

**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

MASTER THESIS

Anastasia Akhvlediani

Expanding the Dataspecer Tool for Streamlined API Creation and Management

Department of Software Engineering

Supervisor of the master thesis: doc. Mgr. Martin Nečaský, Ph.D.

Study programme: Computer Science - Software and Data Engineering

Prague 2024

Acknowledgement

I would like to thank my supervisor doc. Mgr. Martin Nečaský, Ph.D. and the project consultant Mgr. Štěpán Stenclák for their help and supervision of my thesis.

Separately, I would also like to thank my family for their support and encouragement.

Title: Expanding the Dataspecer Tool for Streamlined API Creation and Management

Author: Anastasia Akhvlediani

Department / Institute: Department of Software Engineering, Charles University

Supervisor of the master thesis: doc. Mgr. Martin Nečaský, Ph.D, Charles University

Abstract: The aim of this thesis is to enhance the functionality of the Dataspecer tool which primarily focuses on modeling and maintaining schemas of data structures which are based on various conceptual models. The extension allows the users of the tool to generate OpenAPI specifications for the aforementioned data structures. This thesis provides state of the art regarding the following concepts: API, REST API, API specification and OpenAPI specification. What's more it analyzes various sources and defines characteristics of a good REST API (with respect to API specifications). The developed extension facilitates generation of OpenAPI specifications that adhere to the characteristics of a good REST API.

Keywords: Dataspecer, OpenAPI, REST, API Specification, OperationCard

Table of Contents

Introduction	1
Purpose of Thesis	1
Relevance of the Topic	2
Work Structure and Scope.....	3
1. Research Approach	4
2. APIs and their Specifications – State of the Art	5
2.1 API Overview	5
2.2 REST API Overview	6
2.3 API Specifications and their Benefits	9
2.4 OpenAPI Specifications	10
2.4.1 Basic Structure of OAS.....	10
2.4.2 Benefits of OAS Utilization.....	12
3. Criteria for Good APIs	13
3.1. Robust API Knowledge.....	13
3.2 Overview of Good API Criteria	15
3.3 Defining Good API	17
3.4 Defining Good REST API.....	17
4. Dataspecer Tool	21
4.1 Dataspecer Principles	21
4.2 Creation of Data Structures	24
4.3 Exploring Data Structures with Dataspecer	27
5. Expanding the Dataspecer Tool with Streamlined API Creation and Management	30
5.1 Requirements Analysis.....	30
5.2 Solution Design	36
5.3 Extension Demonstration	44

5.3.1 Capturing User Input and its Conceptual Alignment.....	46
5.4 Output OAS.....	52
5.4.1 Paths and Operations	53
5.4.2 Components	59
5.4.3 Supported Constructs in the Output OAS.....	61
5.5 Architecture	63
5.6 Implementation.....	66
6. Evaluation	75
Conclusion	85
Bibliography.....	86
List of Figures	89
List of Tables.....	92
A Attachments	93
A.1 Accessing the Extension Application.....	93
A.2 Build Instructions	94
A.3 Electronic Attachments	95

Introduction

API (Application Programming Interface) represents a set of rules that dictates how software programs communicate with each other. In particular, API specifies how one program can access data or functionality of another program [1]. In this day and age APIs as well as API specifications are of vital importance. What's more, API specification represents the backbone of successful development of an API [2]. **Dataspecer** [3] represents a tool that has been in development for several years at the faculty of Mathematics and Physics at Charles University in Prague. To be more precise, it is a tool that aims to manage as well as model schemas of data structures. These schemas are based on conceptual models sourced from the Internet. Furthermore, Dataspecer (semi)automates various tasks which are related to data structure schemas. For example, the tool supports schema creation in various formats such as XML, JSON and CSV. Despite its advanced capabilities, **currently, Dataspecer does not support creation of API specifications.**

Purpose of Thesis

This thesis serves the purpose of expanding the Dataspecer tool with the features necessary for API design. The main mission of this thesis is to create an extension which will provide the functionality to design API specifications according to the OpenAPI standard based on data structure schemas designed in Dataspecer. OpenAPI specification represents a formal document holding the information about the elements that the API contains [4], [5]. It follows the OpenAPI standard, which in turn represents language-agnostic interface for RESTful APIs [5]. This extension will allow the users to create OpenAPI specifications that are tailored to their specific needs. The process of achieving the aforementioned mission is divided into following goals:

- Determination of what constitutes as a good, high-quality API: The thesis will analyze the ways of designing good, quality APIs (with respect to API specifications).
- Assessing the involvement of the user in the process of creating API specification: Providing input for API specifications represents the most critical responsibility of the user, due to the main goal of generating API specifications that are tailored to user's particular needs. The thesis will specifically focus on analyzing how and where the user should be involved in

designing API specifications and what kinds of inputs are required to design high-quality API specifications.

Relevance of the Topic

According to CISCO, the concept of Internet of Everything (IoE) is a networked connection of four entities: people, process, data and things [6]. Moreover, these entities interact as well as exchange real-time data with each-other. APIs are the point of connection between products or services which means that they provide the opportunity for these entities to communicate with one another. Utilization of APIs has many advantages. Developers are able to implement new services as well as gain an essential insight when it comes to interacting with existing functionalities [7]. Based on this information the APIs represent a crucial part of the tech industry. More precisely, they as the points of connection are of vital importance especially in the context of IoE, since in this scope the connection between aforementioned four entities (people, process, data and things) represents the core element [7].

Now that the relevance of APIs in general is already considered, the relevance of API specifications may be discussed. API sprawl is a term which is often used nowadays when it comes to APIs. This term is used to represent the growth of APIs inside and outside of the companies [8]. Moreover, the term API sprawl carries some negative context due to its usual association with inadequate planning [8]. When it comes to software development, API specifications are of vital importance since they represent the backbone of successful development process [2]. More precisely, API specifications help people in the tech industry navigate possible disorder resulting from API sprawl and offer a lot of benefits to the companies as well as the teams, regardless of their specific demographics [8].

Given the significance of the aforementioned concepts in contemporary discourse, the relevance of the thesis topic becomes all the more apparent. The concept of API as well as API specification will be discussed in more detail in the second chapter – APIs and their Specifications – State of the Art.

Work Structure and Scope

The thesis is divided into six chapters each of which serve its dedicated purpose.

First chapter considers **research approach** of the thesis.

Second chapter aims to review the **state of the art** by providing a comprehensive overview of APIs including an in-depth examination of REST APIs in particular. What's more, this chapter considers API specifications in general and specifically OpenAPI specifications as well.

Third chapter of this thesis is concerned with understanding what is meant by a good API. The goal of this chapter is to provide a set of characteristics that **define a good, high-quality API** in the scope of **REST APIs** with respect to API specifications.

Fourth chapter is dedicated to **Dataspecer tool**. More precisely, it gives the reader understanding of basic fundamental principles of Dataspecer. Next it highlights the most relevant feature in the context of this thesis – creation of data structure schema. Lastly it provides a comprehensive overview of the created data structure schema – *Tourist destination* which is then utilized as a **running example** across this thesis.

Fifth chapter is solely concerned with “**Expanding the Dataspecer tool with API creation and Management**”. This is the developed extension. This chapter considers all relevant aspects of the project. First **requirement analysis** is provided. Next, **solution design** is considered. What's more, this chapter provides extension demonstration along with **the mechanism of capturing necessary input from the user**. This chapter also considers **output** OpenAPI specification in detail. **Architecture and Implementation** [9] of the project are discussed in this chapter as well.

Last but not least, **sixth** chapter considers **how does the extension facilitate generation of specifications for good REST APIs**.

1. Research Approach

Now that the general idea of this thesis is already considered, the flow of achieving the aforementioned goals and objectives may be discussed. This paper is based on research project “Expanding the Dataspecer Tool for API Creation and Management” conducted by myself in the scope of class Research Project (NPRG070) at Charles University [9]. The research project mentioned prior, encompasses developed software. What’s more, the project of writing software was being conducted in parallel with the writing of this thesis. As for the workflow, firstly the Dataspecer tool was explored. Next the schemas of the data structures created using Dataspecer were analyzed. An important step following this process was determination of state of the art. Based on this information as well as insights of various sources the characteristics of a good REST API (with respect to API specifications) were defined. As mentioned, these steps were taken in parallel to developing corresponding software. Lastly, the results were analyzed and the project was evaluated in terms of how it facilitates generation of API specifications for good REST APIs. The workflow is illustrated by Figure 1.

