with the template created in section 3.2.

HM	P&F	C	CN	MT	E
++++	+++	+++	+++	++	-

Table 5.7.: Results of the template analysis of the framework developed in this work.

Hypermedia

The framework supports the developers to link resources between each other and to define the possible state transitions. Since the transitions must be defined within a single method, the developers are guided in which place they should define these. Additionally, the framework provides helper methods to define links and also constrained links, which are only added if a condition, for example the role of the current user, is fulfilled.

Pagination & Filtering

The developers are forced to consider pagination by the abstract *AbstractGetCollection-State* class. Furthermore, the framework provides the offset-size pagination strategy built-in. However, the developer is not supported in the creation of search template, but only in the extracting of query parameters.

Caching

Via the *AsbtractStateWithCaching* the developers have the capability to define the caching behaviour via the *Cache-Control*, *Last-Modified* and *ETag* headers. Additionally the base states *AbstratPostState*, *AbstratPutState* and *AbstratDeleteState* implements conditional requests.

Content Negotiation

There is good support for content negotiation. The developers are able to define different route handlers for the same route.

Media Types

The framework has built-in support for any JSON based media type and *plain/text*. To support other media types, the underlying *Fastiy* API can be used.

Extensibility

Since the framework is very opinionated and it enforces a very strict way how to implement a REST API, it is hard to implement an API that does not follow this strict way. Nevertheless, the developers can override any step of the execution flow within the template method. With hooks, they can even implement any kind of logic between these steps.

5.3. Discussion

As one can see in the template analysis in section 5.2 the framework supports the developers in complying with the REST constraints. This can be also seen in the analysis of the submitted code. While group **A** has problems implementing the REST principles, group **B** only has problems familiarising itself with the framework. The reason for this good support for RESTful development could be the very strict approach of the framework. Even though this strict approach is very uncommon compared with other frameworks, the participants were very positive about this approach. One participant even wants to continue using the class-based approach with the template pattern.

Hypermedia is one of the most difficult topics in REST, which makes the answers to the question "How challenging was it to implement the hypermedia principles?" very interesting. Thereby, one sees a lower value for group **B** than for group **A**, which means

for the participants of group **B** it was easier to obey the hypermedia principle. This is an indication that the framework helps developers to implement even the difficult issues of REST.

The time needed to solve the task is not considered here, as more than half of the participants did not solve the task completely.

5.4. Future Research Opportunities

In the following section, new research questions that occurred during the analysis of the data will be discussed.

As already mentioned, the approach with the state classes is well accepted by the developers. Additionally, the developers from group **A** had structural issues while developing the API. These two circumstances could be indicative of another possible problem in the development of REST APIs. This problem may be not caused by some misunderstanding of the REST constraints but by an architectural issue. A common way to structure the code is that one method (often placed in a so-called *Controller*) is registered that receives the HTTP request. Within this method, all required information is extracted. With this information, a service method is called. This service method, usually placed in a so-called service class, contains the business logic. On the one hand, it is very hard to avoid repetitive tasks with this approach. On the other hand, since the data that is processed in the business logic is also required in several other places, like link creation or conditional requests, it is very hard for the developers to know in which places they need which information. This can be observed in the answers to the open-ended question from group **A**. Both problems could be solved by the class-based approach used by the framework designed in this work.

Although the participants from group **B** had problems getting used to the framework, the results of this group were overall significantly better. One reason for this may be that group **A** participants would need to have a better understanding of REST to produce the same results as group **B** while the participants of group **B** would only have to learn the concepts of the framework and not the difficult-to-understand concepts of REST. So it could be easier for developers to learn a new very concrete concept in the form of a new framework than the very abstract concepts of an architectural style, namely REST. Additionally, since the familiarization problems with the framework are caused by the lack of documentation, the problems of **B** can be solved more easily by just providing more complete documentation.

6. Conclusion

To conclude this work, a summary of all the results will be given. Afterwards, there is an outlook on possible future features of the framework. Afterwards, the problems that occurred during the survey are discussed.

6.1. Summary

The goal of this work was the design and implementation of a Node.js framework for RESTful API development. More precisely, this work first presented the current problems that require the creation of another framework. Next, the design of this framework was explained. The implemented framework was then evaluated through a developer survey, which verifies that the framework enables the developers to implement APIs REST compliant. Before all this, there was an introduction to the topics that are necessary to understand the work. These include the REST constraints and the basics of Node.js and TypeScript.

Several reasons have been given that make it so difficult for developers to implement REST APIs. These include the fact that the style and level of abstraction, in which Fielding's thesis [5] is written, makes it very hard for developers to understand the concepts [19, p. 343]. Furthermore, there is a lack of tools that could effectively support developers on developing RESTful [20]. To illustrate this circumstance in the Node.js ecosystem, an analysis of several existing frameworks was presented. Thereby the frameworks were analysed concerning different aspects, that are important for the development of REST APIs. As a consequence, it is very hard for developers, even regardless of whether or not they have understood the REST concepts, to implement Web APIs REST compliant and at the same time effectively.

To overcome this issue, a Node.js framework has been proposed that on the one hand should support the developers to implement REST APIs more effectively and on the other hand, makes it harder for them to violate the REST constraints than to comply with them. Therefore multiple framework design characteristics were presented. These design characteristics are intended to enforce the users to develop REST and HTTP compliant and to give as few opportunities as possible to make mistakes in the first place. The most characteristic one is the concept of state templates. Each state class represents a single endpoint that corresponds to a specific path, HTTP method and resource type. These state templates provide an effective way to refactor common tasks

in such a way, that developers do not have to handle these over and over again. With abstract methods, it is also possible to enforce developers to consider important topics like the definition of the next possible state transitions. Besides the state templates, the HTTP router was also introduced, which can differentiate between endpoints depending on the media type the endpoints produces and/or consumes.

Afterwards, the implemented framework was evaluated with a developer survey. For this purpose, one part of the participants had to develop a REST API with Express and the other ones with the framework implemented in this work. The results from this survey indicated that the framework enables developers to develop REST APIs significantly more effective and less error-prone. According to the results of the survey, the framework also makes it easier to handle hypermedia, which is one of the most difficult concepts of REST.

6.2. Outlook

After a summary of this work, an outlook on possible further features of the framework is given. This is followed by a discussion of the problems that occurred during the survey.

6.2.1. Framework

Considering the situation where the users of the framework want to describe their developed REST API with an OpenAPI[1] document. Currently, they have to create and maintain this document manually. Since JSON schemas are used to describe the incoming and outgoing data, the framework actually already has access to the information required to generate an OpenAPI document from it. This makes it unnecessary to manually maintain OpenAPI documents. So one could implement a new framework component that generates the OpenAPI document based on the JSON Schema provided respectively generated.

At this moment, the framework allows developers to define different views of the same resource, which are always sent to the client in JSON syntax. Since clients might also request HTML or XML representations, the framework should support the developers to convert the views to such formats. In addition, the framework could also provide support for binary data. This feature would enable the client to request images for example.

Even though the framework allows the developers to register multiple endpoints with the same path but with different producing or consuming media types, this is just possible within the same controller class. One use case where this could be problematic is

[1] https://www.openapis.org/

versioning. Versioning should be done by defining different media types for the different versions. So a versioned endpoint returns different media types based on the client's request. But these endpoints must be implemented in the same controller class, so it is not possible to create different controllers for different versions which, however, would be much cleaner. Accordingly, the framework should support this approach in the future.

6.2.2. Survey

During the survey, different problems occurred which are discussed in the following. The first one was the challenge to find enough participants for the survey. In contrast to normal surveys, where there are often no requirements for participants, the participants of this survey had to have experience in software development on the one hand, and also experience in Node.js and TypeScript on the other hand. This restricts the number of possible participants significantly. In addition, as outlined in subsection 5.1.1, only 5 of the original 8 participants then took part in the coding task and only 2 completed the task. The reason for this may be a large amount of time required to work on the task. Thus, developing a concept for recruiting participants and for minimizing the time required for coding tasks would be a possible research topic.

List of Figures

List of Tables

Listings

Bibliography

[1] Mario Casciaro. *Node.js Design Patterns - Second Edition.* 2nd ed. Packt Publishing, 2016.

[2] Brajesh De. *API Management. An architect's guide to developing and managing APIs for your organization.* 1st ed. Apress, 2017.

[3] Steven Fenton. *Pro TypeScript.* 2nd ed. Apress, 2018.

[4] Doglio Fernando. *REST API Development with Node.js.* 2nd ed. Apress, 2018.

[5] Roy T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures". dissertation. University of California, Irvine, 2000.

[6] Roy T. Fielding. *REST APIs must be hypertext-driven.* Oct. 20, 2008. URL: `https://roy.gbiv.com/untangled/2009/it-is-okay-to-use-post` (visited on 04/22/2021).