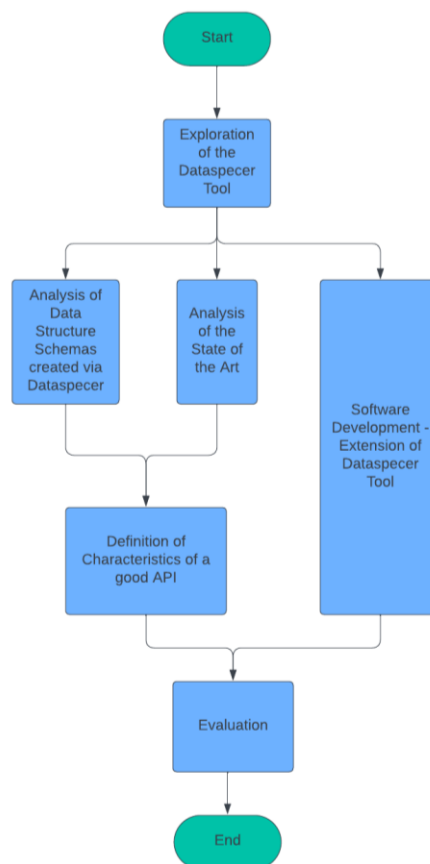


Figure 1 – Research Workflow (Source: Author)

2. APIs and their Specifications – State of the Art

The aim of this chapter is to establish general understanding of following concepts: API, REST API, API specifications and OpenAPI specifications based on relevant academic and professional sources. Because of the fact that the extension of the Dataspecer tool generates API specifications in accordance to REST, it is essential to cover these particular topics. Subchapter 2.1 will focus on APIs in general whereas 2.2 will consider REST APIs. Finally, 2.3 will cover the topic of API specifications and 2.4 will consider OpenAPI Specifications in particular.

2.1 API Overview

API is an acronym for Application Programming Interface and represents protocols (a set of rules) which allow software programs to communicate with each other and determine the format of this communication [1]. More precisely, these rules determine how a software program can access data and/or functionality provided by another software program [1].

In this day and age APIs represent an essential part when it comes to software development. As said, they aim and provide an efficient way of allowing two software programs to communicate with one another and share functionality [1]. Due to APIs being very flexible companies are able to connect with new partners and offer more services to their current customers [10]. This helps them reach new markets and make big profits while upgrading digitally [10]. What's more, APIs make it easier to design and develop new applications and services, as well as integrate and manage existing ones [10]. More particularly, APIs help companies connect their (many) applications, which often operate separately [10]. This kind of integration allows apps to work together smoothly, automating tasks and enhancing collaboration among employees [10]. Without APIs, companies could face information gaps, that slow down work and reduce efficiency [10].

The foundation of the API lies in the client-server model [1]. When it comes to this type of communication, one program – the client requests a service (or resource) from another program – the server [1] [11]. More precisely, when the client provides a request to the server, the server acknowledges the request, processes it, and subsequently provides the response back to the client [12]. This process is illustrated by Figure 2 below.

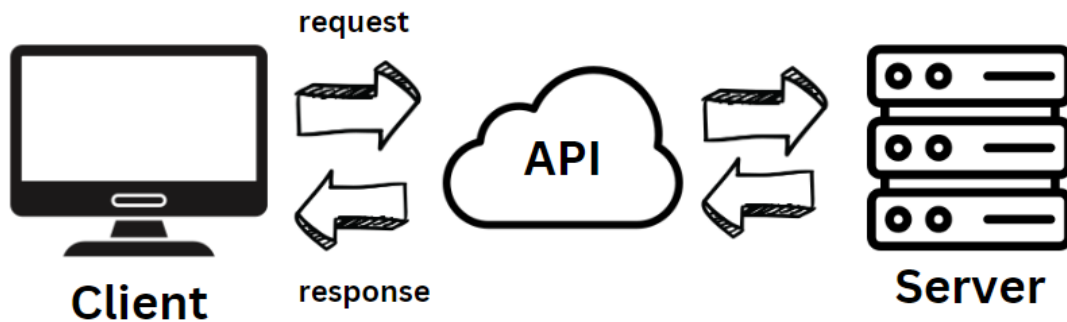


Figure 2 – Client-Server Communication (*Source: Author*)

This mechanism is also utilized in APIs – Initially a request is sent by the client to the server [1]. The request is made using a specific protocol (commonly HTTP) and contains information such as the operation the client needs to perform accompanied by the necessary parameters if they exist [1]. As usual the request is sent via the Internet or local network [1]. Once the request is received by the API server, processing can start [1]. For instance, the API server may validate and/or authorize the request, authenticate the client or perform other necessary operations [1]. The API server sends a response to the client, which can comprise data, an error message, or a status code indicating the operation's outcome [1]. Once the client receives the response it processes it and behaves accordingly [1].

2.2 REST API Overview

Now that the general API overview is already provided, the concept of REST API may be discussed. REST APIs conform to Representational State Transfer architectural style [13]. It is important to note that REST functions independently of any underlying protocol and is not inherently tied to HTTP [14]. However, HTTP is commonly utilized as the application protocol in most REST API implementations [14].

The concept of **resource** represents the foundation of RESTful APIs. It encompasses an entity which possesses a type, associated data, relationships to other resources as well as collection of operations performing various operations on the resource itself [15]. The **type** of a resource **specifies** its **class**. For example, a resource of type *chair* implies that the class of this resource is *chair*. A REST resource has to be identifiable via unique identifier (usually URI) [15]. For example, if the resource represents an instance of *chair* class with ID 1, it has to be identifiable via a URI which resembles the following: “/chairs/1” [16]. It is also important to note that it is possible for

resources to be grouped into collections [15]. When it comes to REST, each collection of resources is homogeneous, which means that, a collection of *chairs* would only contain resources of type *chair*. It is also possible for a resource to exist outside of a collection as a singleton instance [15]. What's more, sub-collections inside a resource could also exist [15]. For example, in a chair management service each *chair* is a resource. All *chairs* can be grouped into a *chairs* collection which in turn also represents a resource. Within each *chair*, a sub-collection of *colors* could exist. It would specify all the colors present in one *chair*. Furthermore, Parallels can be drawn between resources in REST and the objects in OOP (Object-oriented Programming), since both of these entities encapsulate state as well as behavior. However, the behavior of RESTful resources is defined by fewer standard methods, such as GET, POST, PUT, PATCH, DELETE [15]. Possible sample operations for *chair* resource are:

- Retrieve collection of *chairs* (GET)
- Create a new *chair* (POST)
- Delete particular *chair* (DELETE)

It is important to consider that, like any other architectural style there are several principles/constraints that need to be satisfied for a service interface to be considered as RESTful. According to [13], these principles are:

- Uniform Interface
- Client-Server
- Stateless
- Cacheable
- Layered System

In general REST API has aforementioned principles, whereas in the scope of the extension of the Dataspecer tool the aspects from only first three principles – uniform interface, client-server and stateless are relevant. These individual concepts are discussed below.

Uniform Interface

According to the principle of generality, when it comes to designing software, it is crucial that it is free from unnatural restrictions and limitations [17]. A classic example is Y2K problem also known as the “Millenium Bug” [17]. In this case, only two digits were utilized in order to represent years which is an unnatural restriction [17]. At the time when this limitation was implemented, it might have seemed reasonable [17]. However, as time went by and the year 2000 approached, the limitation led to unforeseen consequences [17]. In particular, the systems that relied on two-digit year numbers encountered the Y2K problem, where they couldn't distinguish between the years 1900 and 2000, potentially causing errors or system failures [17]. As said, uniform interface is one of the key principles when it comes to software being considered as restful [13]. It is important to note that by applying aforementioned principle of generality to components interface, systems architecture becomes simpler, and the visibility of how different parts interact with each other is enhanced [13]. Moreover, there are various architectural restrictions that contribute to creating a uniform interface as well as directing how the components should behave [13]. The restrictions that are able to achieve a uniform REST interface are:

- **identification of resources** – For each resource that is involved in the client-server communication a distinct identification needs to be provided by the interface [13]. This means that each resource has to be identifiable usually via a unique URI – for example: “/chairs/1” [16]. The id (1) inside the URI has to be unique as well in order to access particular instance.
- **manipulation of resources through representation** – The resources have to have a uniform representation in the response sent from the server [13]. The consumers of the API use these representations in order to modify the state of the resource in the server [13]. This means that output models have to be defined. In the context of *chair* resource example, a response model for *chair* has to be established.
- **self-descriptive messages** – Each message has to carry sufficient information to explain how to handle the message [13].
- **hypermedia as the engine of application state** – Client should only possess initial URI of the application [13]. The client application has to dynamically

drive all additional resources and interactions through the utilization of hyperlinks [13].