[7] Roy T. Fielding, Mark Nottingham, and Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Caching.* RFC 7234. 2014. URL: `https://rfc-editor.org/rfc/rfc7234.txt`.

[8] Roy T. Fielding and Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests.* RFC 7232. 2014. URL: `https://rfc-editor.org/rfc/rfc7232.txt`.

[9] Roy T. Fielding and Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content.* RFC 7231. 2014. URL: `https://rfc-editor.org/rfc/rfc7231.txt`.

[10] Roy T. Fielding et al. *URI Template.* RFC 6570. 2012. URL: `https://rfc-editor.org/rfc/rfc6570.txt`.

[11] Adam Freeman. *Essential TypeScript.* 1st ed. Apress, 2019.

[12] Erich Gamma et al. *Design Patterns.* 1st ed. Addison-Wesley professional computing series, 1994.

[13] Dirk W. Hoffmann. *Theoretische Informatik.* 4th ed. Hanser, 2018.

[14] Wallace Jackson. *JSON Quick Syntax Reference.* 1st ed. Apress, 2016.

[15] Christoph Meinel and Harald Sack. *WWW.* 1st ed. Springer, 2004.

[16] *Node.js Documentation.* URL: `https://nodejs.org/api/` (visited on 06/30/2021).

[17] Mark Nottingham. *Web Linking*. RFC 5988. 2010. URL: `https://rfc-editor.org/rfc/rfc5988.txt`.

[18] Sandro Pasquali. *Mastering Node.js*. 1st ed. Packt Publishing, 2013.

[19] Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs. Services for a Changing World*. 1st ed. O'Reilly, 2013.

[20] Vitaliy Schreibmann and Peter Braun. "Model-driven Development of RESTful APIs." In: *WEBIST*. 2015, pp. 5–14.

[21] Stefan Tilkov and Steve Vinoski. "Node.js: Using JavaScript to Build High-Performance Network Programs". In: *IEEE Internet Computing* 14.6 (2010), pp. 80–83.

[22] Stefan Tilkov et al. *REST und HTTP*. 3rd ed. dpunkt.verlag, 2015.

[23] *TypeScript*. URL: `https://www.typescriptlang.org/` (visited on 06/29/2021).

[24] *TypeScript Documentation*. URL: `https://www.typescriptlang.org/docs/` (visited on 06/28/2021).

[25] Frank Zammetti. *Modern Full-Stack Development*. 1st ed. Apress, 2020.

Appendices

A. Code Listings

```

class User extends AbstractModel {
  @modelProp()
  name: string

  @modelProp()
  age: number

  @modelProp()
  email: string
}

class UpdateUserView extends AbstractViewModel {
  @viewProp()
  name: string

  @viewProp()
  age: number

  @viewProp()
  email: string
}

const user = new User()
user.id = 13
user.name = 'Matthias'
user.age = 22
user.email = 'student@mail.de'

const userView = new UpdateUserView()
userView.name = 'Matthias'
userView.age = 23

/*
User { id: 13 name: 'Matthias', age: 23, email: undefined }
*/
merge( userView, user )
```

Listing A.1: Exemplary use of the merge method.

B. REST Workshop Slides

REST UND HTTP

- Ein Schnelldurchlauf -

1

ICH DACHTE DAS IST REST?

```
POST /getUsers HTTP/1.1
Host: rest-love.com
Accept: application/json
```

- Das widerspricht nicht nur REST sondern grundsätzlich dem darunterliegendem HTTP Protokol
- "POST is used for the following functions (among others): [FieldingRFC7231]
  - […]
  - Creating a new resource that has yet to be identified by the origin server

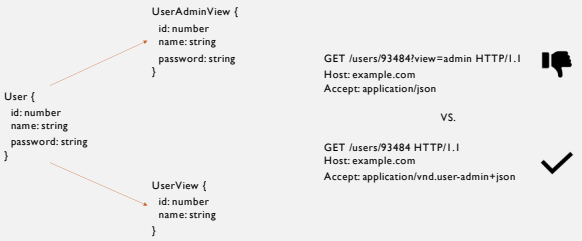
2

OVER- AND UNDER-FETCHING

- Over-fetching: Meine Repräsentation enthält Daten, die ich im Frontend nicht brauche
- Under-fetching: Meine Repräsentation enthält nicht alle Daten, die ich im Frontend brauche → ich muss zuätzliche Requests machen
- Oft gelesene Aussage: REST kann dieses Problem nicht lösen, aber GraphQL kann das
- ABER: REST liefert in seiner Defintion eine Lösung für dieses Problem
- Einfach neue Repräsentation definieren und dem Client zur Verfügung stellen

3

UNTERSCHIEDLICHE VIEWS



4

(REST OVER) HTTP

- HTTP = TCP basierendes zustandsloses Protokoll (Neuste Version HTTP/2 in RFC 7540)
- REST = Architekturvorschlag für verteilte Hypermedia Systeme
- REST != HTTP
- ABER: REST/ HTTP Object Model hat die Entwicklung des Protokolls beeinflusst, da Roy Fielding an der Spezifikation des Protokolls beteiligt war [Tilkov]

5

CONSTRAINTS [FIELDING]

1. Client-Server
2. Stateless: Alle Zustandsinformationen werden beim Client gehalten
3. Cache: Der Inhalt einer Server-Antwort muss als cachbar oder nicht cachbar ausgezeichnet werden
4. Uniform Interface: HATEOAS, self-descriptive messages, manipulation of resources through representation of resources, identification of resources
5. Layered System: Zwischen Client und Server können beliebig viele Layer gezogen werden (z.B. Load-Balancer, Reverse Proxy, etc.)
6. (Code-on-Demand)

6

GRUNDPRINZIPIEN [TILKOV]

1. Ressourcen mit eindeutiger Identifikation: Alles was "benannt" werden kann ist eine Ressource und eine eindeutige ID + URI/URL
2. Hypermedia: HATEOAS
3. Standardmethoden: Jede Ressource unterstützt die gleichen Methoden (GET, POST, DELETE, PUT, etc.)
4. Unterschiedliche Repräsentationen: Unterschiedliche Repräsentationen für unterschiedliche Anwendungsfälle
5. Statuslose Kommunikation: Zustand wird von Client gehalten und bei jedem Request an Server gesendet; Zustand kann z.B aber auch in Ressource verwandelt werden (Einkaufskorb)

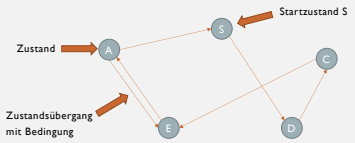
7

KONKRETE IMPLEMENTIERUNG

- DISCLAIMER: Es gibt nicht DIE EINE Art und Weise RESTful API zu implementieren (HAL,Collection+JSON, etc.)
- Wir konzentrieren uns hier auf eine Sichtweis, wie man REST umsetzen kann
- ABER: Hypermedia/ HATEOAS ist ein MUSS → ohne HATEOAS ist es keine REST API, was ja nicht schlimm ist, da es "nur" ein Architekturvorschlag ist

8

ENDLICHER AUTOMAT



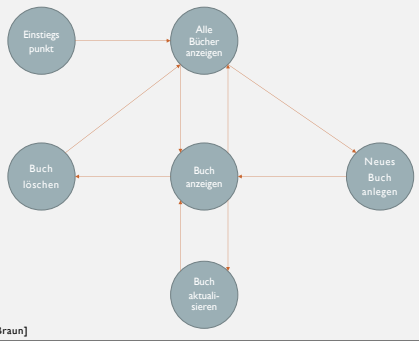
9

APPLIKATIONEN ALS ENDLICHER AUTOMAT

- Bsp.: Bibliothek/ Bücherverwaltung
- Zentrale Ressource = Buch/ Book
- Für die Ressource Book sollen alle möglichen Zustände und Zustandsübergänge modelliert werden
- Client muss nur eine URL kennen (= Startzustand): Alle anderen Übergänge erhält Client von Server

```
Book {
  id: string
  lastModifiedAt: number
  title: string
  description: string
}
```

10



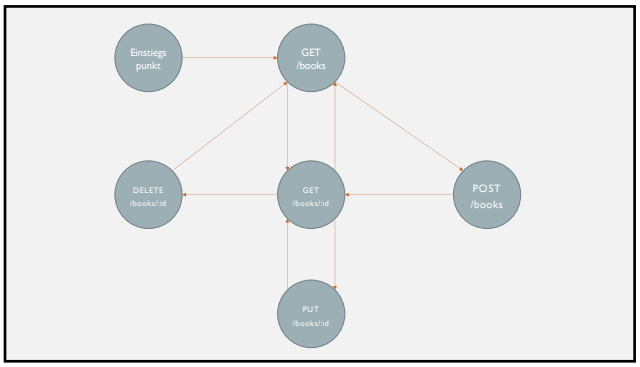
Inspiriert durch [Braun]

11

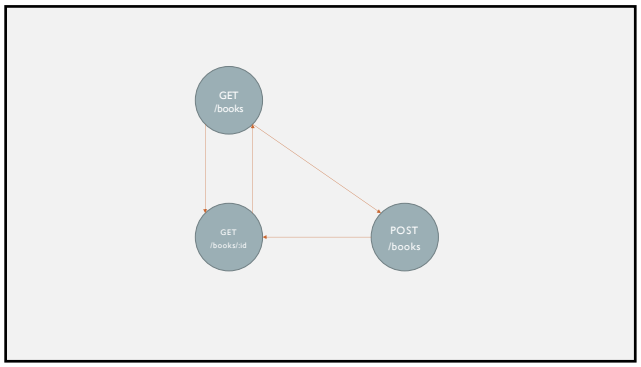
HTTP VERBEN

- GET: Verwendet um Repräsentationen von Ressourcen vom Server anzufragen
  GET /books/:id
  GET /books
- POST: Verwendet um neue Ressourcen zu erstellen
  POST /books
- PUT: Verwendet um Ressource zu erzeugen/ upzudaten
  PUT /books/:id
- DELETE: Verwendet um Ressourcen zu löschen
  DELETE /books/:id

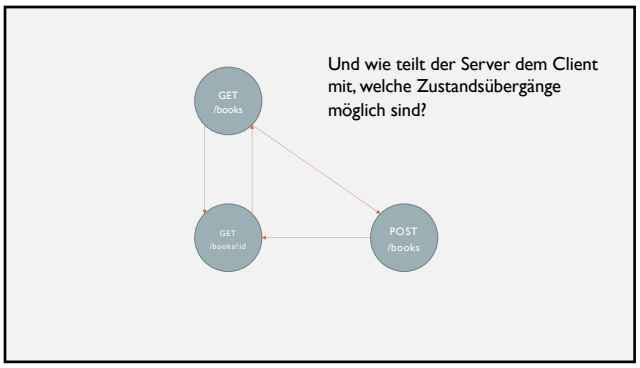
12



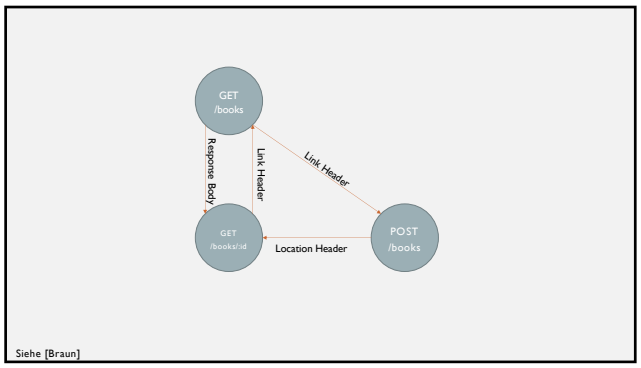
13



14

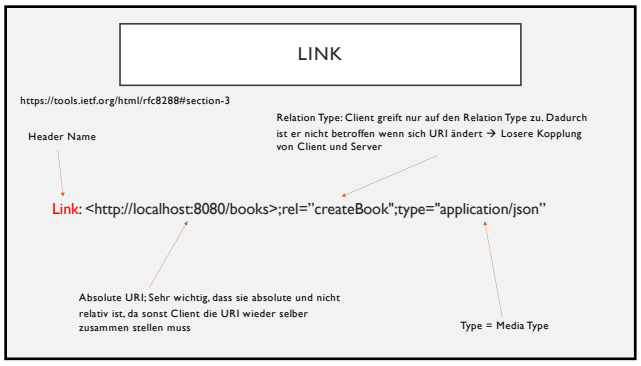


15



16

17



18

GET /books/d9sabh HTTP/1.1
Host: localhost:8080

→

HTTP/1.1 200 OK
ETag:"2b-DsZOsh0dOp30NrXYgSUJa+hWheM"
Cache-Control: public, max-age=86400
Content-Length: 382
Content-Type: application/json
Link: <http://localhost:8080/books>;rel="getAllBooks";type="application/json"

19

POST /books HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Content-Length: 34

{
"title": "HTTP für Dummies"
}

→

HTTP/1.1 201 CREATED
Location: http://localhost:8080/books/faf78sG

20

GET /BOOKS/:ID

1. Client fragt gewünschte Repräsentationsformat via Accept Header z.B.: "appliation/json;q=0.8,application/xml;q=0.4" an
   - Überprüfe ob angefragte Repräsentation vom Server zur Verfügung gestellt werden kann → 406 Not Acceptable
   - Entscheidung kann auch von den Rechten des Clients beeinflusst werden
2. Authorization/ Authentication Check (→ 401 Unauthorized, 403 Forbidden, 404 Not Foud)
3. Überprüfung ob angefragte Ressource existiert (→ 404 Not Found)
4. Kennt der Client bereits den aktuellen Zustand der Ressource?
   - Überprüfe ob Client "If-none-match" Header mitgesendet hat, wenn ja vergleiche ihn mit dem Hashwert der aktuellen Ressource; wenn identisch → 304 Not Modified
   - Überprüfe ob Client "If-modified-since" Header mitgesendet hat, wenn ja vergleiche Datum mit lastModifiedAt Attribut der Ressource → Wenn Client-Datum später liegt → 304 Not Modified
5. Wandle Ressource in View/Repräsentation um, die für Client "bestimmt ist" (zum Beispiel nicht alle Attribute der Ressource oder unterschiedliche embedded Links basierend auf der Rolle des Clients)
6. Definiere Caching-Strategie
   - Setze Cache-Control oder Expires Header (Expirationsmodell)
   - Generiere Hashwert der Ressource und setze dementsprechend den ETag Header (Validierungsmodell)
7. Definiere alle möglichen Zustandsübergänge/ Transitionen ausgehend vom aktuellen Zustand (Link Headers, Response Body)
   - Oft auch abhängig von der Rolle des Clients → meist recht zeitaufwändig, aber Aufwand lohnt sich!

21

→ In unserem konkreten Beispiel sollten wir folgende Links mit senden:

- Link um aktuelle Ressource zu löschen (aus Bibliotheksbestand zu entfernen)
- Link um aktuelle Ressource zu aktualisieren (neuere Edition z.B.)
- Link zu Collection Ressoure "/books"
  - Besonderheit dabei ist, dass wir dem Client mitteilen müssen wie wir er die Collection filtern/ durchsuchen kann, da wir ihm nicht die Möglichkeit geben alle Bücher auf einmal anzufragen
  - In unserem konkreten Fall kann unser Client mit dem Buchtitel filter und mit Offset/Size-Pagination durch die Collection gehen
  - Linkgenerierung für die Pagination schauen wir uns im nächsten Fall an
  - ABER: Initialen Template Link müssen wir hier dem Client zur Verfügung stellen
  - http://localhost:8080/books/?title={bookTitle}
  - Das ist wichtig, da sonst der Client die "Engine of Application State" ist, was den REST Contraints widerspricht

22

GET /BOOKS MIT OFFSET/SIZE PAGINATION

1. Client fragt gewünschte Repräsentationsformat via Accept Header z.B.: "appliation/json;q=0.8,application/xml;q=0.4" an
   - Überprüfe ob angefragte Repräsentation vom Server zur Verfügung gestellt werden kann → 406 Not Acceptable
   - Entscheidung kann auch an den Rechten des Clients beeinflusst werden
2. Authorization/ Authentication Check (→ 401 Unauthorized, 403 Forbidden)
3. Wandle Ressourcen wieder in die gewünschte View um
4. Collection Ressourcen sollten nicht gecacht werden, da bei diesen sowieso meist ein Cache Missmatch auftreten würde (wir setzen z.B. den Cache-Control Header auf "no-cache, no-store")
5. Definiere alle Links mit dem der Client weiter die Collection durchsuchen kann (Link Header und nicht Response Body)
6. Definiere alle möglichen Zustandsübergänge/ Transitionen ausgehend vom aktuellen Zustand (Link Headers, Response Body)
   - Oft auch abhängig von der Rolle des Clients → meist recht zeitaufwändig, aber Aufwand lohnt sich!