Client-Server

The client-server model represents an underlying structure and provides separation of concerns [13]. Independent evolving of client as well as server components are possible due to separation of concerns [13]. This concept is advantageous for portability (of UI across multiple platforms) as well as scalability of server components [13]. However, despite the fact that client and server components evolve independently it is vital to maintain the connection between them [13].

Stateless

When it comes to software being restful, it needs to be stateless. This means that each request sent by the client to the server has to include sufficient information so that the server is able to understand and process the request accordingly. It is the client applications responsibility to keep session state since the server is not able to take the advantage of any previously stored context information on the server [13].

2.3 API Specifications and their Benefits

Having discussed APIs and REST APIs in particular, API specifications may be considered. The term API specification describes a formal document holding the information regarding the elements that the API has to contain [4]. What's more, usually the creation of API specification is the process which is done before the development phase, which means that the specification is created before the software engineers build the actual API [4]. In fact, according to a poll, mentioned by [18], when it comes to deciding between spec-first and code-first approaches, the majority of the developers lean towards the first (spec-first) option. Whilst employing a spec-first approach a commitment focusing on intentionality is made to the consumers. According to [18], this approach offers a lot of benefits some of which are:

- **Clarity and Consistency** – Due to the utilization of spec-first approach guesswork on software engineers end is eliminated and a clear understanding of the tasks is established.

- **Early Error Detection** – The process of specifying the behavior as well as the requirements of the software in advance reduces the likelihood of errors by identifying the errors in the early stages.
- **Improved Collaboration** – An API specification represents a shared platform for the team of software engineers as well as different stakeholders and promotes discussions around this topic.
- **Time Efficiency** – As mentioned, by utilizing spec-first approach a lot of work is done before the development phase. This saves time during the subsequent phases such as development, testing as well as debugging.

A related study was conducted, where the approach of API-first design as well as various tools were assessed and examined by implementing an API as well as a corresponding client application [2]. According to [2], the API-specification was a central component, a core foundation for this process.

2.4 OpenAPI Specifications

OpenAPI specification (OAS) represents a special case of API specification. More precisely, OpenAPI is a standard representing a language-agnostic interface for RESTful APIs [5].

2.4.1 Basic Structure of OAS

OAS contains a lot of different constructs. These sections are: metadata, *servers*, *paths* (which in turn consists of other sub-sections), input and output models as well as authentication [19]. Each of the aforementioned sections serve their own purpose:

- **Metadata** – OAS metadata contains two sub-constructs – *openapi* and *info*.
 - *openapi* – specifies the version of the OpenAPI defining the overall structure of the OAS [19].
 - *info* – specifies non-functional information [20]. Title, description as well as the version of the actual API are specified in this section [19].
- **Servers** – specifies the base URL of API requests which means that all API paths are relative to the base path specified in the servers [19], [20].
- **Paths** – specifies individual paths of the API. As mentioned, it consists of sub-constructs – operations representing HTTP operations for particular paths. The *operations* construct in turn consists of other sub-constructs. Request body

(*requestBody*) as well as responses (*responses*) represent two of the most important subsections of the operations construct [19], [20].

- Request Body – If request body is sent by an operation, it is specified via this subsection.
- Responses – Specifies the status codes as well as schema of the response object.
- **Input and Output Models** – specified via *components/schemas* construct. These input and output models represent the data structures which are utilized across the API. More precisely, it is possible to reference these schemas in different subsections of the OAS [19].
- **Authentication** – specified via *securitySchemes* and *security* constructs [19].

Figure 3 exemplifies the structure of an OAS for sample tourist destinations management API specification. *Openapi* and *info* constructs represent the metadata, whereas the base URL is specified in the *servers* construct. *Paths* construct contains URLs exposed by the API and their respective operations while the *components* construct is populated with models utilized for input and output. As for the *security*, it specifies the authentication mechanism for the current API.

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "TouristDestinationsAPI",
    "description": "test",
    "version": "1.0"
  },
  "servers": [ ...
  ],
  "paths": {
    "/Touristdestinations/{id}/contacts": { ...
    },
    "/Touristdestinations": { ...
    },
    "/Touristdestinations/{id}/contacts/{id}": { ...
    },
    "/Touristdestinations/{id}": { ...
    }
  },
  "components": { ...
  },
  "security": [ ...
  ]
}
```

Figure 3 – Sample OpenAPI Specification for Tourist Destinations Management API
(Source: Author)

2.4.2 Benefits of OAS Utilization

OAS and in particular OpenAPI-driven API development offers a lot of benefits to its users. [21] points out following benefits:

- **Improved Developer Experience** – Positive developer experience of the API consumption is of vital importance in the context of API ecosystems. Taking the needs of the consumers into consideration in advance allows adopting developer-centric approach as well as catering to the end-users needs before actually tackling the challenges of the development phase.
- **Fostering Independence** – The problem of different teams depending on each other is solved by adopting the OpenAPI-driven API development approach. More precisely, having OpenAPI specification ensures that different stakeholders are on the same page when it comes to understanding what are the goals of the API and what it is supposed to do.
- **Accelerated Market Introduction** – As mentioned in the previous point, OpenAPI-driven development approach fosters independence of various teams. Because of this, these teams are able to do their job in a more efficient manner which results in a faster release of the product.

Furthermore, the utilization of OAS increases the likelihood of stable API implementation [22]. Due to a strong open-source community and positive performance history, OAS can seamlessly be integrated into API design process. What's more OAS can also represent a foundation for API documentation [22]. More precisely, it is possible to auto-generate API documentation based on an existing OpenAPI specification. This is beneficial not only for the developers but also for the clients, since API documentation is frequently utilized by both sides [22].

3. Criteria for Good APIs

This chapter serves the purpose of defining what is a good API with respect to API specifications. In order to do so, firstly the process of having a robust knowledge of API will be defined. Next, the concept of having a good API will be considered. This section will consolidate this information and will provide criteria for good, high-quality REST APIs (with respect to API specifications).

3.1. Robust API Knowledge

According to [23], the notion of Robust API knowledge was defined in the context of a developer in isolation that needs to comprehend the code using API as well as consider options for its utilization and implement or modify code that uses API in order to achieve specific behavior. Robust API knowledge is structured based on three main concepts. These concepts are: Domain Concepts, Execution Facts and API Usage Patterns [23]. According to [23], the knowledge consolidated by these three components give developers opportunity to have a good understanding of the API they are trying to work with and therefore be able to successfully utilize this API. Now that the overview of the Robust API Knowledge notion is already provided, the definition of individual components may be considered.

Domain Concepts

According to [23], the umbrella of domain concepts consolidates two aspects. Abstract ideas existing outside of an API that the API tries to model, represents the first aspect. The second aspect is terminology used by the API and by the documentation in order to refer to the concept [23]. For instance, if the API designer aims to model a chair, they have to take essence of chair into account and refer to it as a chair. The simplest way of explaining essence of chair would be following: it is a piece of furniture on which people sit. What's more it is crucial that this concept is called chair. Otherwise, semantic errors could occur. For example, someone referring to a piece of furniture on which people sit as a piano would result in confusion. Even though it is technically possible to sit on a piano, it does not reflect the essence of it. On the other hand, the definition of a keyboard musical instrument does capture the essence of piano.

Execution Facts

Execution facts represent declarative knowledge which is structured in the form of simplified rules about the execution behavior of an API [23]. These execution facts have to be sufficient for making predictions about the execution of the API as well as understanding this process and being able to explain it [23]. More precisely, these facts include the information about various programming concepts such as types, inputs, outputs as well as the effects of executing different parts of an API [23]. However, these details may vary in terms of abstraction [23]. In this case, if the API designer would want to define execution facts for chair management API, they would specify facts related to this concept. A lot of these facts would relate to types (classes) as well as inputs and outputs of relevant requests/operations. For example, in order to create a *chair* a developer needs to know which parameters and what kind are required for *chair* creation. Understanding that in order to create a new instance of *chair* the *material* as well as *leg count* need to be utilized as parameters represents an example of execution facts.

API Usage Patterns

When it comes to API Usage Patterns, this concept mainly considers the examples that showcase how this API can be used. To be more precise, some form of a code snippet may be included in this section of the Robust API Knowledge [23]. According to [23], API usage patterns should also include the rationale utilized for the construction of the (code) pattern mentioned above. This is of vital importance because a code pattern accompanied by the rationale carries information which is based on the other two concepts – execution facts and domain concepts which allows developers to understand the possibilities of changing the code in order to achieve desired behavior [23]. Figure 4 represents an example of API usage patterns. This example illustrates the code syntax of API utilization and corresponding comments. For instance, the example shows the exact code snippet of *camera* instantiation and what's more, this code is provided with a corresponding comment explaining what this particular code snippet does. In simple terms, API usage patterns represent sample code snippets for API utilization with comments. This aspect of robust API knowledge does not focus on API specification and is it out of scope of this project.

```

Create a reflective material.
JS:
// Create a camera to capture reflection view.
reflectCubeCamera = new THREE.CubeCamera(...);

// add camera to scene and update the camera
scene.add( reflectCubeCamera );
reflectCubeCamera.update( renderer, scene );

// Create material with reflection view as the environment Map
var material3 = new THREE.MeshBasicMaterial( {
    envMap: reflectCubeCamera.renderTarget.texture
    ...
} );
material3.envMap.mapping = THREE.CubeReflectionMapping;

```

Figure 4 – Example of API Usage Patterns (*Adapted from [23]*)

3.2 Overview of Good API Criteria

Having considered the notion of Robust API Knowledge the criteria for a good, high-quality API may be discussed. According to [7], despite the fact that APIs are a big part of the tech industry and in particular with respect to IoE, surprisingly there has been done only little work in terms of researching what it means to design a high-quality API. The research conducted by Kiesler and Schiffner [7] aims to contribute to this matter by answering following research question “What are important factors for developers when it comes to an API’s quality?”. The authors of the research have conducted an online survey in order to understand the perspective of the developers with respect to important factors for API development [7]. As for the respondents, most of them were experienced professionals/developers [7].

According to the respondents, **understanding (required effort for familiarizing) how the API functions** is a significant factor [7]. The respondents had to also answer an open question and provide their own criteria for a good API [7]. The answers include: “**ease of use**”, documentation, dependencies as well as time needed for integration [7]. The concept “ease of use” is mentioned in other sources as a characteristic for good APIs as well. According to [24], a well-designed API has to be easy to read as well as easy to work with. This goes hand-in-hand with another characteristic – “hard to misuse”. This means that implementation and integration of a good, well-designed API have to be a straightforward whereas the likelihood of writing incorrect code has to be low [24].

The developers were also asked if they adhere to their own rules the contrasting answers were distributed almost evenly [7]. Furthermore, the respondents were asked

about important factors regarding the integration of APIs and the answers were rather conservative [7]. To be more precise, even though no clear trend was identified, **clear structure** and **modularization** was favored in the responses. Lastly, according to the respondents, identified challenges related to API implementation include following: lack of availability (if provided by third party), long-term maintenance and compatibility. It is important to consider, that a lot of developers overcome these issues via **Developer User Experience** [7]. According to [7], even though it is not stated explicitly a lot of the answers are aimed at better Developer User Experience and the concepts “ease of use”, “easy to learn” and community contribute to a better DevUX [7]. This means that nowadays due to the high number of existing APIs sole functionality is not a deciding factor anymore, the developers take other aspects such as DevUX into consideration as well [7].

In addition to a better DevUX, **completeness** and **conciseness** of an API represents contribute to the quality of the API [24]. A complete and concise API is a way of creating fully fledged applications [24]. However, completing an API represents an iterative process, which means that the people working on the API build on their past progress (on the existing API) [24].

3.3 Defining Good API

Having considered not only the notion of robust API knowledge as well as other characteristics of good, well-designed, APIs, a good API can be defined as one exhibiting characteristics listed in Table 1.

Characteristic	Definition
Robust API Knowledge	<ul style="list-style-type: none">• The developers are able to have a comprehensive understanding of the API – its domain concepts as well as execution facts.
Usability (Usable)	<ul style="list-style-type: none">• The API is easy to use, intuitive.• It hard to misuse API – the likelihood of errors is minimized.
Structured	<ul style="list-style-type: none">• The API components are defined in a well-structured manner.• The API has modular design promoting reusability.
Documented	<ul style="list-style-type: none">• The API documentation supports the team working on this API to understand and work with the API in an effective manner.• The time required by familiarizing with the API is minimized.
Enhanced Developer User Experience	<ul style="list-style-type: none">• The API is easy to use and learn.• The developers have positive experience when it comes to interacting with the API.

Table 1 – Characteristics of Good API (Source: [7])

3.4 Defining Good REST API

Now that the characteristics of a good API are already defined, the notion of a good REST API may be considered. As mentioned in subchapter 2.2 a RESTful API has to provide uniform interface and has to be stateless [13]. As noted, uniform interface encompasses: identification of resources, manipulation of resources through

representation, self-descriptive messages and hypermedia as the engine of application state [13]. However, for this research hypermedia as the engine of application state was out of scope. It is more of an implementation choice and is not commonly utilized in OpenAPI specifications [25], [26]. What’s more, there are notable best practices that need to be followed when dealing with REST APIs. According to [27], some of the notable best practices are:

Practice	Definition
Utilization of recommended naming conventions	<ul style="list-style-type: none"> • naming conventions have to be not only clear and precise but also aligned with their corresponding functionality.
Usage of appropriate HTTP method	<ul style="list-style-type: none"> • when it comes to selecting HTTP method for a particular endpoint, the choice has to be based on the essence of the operation which is being performed.
Management of REST API requests and resources	<ul style="list-style-type: none"> • Effective management of requests and resources of an API establish a positive user experience as well as effective client-server communication
Distinction between path and query parameters	<ul style="list-style-type: none"> • Path parameters aim to identify a particular resource. • query parameters are utilized for different purposes – for example: filtering