23

→ In unserem konkreten Beispiel sollten wir folgende Links mit senden:

- Link um neue Book Ressource zu erzeugen (natürlich nur wenn der aktuelle Nutzer die Berechtigung hat; in unserem Beispiel wird das nur der Fall sein wenn er einen validen API-Key mit sendet)
- Es gibt unterschiedliche Arten zur Pagination; Wir nutzen hier Offset/Size da sie einfach zu verstehen und zu implementieren ist
- Alle Links mit dem der Client weiter die Collection durchsuchen kann; Bsp.:
  - Initialer Request mit "localhost:8080/books?title=REST&size=10&offset=20"
  - Da die ersten 20 Einträge übersprungen worden sind gibt es auf jeden fall vorherige Einträge
    → Link "localhost:8080/books?title=REST&size=10&offset=10" mit Relation Type "prev"
  - Wenn es mehr als 30 Einträge gibt dann muss auch ein Link für das nachfolgende Fenster erzeugt werden
    → Link "localhost:8080/books?title=REST&size=10&offset=30" mit Relation Type "next"

24

POST /BOOKS

1. Client sendet Media Type seiner Repräsentation im Content-Type Header mit; Server checkt ob dieser Media Type unterstützt ist und verwendet werden kann um Ressource anzulegen → 415 Unsupported Media Type
2. Authorization/ Authentication Check (→ 401 Unauthorized, 403 Forbidden)
3. Ressource wird serverseitig angelegt und erhält eindeutige ID
4. Setze Location Header mit der URI, die auf die neu angelegte Ressource zeigt, also z.B
   → http://localhost:8080/book/38kdf80
5. Antwort sollte keinen Payload beinhalten um so Client zu zwingen einen GET auf die Location URI zu machen → Cache wird befüllt
6. Definiere alle möglichen Zustandsübergänge/ Transitionen ausgehend vom aktuellen Zustand (Link Headers, Response Body)
   - Oft auch abhängig von der Rolle des Clients → meist recht zeitaufwändig, aber Aufwand lohnt sich!

25

(CUSTOM) MEDIA TYPES [BRAUN]

- Media Types beschreiben die Repräsentationen der Ressourcen
- Jeder kennt die Standard Media Types: application/json, text/plain, etc.
- application/json beschreibt den Syntax der Repräsentation aber macht keinerlei Aussage über den Inhalt/ die Semantik
- Daher bietet es sich an anwendungsspezfische Media Types zu definieren
- Bei unserem Beispiel z.B. application/vnd.book+json
- Client kann dem Server sehr spezifisch mitteilen, welche Repräsentation er gerne hätte → Dafür verwendet wird die sogenannte Content Negotiation

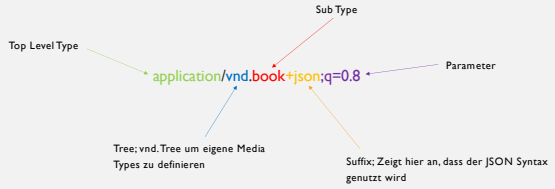
  Bsp.: Accept: application/json;q=0.9, text/plain;q=0.8, text/*;q=0.5, */*;q=0.1

  → Übersetzt: Am liebsten möchte der Client eine Repräsentation in JSON Format. Wenn es das nicht gibt, dann bitte in plain text Format. Sollte es das auch nicht geben, dann bitte in irgendeinem Text Format und als Notfalllösung, falls es auch das nicht gibt, dann nehme ich jedes Format.

26

AUFBAU VON MEDIA TYPES

Beispiel von gerade:



27

VORTEILE UND NACHTEILE + USE CASES

- Media Type drückt jetzt Syntax UND Semantik aus
- Unterstützung bei OpenAPI/Swagger
- Allerdings muss der Client auch die anwendungspezifischen Formate verarbeiten können
- Unterschiedliche Views auf die selbe Ressource realisieren:
  - application/vnd.book+json
  - application/vnd.book-admin+json
- API Versionierung:
  - application/vnd.book.v1+json
  - application/vnd.book.v2+json

28

"REST doesn't eliminate the need for a clue. What REST does is concentrate that need for prior knowledge into readily standardizable forms." *[FieldingHypertext-Driven]*

29

EURE AUFGABE

- Zentrale Resource **Book**
- Um diese Resource soll eine kleine CR(UD) REST API geschrieben werden
- Das Model Book sowie die Datenbankschicht sind bereits implementiert und sollen als Grundlage genutzt werden
- Die Zeit die für das Entwicklen benötigt wird, soll dabei gemessen werden
- Nach der Entwicklung werden dann noch einige weitere Fragen gestellt
- Finale API wird dann darauf untersucht, ob alle REST Constraints eingehalten worden sind

```
export class Book {
  public id: string

  public lastModifiedAt: number

  public title: string

  public description: string

  public self: Link
}
```

30

FEATURES

- Validation mit beliebigem npm module (bei express.js)
- Endpunkt um neues Buch zuerstellen → POST /books
- Endpunkt um alle Bücher mit bestimmten Titel anzuschauen → GET /books
- Endpunkt um einzelnens Buch anzuschauen → GET /books/:id

31

POST /BOOKS

- Media Type: "application/vnd.book+json"
- Nur erlaubt wenn valider API Key mitgesendet wird (x-api-key: s3cr3t)

```
CreateBookView {
  title: string
  description: string
}
```

32

GET /BOOKS/:ID

- 2 verschiedene Views/ Media Types
- "application/vnd.book+json", "application/vnd.book-admin+json"
- Admin View kann nur zurück gegeben werden, wenn valider API Key mitgesendet wird
- Link zu "/books" soll immer in Antwort enthalten sein
- Links um das Buch zu löschen bzw. upzudaten sollen nur enthalten sein, wenn Nutzer einen gültigen API Key mit sendet

```
AdminBookView {
  title: string
  id: string
  description: string
  self: Link
}

BookView {
  title: string
  id: string
  self: Link
}
```

33

GET /BOOKS

- Media Type: "application/vnd.book+json"
- Bücher müssen mit "title" gefiltert werden → Wenn nicht gegeben 400 Bad Request
- Offset-Size Pagination
- Link um neuen Buch zu erstellen in Antwort vorhanden

```
BookView {
  title: string
  id: string
  self: Link
}
```

34

LITERATUR

- [Tilkov] S. Tilkov et al. , REST und HTTP. 3rd ed. dpunkt.verlag, 2015
- [FieldingRFC7231] R. Fielding et al. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231. June 2014. URL: https://tools.ietf.org/html/rfc7231
- [Fielding] R. Fielding. "Architectural Styles and the Design of Network-based Software Architectures". Dissertation. University of California, Irvine, 2000.
- [Braun] Peter Braun, Parallele und Verteilte Systeme, 2019. URL: https://elearning.fhws.de/course/view.php?id=12273
- [FieldingHypertext-Driven] https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven; abgerufen am 03.05.21 um 19:54 Uhr

35

C. Class Diagrams

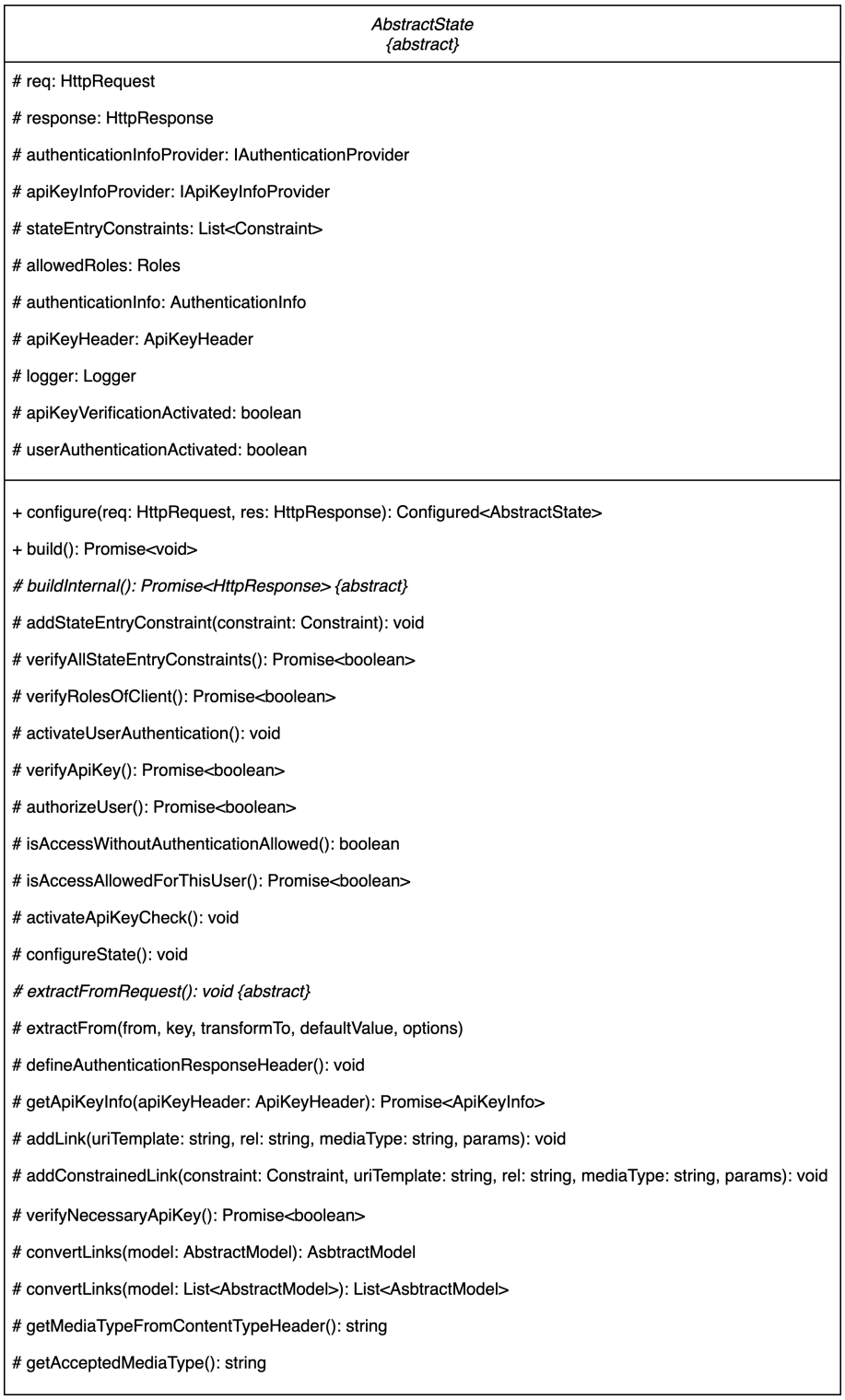


Figure C.1.: Class diagram AbstractState.

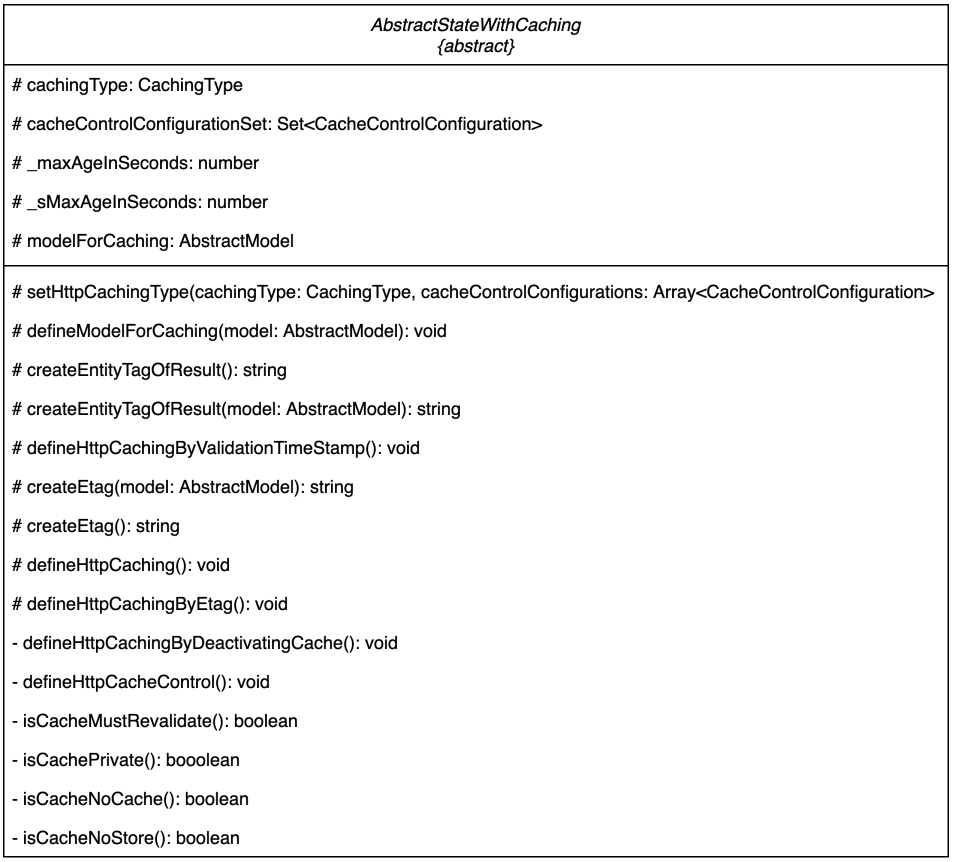


Figure C.2.: Class diagram AbstractStateWithCaching.

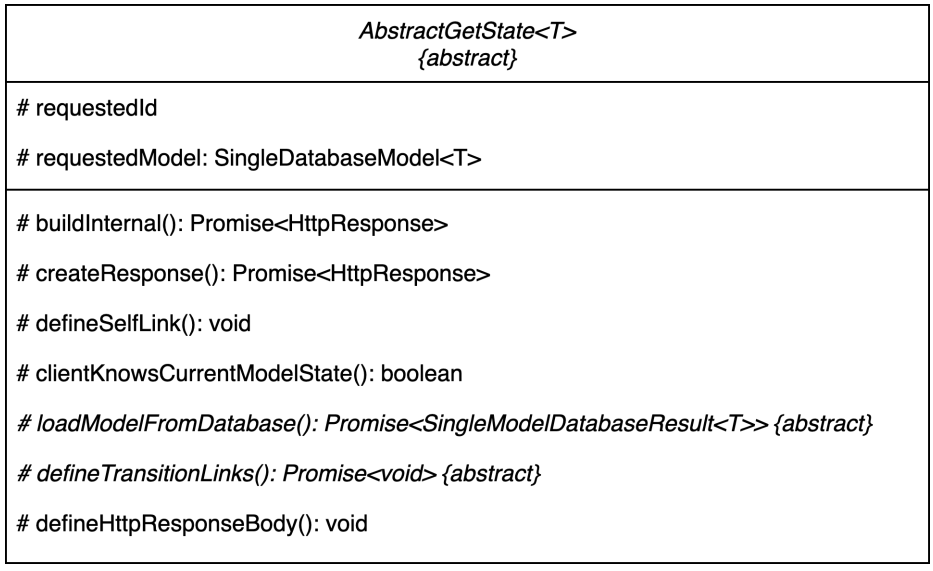


Figure C.3.: Class diagram AbstractGetState.

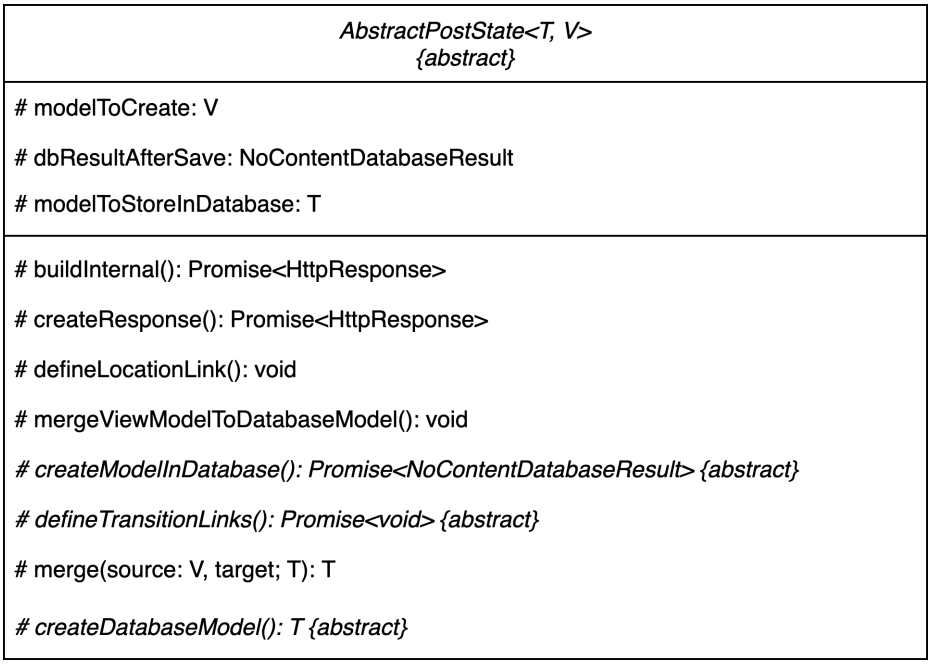


Figure C.4.: Class diagram AbstractPostState.