Table 2 – REST API: notable Best Practices (*Source: [27]*)

Having considered good API characteristics as well as best practices of dealing with REST APIs, good REST APIs may be defined. Table 3 illustrates the characteristics of a good RESTful API (with respect to API specifications). It is important to note that, **the characteristics of a good REST API have to be reflected in its API specification.**

Characteristic	Definition
Robust API Knowledge	<ul style="list-style-type: none"> • The developers are able to have a comprehensive understanding of the API (Robust API Knowledge) – its domain concepts as well as execution facts. • For each resource that is involved in the client-server communication a distinct identification is provided
Usability (Usable)	<ul style="list-style-type: none"> • The API is easy to use, intuitive. • It hard to misuse API – the likelihood of errors is minimized. • Recommended naming conventions are utilized – they are clear and aligned with their corresponding functionality. • Appropriate HTTP methods are chosen based on the essence of performed operation. • Requests and responses are managed effectively.

Structured	<ul style="list-style-type: none"> • The API components are defined in a well-structured manner. • The API has modular design promoting reusability. • Includes distinction between path and query parameters • Manipulation of resources through representation – server sends the response in its consistent format so that clients can exploit it.
Documented	<ul style="list-style-type: none"> • The API documentation supports the team working on this API to understand and work with the API in an effective manner. • The time required by familiarizing with the API is minimized.
Enhanced Developer User Experience	<ul style="list-style-type: none"> • The API is easy to use and learn. • The developers have positive experience when it comes to interacting with the API.
Stateless	<ul style="list-style-type: none"> • The client request contains sufficient information, so that the server can process it without relying on stored context.

Table 3 – Characteristics of a good REST API (with respect to API specification)
(Source: [13], [27])

4. Dataspecer Tool

Now that the state of the art is already considered and criteria for good, high-quality APIs are already defined, Dataspecer tool may be introduced. The Dataspecer tool aims to model as well as manage schemas of data structures based on various conceptual models from the internet [3]. What's more, Dataspecer enables semi-automation of various tasks related to data schemas including schema creation in JSON and XML formats [3].

4.1 Dataspecer Principles

As mentioned, Dataspecer tool serves the purpose of creating as well as managing schemas of data structures. However, it is important to note that conceptual models available on different sources from the internet represent the foundation of these data structures. High-Level, abstract representation of the data utilized by the companies is portrayed by the conceptual (data) models [28]. Conceptual models aim to focus on the bigger picture and remain technology-neutral. This means that their main goal is to foster a shared/unified understanding of the business by pinpointing the core elements. To be more precise, the expression – shared understanding means that different stakeholders, involved teams are on the same page and have the same understanding/definitions of the essential concepts [28]. What's more, there are a lot of advantages when it comes to conceptual data models. According to [28], these advantages are:

- **Connection between the Data Needs and Business Goals** – conceptual models connect strategic goals as well as business drivers with the facts and business questions.
- **Enhanced Clarity** – conceptual models foster clarity for future models.
- **Clear Scope and Context Definition** – Conceptual models represent the foundation for setting scope boundaries as well as adopting a clear framework.
- **Shared Understanding** – As mentioned, conceptual models provide a shared understanding between different stakeholders, which means that they set a common ground facilitating effective communication.

[28] also provides visual representations for conceptual models as illustrated by Figure 5.

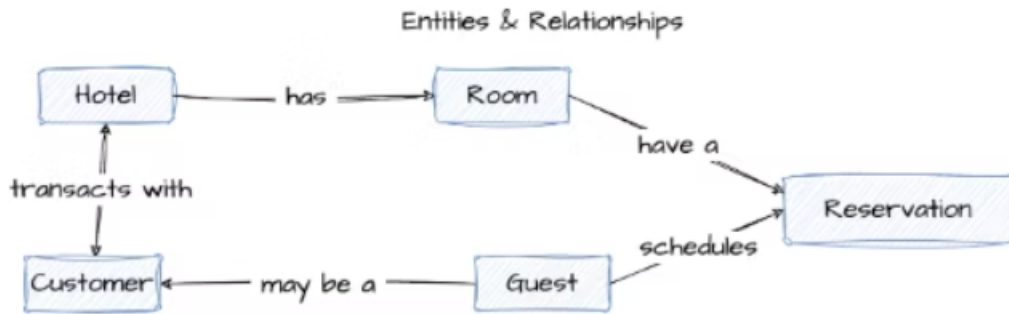


Figure 5 – Transactional Conceptual Data Model (Entities and Relationships only)
 (Source: Adapted from [28])

As observed, conceptual (data) model illustrated by Figure 5 only depicts essential entities as well as relationships between them. Despite the fact that the main goal of the conceptual models is capturing the essence of the businesses, they may be much more complex than simply identifying entities and relationships. This means that they may also include identifiers, different attributes as well as cardinality information. Based on the conceptual model provided by Figure 5 various data structures could exist. One of the examples is illustrated by Figure 6 below. As evident, *Hotel* data structure has simple properties such as: *hotelName*, *hotelId*, *starNumber*. Room data structure has its own simple properties: *sizeInSquareMeters*, *wallColor* and *seaView*. What's more, *hotel* has one or many *rooms*.

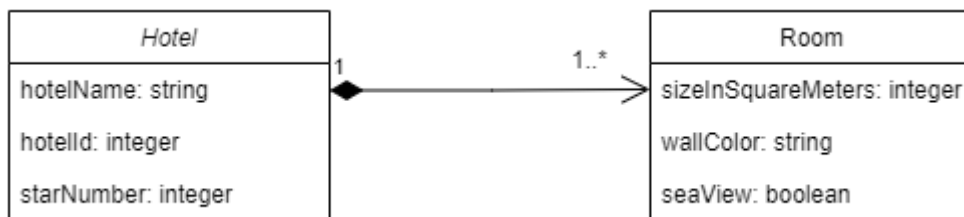


Figure 6 – Sample Data structures based on the Conceptual Model of Figure 5
 (Source: Author)

Dataspecer utilizes complex conceptual models retrieved from various sources as the basis for data structure (schema) creation. In the context of the Dataspecer tool, conceptual model can be defined as a set of classes with attributes and connections named associations. To clarify, the conceptual models represent a foundational source for the data structure creation. Formally, in this context, data structure represents a rooted tree-graph. The root serves as the point of reference for other nodes of the tree.