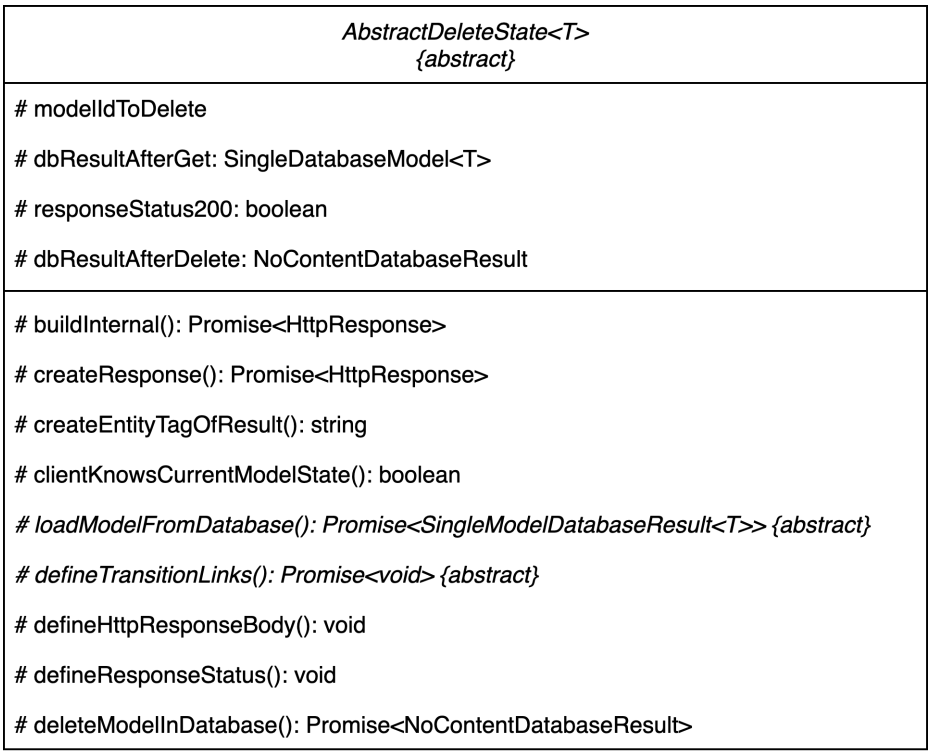


Figure C.5.: Class diagram AbstractDeleteState.

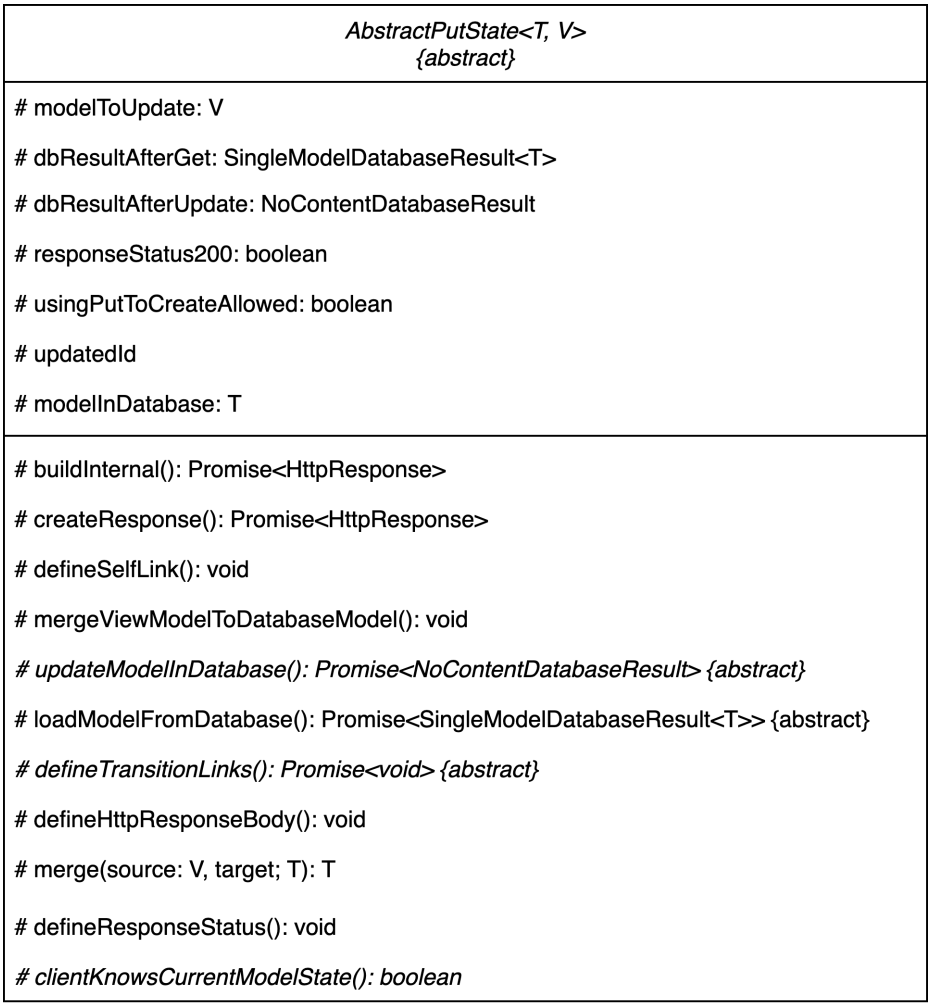


Figure C.6.: Class diagram AbstractPutState.

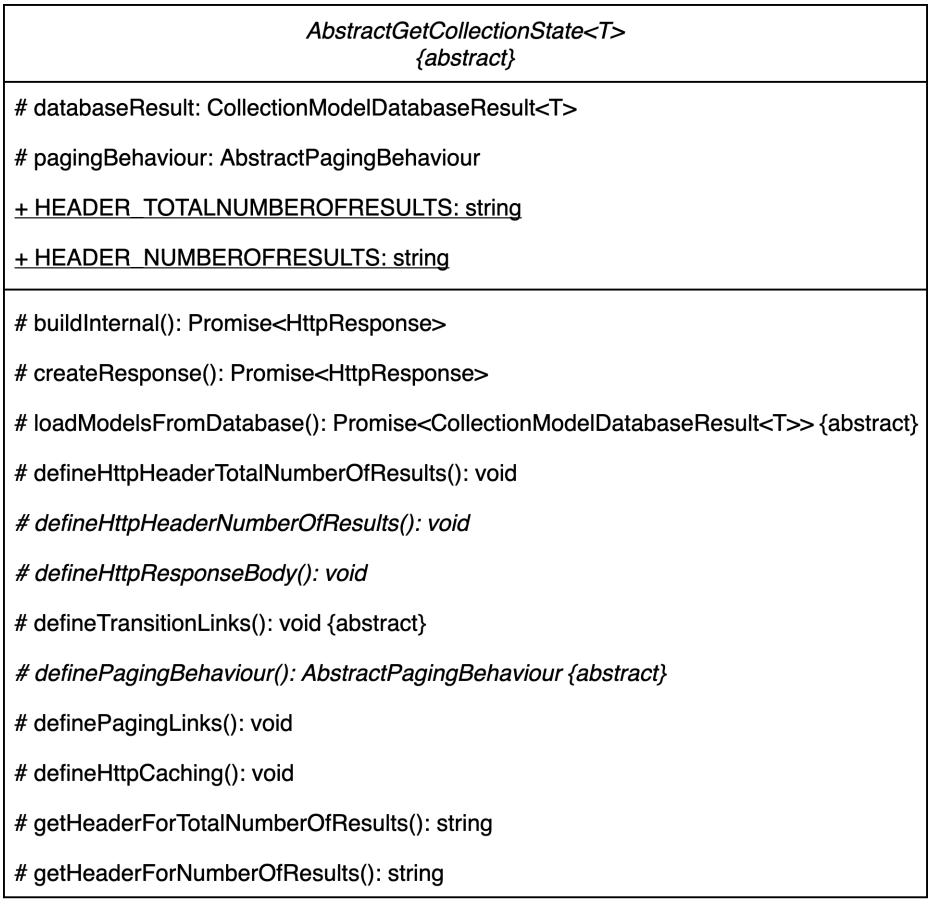


Figure C.7.: Class diagram AbbstractGetCollectionState.

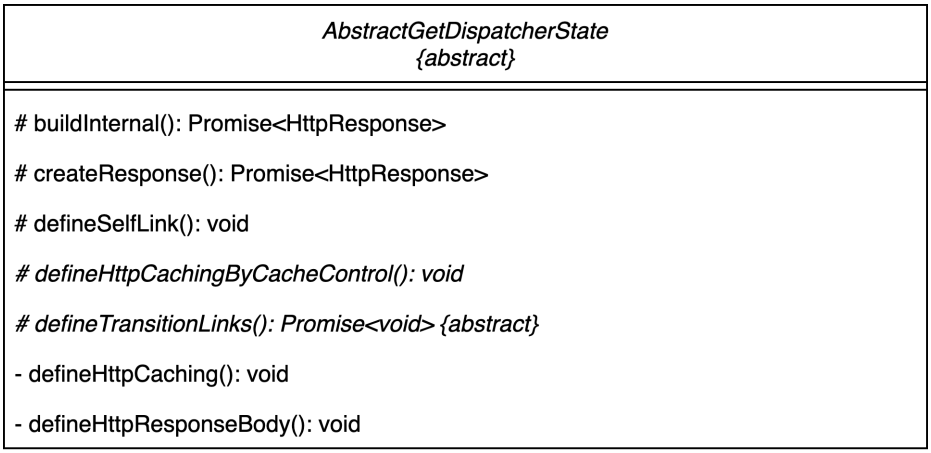


Figure C.8.: Class diagram AbstractGetDispatcherState.

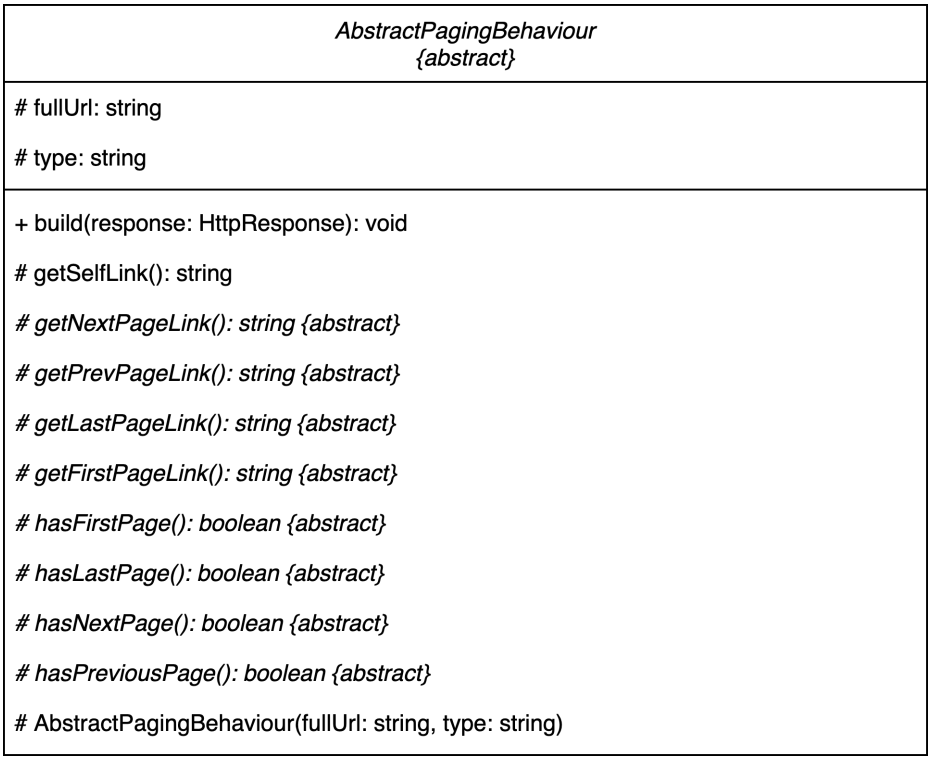


Figure C.9.: Class diagram AbstractPagingBehaviour.

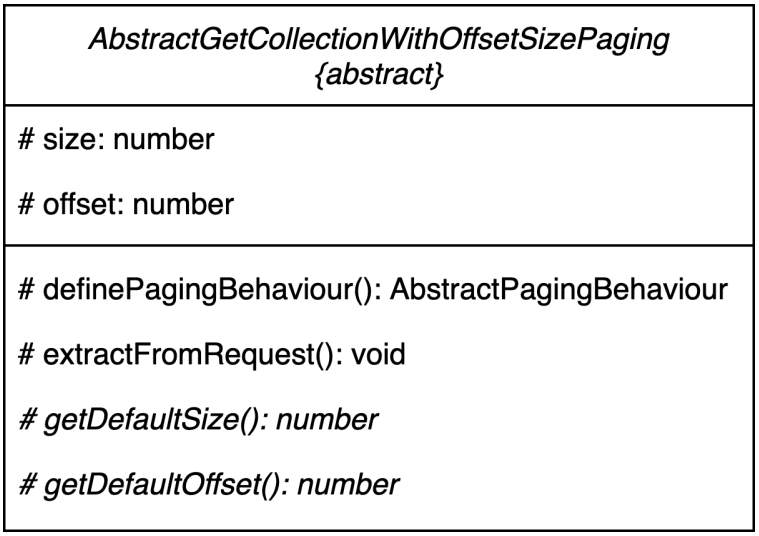


Figure C.10.: Class diagram AbstractGetCollectionWithOffsetSizePaging.

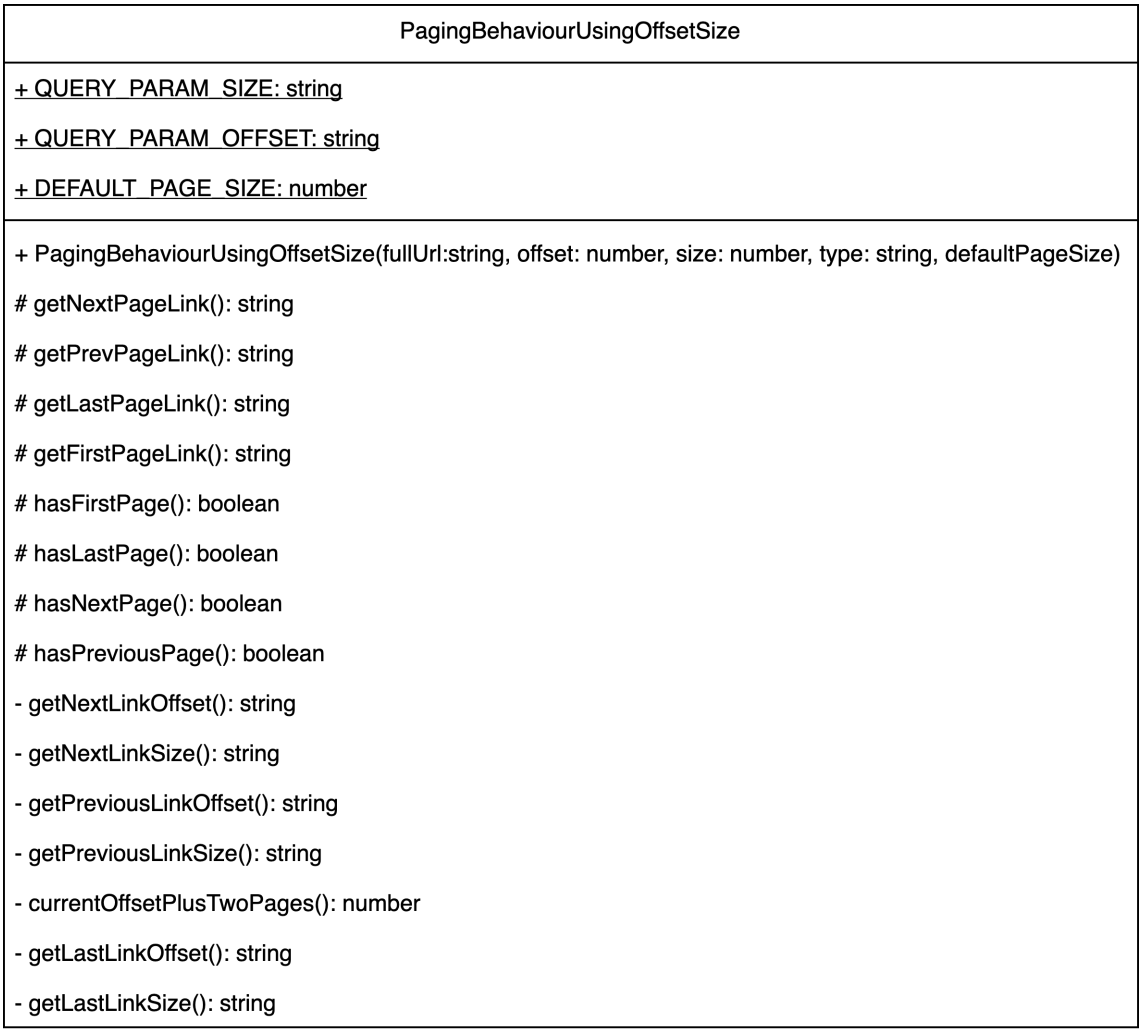


Figure C.11.: Class diagram PagingBehaviourUsingOffsetSize..