What's more, the nodes represent classes, which have attributes and are connected via hierarchical associations. Generally, both – attributes and associations can be considered as features in this context, however the difference is that an attribute points to something simple and primitive, whereas an association points to a complex type – a class. Various sources, such as [29] and [30] reference “The Unified Modeling Reference Manual” [31] when describing the concepts of attributes and associations. According to [31], as cited by [29] and [30], attributes are mostly utilized for the purpose of representing data types – values that have no identity. As for the associations, they are a way of representing classes – values which do have an identity [29], [30], [31]. In the context illustrated by Figure 6, *hotelName* represents a string (primitive) attribute of a *Hotel* data structure. The *Hotel* data structure also has an association of type *Room* (not primitive). It is important to note that these classes are derived from the conceptual model representing the base for the current data structure schema. This means that hierarchical organization of some part of the conceptual model defines the schema of the data structure. Because of this, conceptual models allow creation/derivation of multiple different data structures based on the same conceptual model. For instance, there exists a conceptual model – *Tourist destination*. Based on this conceptual model the users are able to create various representations (data structure schemas) of tourist destinations. Some possible sample options are:

- An empty data structure that contains just a title.
- Data structure that contains title as well as attribute(s).
- Data structure that has a title as well as a multi-layered structure consisting of associations and attributes.

Once the data structure (schema) is instantiated, the Dataspecer tool gives the user opportunity to represent them as a JSON or XML schemas [3]. However, the original version of Dataspecer does not have support for OpenAPI specification generation for these data structures. In particular, while Dataspecer encompasses a lot of different features, **the goal of this thesis is to extend the tool with the functionality of creating API specifications (in OpenAPI format) for the data structures designed in Dataspecer.** The primary focus of the next part of this chapter will be the **structure (schema) editor** component, since it is relevant to this project.

4.2 Creation of Data Structures

Structure (schema) editor represents a component which is responsible for creation of data structures (their schemas) in the Dataspecer tool. However, before proceeding to data structure schema creation, data specification needs to be established. This is done by the specification manager which may be accessed via this URL: <https://tool.dataspecer.com/>. Once the user accesses the specification manager, they need to click “Create Specification” button in order to create data specification as illustrated by Figure 7 and 8.

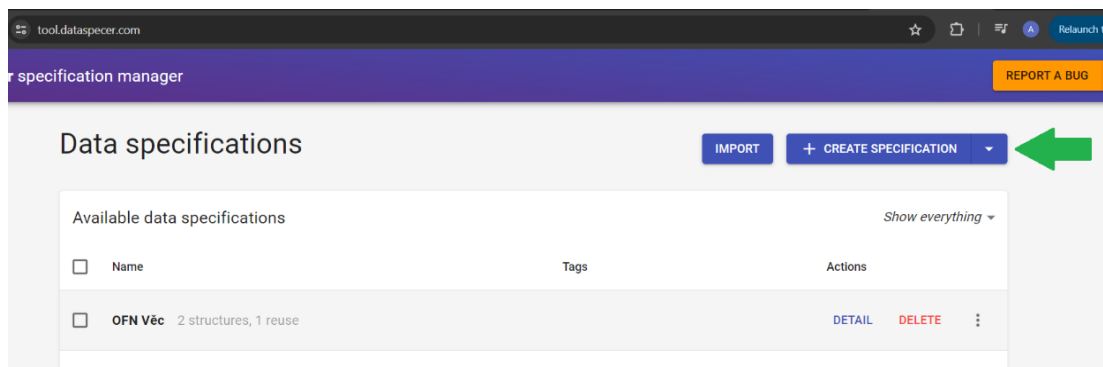


Figure 7 – Dataspecer: Specification Manager (Source: Adapted from [32])

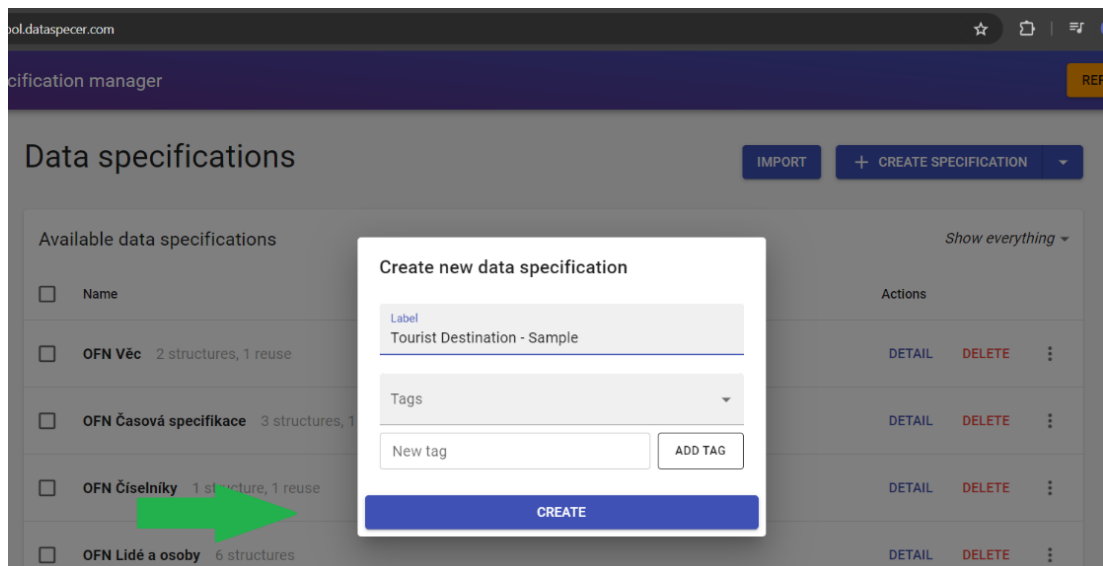


Figure 8 – Dataspecer: Create Data Specification (Source: Adapted from [32])

Having established data specification, data structures may be created within it. This is achieved by clicking “CREATE DATA STRUCTURE” as shown by Figure 9.

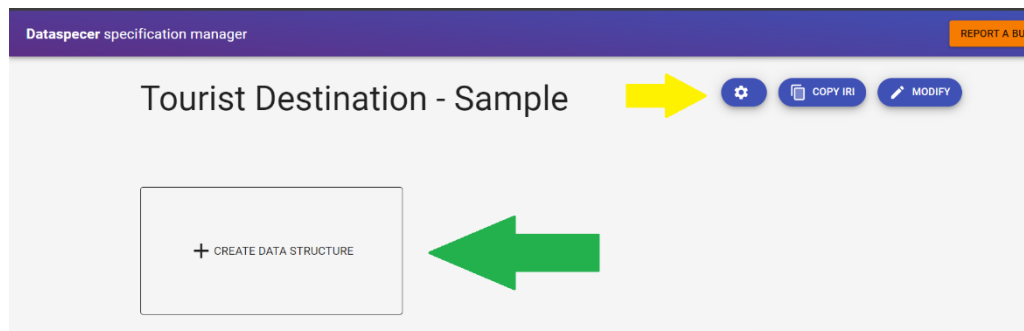


Figure 9 – Dataspecer: Data Structure Creation (Source: Adapted from [32])

Because of the fact that this project is conducted exclusively in English, the English language has to be chosen via utilizing the button, pointed by the yellow arrow on Figure 9. It is important to note that before creating the data structure, source for the vocabulary needs to be selected. The selection of vocabulary sources is possible on the same page by scrolling down as showcased on Figure 10 below.

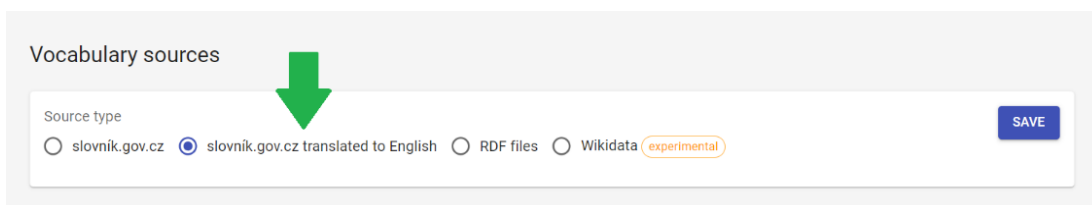


Figure 10 – Dataspecer: Vocabulary Sources Selection (Source: Adapted from [32])

At this point, the users can start designing their data structures. Once “CREATE DATA STRUCTURE” is clicked, the user is redirected to structure editor (Data Structure Editor). Next, the user needs to set the root data structure by clicking “SET ROOT ELEMENT” button. After clicking this button, the user will be prompted to search for a class for the data structure. This is illustrated by Figures 11 and 12.

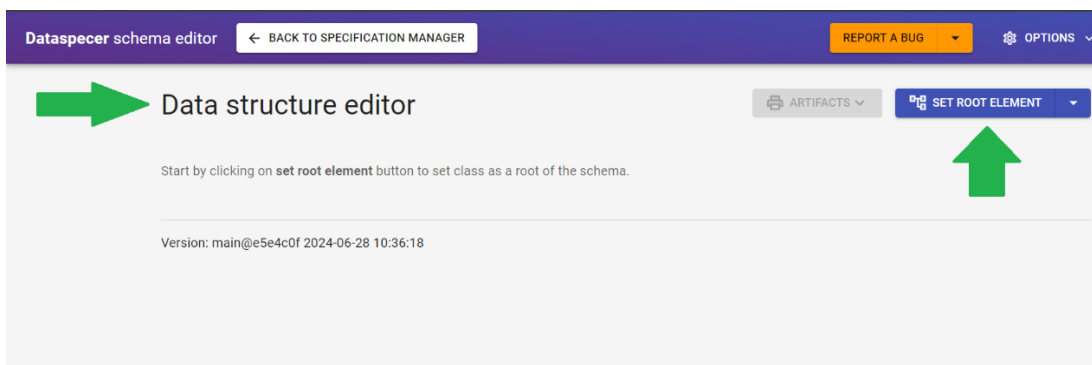


Figure 11 – Dataspecer: Data Structure Editor (Source: Adapted from [32])

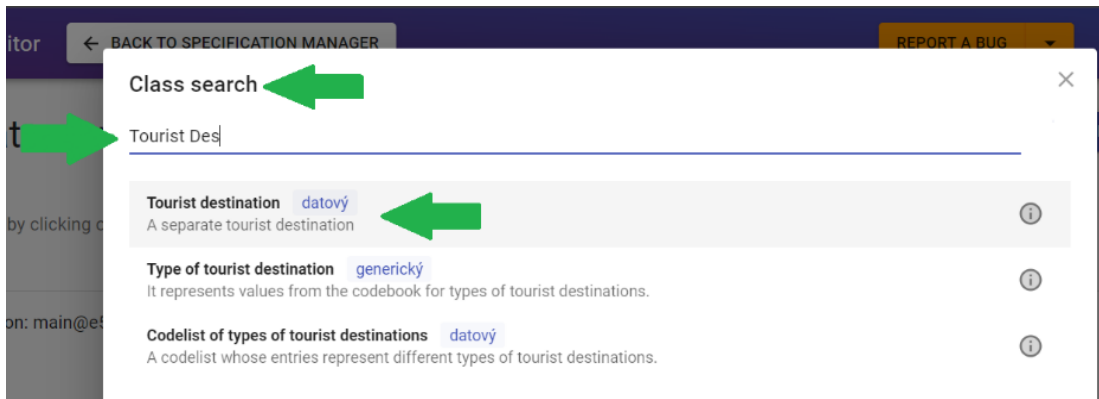


Figure 12 – Dataspecer: Setting Root Element in Structure Editor (Source: Adapted from [32])

Once the root element is set, its fields may be configured. This is done by clicking the plus button next to the root element as shown by Figure 13.

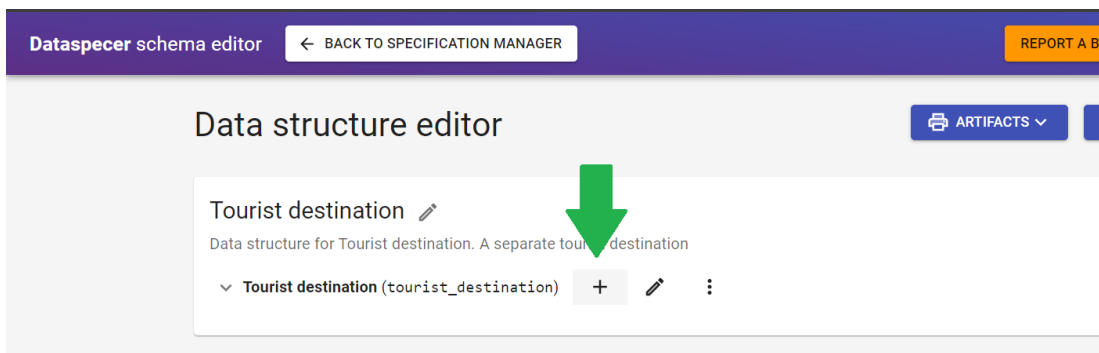


Figure 13 – Dataspecer: Adding fields to the Root Data Structure (Source: Adapted from [32])

After this step, the user is presented with a window listing available fields for the root data structure as shown on Figure 14.

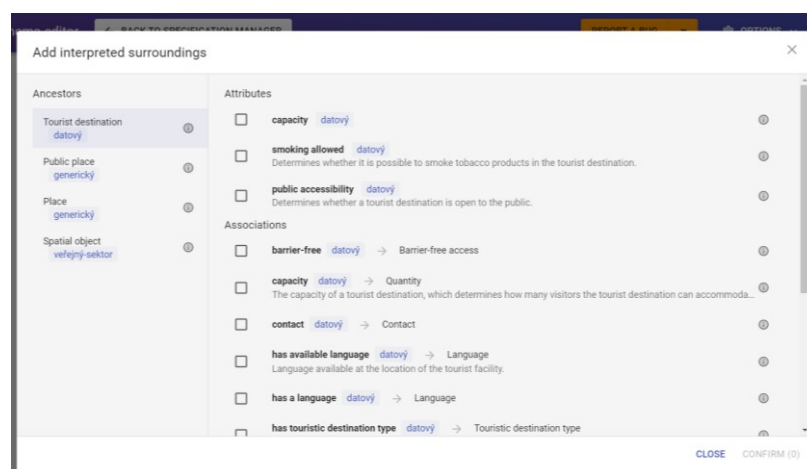


Figure 14 – Dataspecer: Choosing the fields of the Root Data Structure (Source: Adapted from [32])

As one can see, the field options are categorized into attributes and associations. The associations may be elaborated further by following similar process. To be more precise, each association has an adjacent plus button which, when clicked, allows the user to design particular structure of an association. Figure 15 showcases resulting data structure.

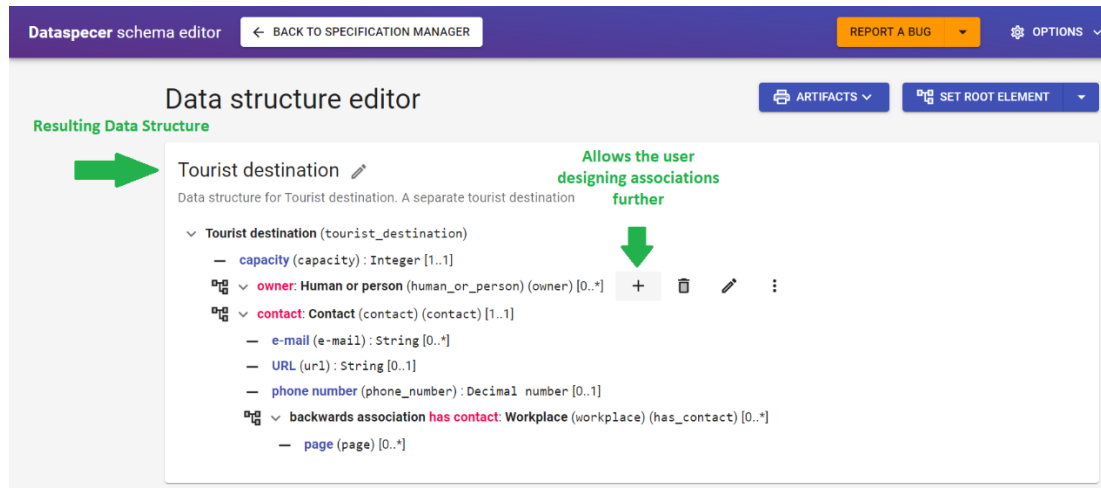


Figure 15 – Dataspecer: Tourist Destination, Resulting Data Structure (Source: Adapted from [32])

4.3 Exploring Data Structures with Dataspecer

Data structure schemas designed via the Dataspecer tool represent multi-level complex structures. *Tourist destination* created in previous subchapter (4.2) will be utilized as a **running example** throughout this thesis. The fields of the root data structure are categorized into attributes and associations. As mentioned prior, **attributes** point to something primitive, while **associations** may have a multi-layered, complex structure themselves [31].

The running example – *Tourist destination*, has three fields: one attribute – *capacity* representing an integer and two associations – *owner* of type *Human or person* and *contact* of type *Contact*. As illustrated in Figure 15, *capacity* (attribute) and *owner* (association) have a relatively simple structure. As mentioned above, attributes point to something primitive, therefore the simplicity in attributes is inherent. By definition they do not represent a nested structure. However, the association – *owner* was intentionally left unconfigured in order to showcase that associations may have a simple structure too. In contrast, another association – *contact* does represent a multi-level, complex structure itself. As observed, it has three attributes: *email*, *URL* and

phone number as well as one association – *has contact*, which in turn has its own attribute – *page*. Figures 16 and 17 below illustrate the attributes and associations of the sample data structure – *Tourist destination*.