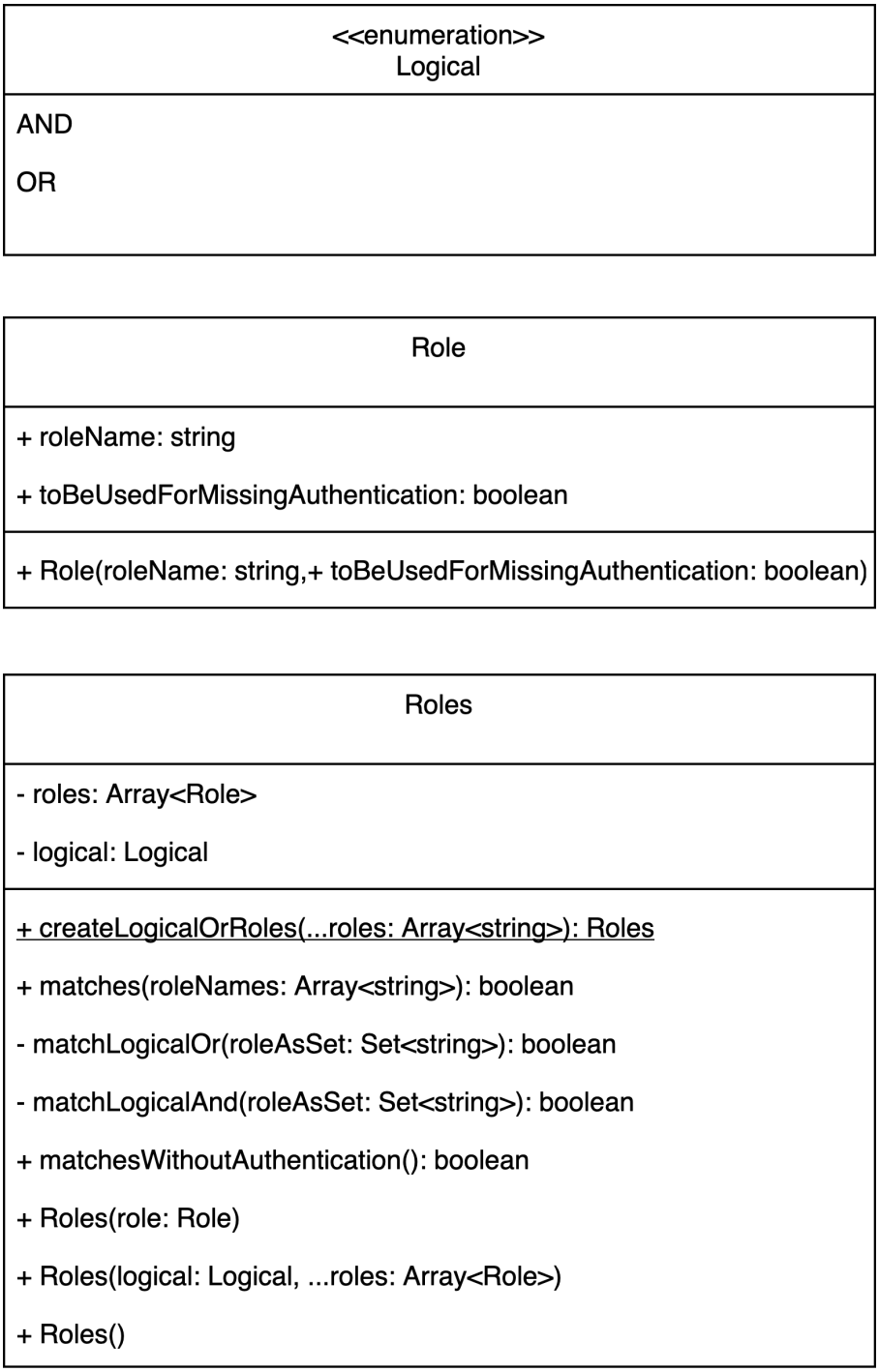


Figure C.12.: Class diagrams Role, Roles and Logical.

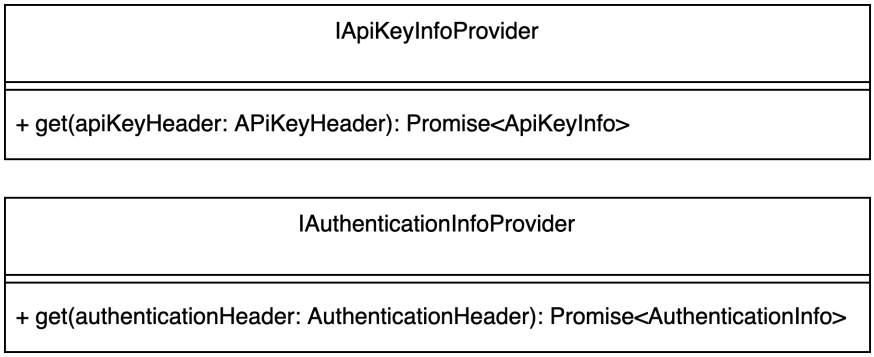


Figure C.13.: Class diagrams IApiKeyInfoProvider and IAuthenticationInfoProvider.

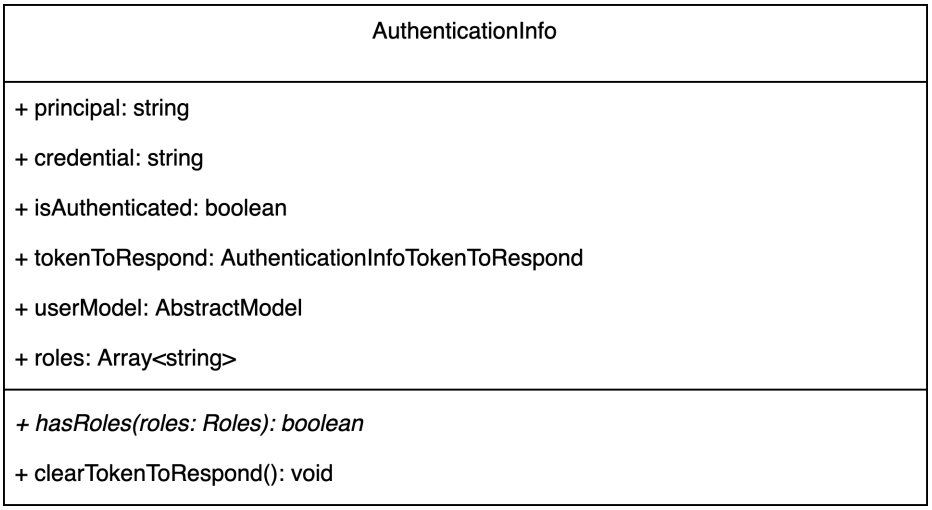


Figure C.14.: Class diagram AuthenticationInfo.

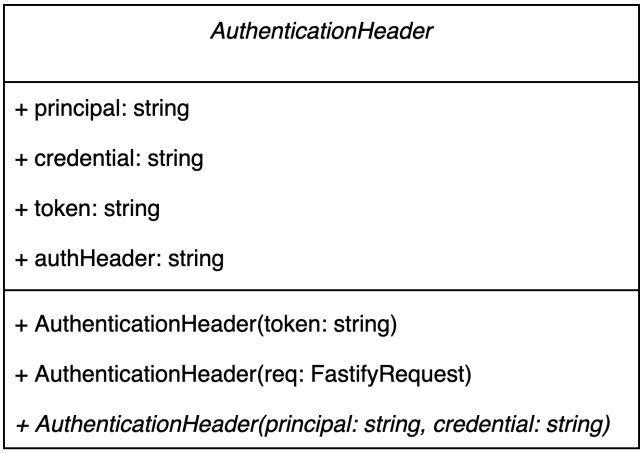


Figure C.15.: Class diagram AuthenticationHeader.

Affidavit

Hiermit versichere ich, dass ich die vorgelegte Bachelorarbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

N. N., am July 18, 2021

Approval for Plagiarism Check

Hiermit willige ich ein, dass zum Zwecke der Überprüfung auf Plagiate meine vorgelegte Arbeit in digitaler Form an PlagScan (www.plagscan.com) übermittelt und diese vorübergehend (max. 5 Jahre) in der von PlagScan geführten Datenbank gespeichert wird sowie persönliche Daten, die Teil dieser Arbeit sind, dort hinterlegt werden.

Die Einwilligung ist freiwillig. Ohne diese Einwilligung kann unter Entfernung aller persönlichen Angaben und Wahrung der urheberrechtlichen Vorgaben die Plagiatsüberprüfung nicht verhindert werden. Die Einwilligung zur Speicherung und Verwendung der persönlichen Daten kann jederzeit durch Erklärung gegenüber der Fakultät widerrufen werden.

N. N., am July 18, 2021