The screenshot shows the 'Data structure editor' interface for 'Tourist destination'. The structure is defined as follows:

- Tourist destination (tourist_destination)**
 - capacity (capacity): Integer [1..1]** (Attribute)
 - owner: Human or person (human_or_person) (owner) [0..*]** (Association)
 - contact: Contact (contact) (contact) [1..1]** (Association)
 - e-mail (e-mail): String [0..*]** (Attribute)
 - URL (url): String [0..1]** (Attribute)
 - phone number (phone_number): Decimal number [0..1]** (Attribute)
 - backwards association has contact: Workplace (workplace) (has_contact) [0..*]** (Association)
 - page (page) [0..*]** (Attribute)

Figure 16 – Running Example: Attributes (Source: Adapted from [32])

The screenshot shows the 'Data structure editor' interface for 'Tourist destination', highlighting the associations. The structure is defined as follows:

- Tourist destination (tourist_destination)**
 - capacity (capacity): Integer [1..1]**
 - owner: Human or person (human_or_person) (owner) [0..*]** (Association)
 - contact: Contact (contact) (contact) [1..1]** (Association)
 - e-mail (e-mail): String [0..*]**
 - URL (url): String [0..1]**
 - phone number (phone_number): Decimal number [0..1]**
 - backwards association has contact: Workplace (workplace) (has_contact) [0..*]** (Association)
 - page (page) [0..*]**

Figure 17 – Running Example: Associations (Source: Adapted from [32])

Furthermore, the attributes and associations of the data structure designed via Dataspecer may represent collections as well as singleton objects. Table 4 illustrates field details – mainly which field is an attribute or association, singleton object or a collection as well as their class types.

Field Name	Class Type	Is Attribute	Is Association	Represents Collection	Represents Singleton
capacity	Integer	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>
owner	Human or person		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
contact	Contact		<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>
e-mail	e-mail	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>	
URL	url	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>
phone number	phone_number	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>
has contact	Workplace		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
page	page	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>	

Table 4 – Running Example: Tourist destinations - Field Details (Source: Author)

Having designed a sample data structure schema, it may be noted that **not all data structures** designed via the Dataspecer tool **are meaningful**. As observed, there are a lot of steps that the user needs to take while designing data structures according to their specific needs. In particular, the user needs to set the root data structure and then configure its fields – attributes and associations manually. What’s more, the user is also responsible for setting meaningful types for these fields. In particular, technical labels for associations one level below the root data structure have to resemble nouns not verbs, for example – owner and contact are both nouns. **These kinds of data structures represent meaningful input to this project** – “Expanding the Dataspecer Tool with Streamlined API Creation and Management”. The details of this process are discussed in the next chapters.

5. Expanding the Dataspecer Tool with Streamlined API Creation and Management

Now that the structure as well as the creation process of Dataspecer data structures is already discussed, the extension – “Expanding the Dataspecer Tool with Streamlined API Creation and Management” may be considered. At first, this chapter will provide the analysis of the requirements. Next it will consider the solution design as well as demonstrate the extension. The involvement of the user and importance of their input will be discussed along with the demonstration. Furthermore, the output OpenAPI specification will be considered. More precisely, the chapter will focus on the main constructs of the resulting OpenAPI specification and their alignment with the good REST API criteria. This chapter will also consider technical aspects – architecture and implementation. As mentioned, corresponding software was developed by myself for the class “Research Project (NPRG070)” at Charles University and this chapter provides details of implementation as well as an additional analysis for this paper [9].

The program operates under the assumption that the data structures designed via the Dataspecer tool are meaningful. More precisely the program assumes **unique (class) names** of root data structures and its associations across the target data specification. This assumption ensures that there will be no conflict between the data structures in the resulting OAS. It is important to note that this project aims to appeal to a **broad target audience**, which means that users with different background (relative to tech industry) are taken into account.

5.1 Requirements Analysis

Because of the fact that, the goal of this project is to extend the Dataspecer tool with a feature allowing its users to generate API specifications for Dataspecer data structures in OpenAPI format, one of the key requirements of the project was utilization of the OpenAPI standard. The question might arise why was OpenAPI chosen as the format of the output API specification. Having analyzed Dataspecer tool and data structure schemas designed with it – it was concluded that the REST API resources and data structures designed via Dataspecer are similar and compatible. As mentioned in 2.2 the resource represents the main concept when it comes to REST APIs [15]. They not only have a type (class), but also associated data as well as relationships to other resources [15]. What’s more, they have collection of operations performing various

manipulations on them as well [15]. As discussed in 4.3, the Dataspecer data structure, it also has a type (class) as well as attributes and associations. However, the collection of operations manipulating this data structure are not defined in the Dataspecer tool. Table 5 as well as Figure 18 illustrate the similarities as well as distinctions.

Resource Concept in REST	Data Structure in Dataspecer	Meaning
Type	Type	Specifies the class of the entity in both cases – in the context of REST resources as well as the Dataspecer data structures.
Associated Data	Attributes	Specifies simple, primitive fields of the entity in both cases. In the context of the running example the field <i>capacity</i> of type integer exemplifies this concept.
Relationships to other Resources	Associations	Specifies the connection, dependency to other entities in both cases. In the context of the running example, field <i>contact</i> exemplifies this concept.
Collection of Operations	Not Defined	The concept of resource in REST is accompanied by a collection of CRUD operations. Such collection is not defined in the context of Dataspecer data structures.

Table 5 – Resources in REST and Data Structures in Dataspecer (Source: [15])

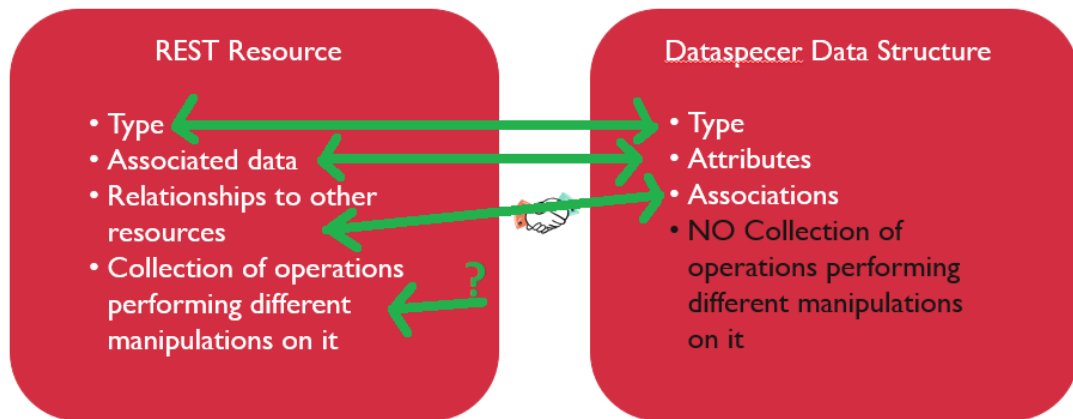


Figure 18 – Resources in REST and Data Structures in Dataspecer (Source: Author)

Based on these similarities and differences, it is evident that there exists a gap between the Dataspecer data structures and the resource concept of REST APIs. In particular, information regarding the operations performing manipulations on the Dataspecer data structures needs to be obtained in order to achieve greater level of compatibility between these two concepts and therefore generate a meaningful OAS. Filling this gap represents one of the most important challenges of this project. **Addressing how the collection of operations for the purpose of manipulating Dataspecer data structures is defined and implemented** represents one of the **key challenges** in this thesis. There are several possible options when it comes to addressing this matter. Before proceeding with these options, it is crucial to define required information and identify which parameters are essential.

As mentioned in 2.2 being stateless represents one of the key characteristics which ensures Restfulness of an API [13]. This implies that each request sent by the client to the server needs to capture essential information so that the server is able to comprehend and process the request adequately [13]. As previously mentioned, when it comes to REST APIs CRUD (Create, Read, Update, Delete) operations are supported. Based on this information, essential parameters for each request/operation with (respect to OpenAPI specification) are:

- Operation Name – specifies the name of the operation.
- Operation Type – specifies operation type, such as: GET, PUT, POST, PATCH, DELETE.
- Endpoint – specifies path exposed by the API.
- Comment – is optional and specifies the summary of the operation.

- Request Body – specifies the parameters passed in the request body. This parameter is not always applicable, for instance in case the operation type is GET.
- Response Body – specifies the expected response.
- Content type – specifies the media type of the response, for example JSON.
- Authentication – specifies authentication method, for example Bearer token.

As mentioned, there are several ways of defining this information for the purpose of API specification generation. In particular, there are two primary alternatives: **enabling user specification of this information** and **automatic generation**. Both of these approaches have their advantages and disadvantages. Enabling the users to specify essential operation details ensures that the resulting OAS is tailored to their specific needs. This means that the users are able to generate a clear and concise OpenAPI specification, which does not include unnecessary request information. By adopting this approach, users only add operations that they think are essential for their API. This would not be possible in case of auto generation of the operations. Without any user input, the developer of this project (myself) would be compelled to consider all possible operations and include each and every one of them in the resulting API specification. In the case of *Tourist destination*, the resulting OpenAPI specification would include all possible CRUD operations for the root data structure and all its associations one level below. This could lead to an excessively large output with a high likelihood of incorporating a significant amount of unnecessary components/information. What's more most of these operations would be unmeaningful, because the user would not be able to specify parameters. Despite the fact that it takes more time for users to provide operation details themselves, the approach of enabling them to provide specifics was adopted. It results in an accurate as well as tailored outcome which **specifically aligns with the user's needs**. What's more, this approach makes it possible to **update the details at a later point** and therefore update the resulting output. These considerations are illustrated by Table 6 and 7 below.

Enabling Specification of Essential Operation Details by User	
Advantages	Disadvantages
<ul style="list-style-type: none"> • Output OpenAPI specification is tailored to users' specific needs. • It is possible to update OAS. • Possibility of a clear and concise output OAS. 	<ul style="list-style-type: none"> • It takes time to design operations (requests). • Users are given more control.

Table 6 – Advantages and Disadvantages of User-Specified Details (*Source: Author*)

Automatic Generation of Essential Operation Details	
Advantages	Disadvantages
<ul style="list-style-type: none"> • Not time-consuming. • Gives more control to the developers, not the user. 	<ul style="list-style-type: none"> • Is general. • Does not reflect the specific needs of the user. • Output OAS may include a lot of unnecessary components.

Table 7 – Advantages and Disadvantages of Automatic Generation (*Source: Author*)

As evident, the key requirements of the project were to extend the Dataspecer tool with a feature that allows the users to generate OpenAPI specifications for the data structures that were designed via the Dataspecer tool. What's more the users have to be able to update their progress. These requirements are satisfied by adopting the approach of allowing the users to specify the details of each operation. It is important to note that the data structure itself represents an input to this extension and it has to be meaningful to ensure that the resulting OAS is also meaningful. As said, the fact that provided data structures are meaningful is the assumption, under which the program operates.

Last but not least, one of the main requirements of the project was to determine for which data structures would it be appropriate to define CRUD operations. More precisely, the discussion centered on whether the operation creation would be limited to the root data structure or extended to its successors – associations as well. In the context of API specification generation, it is logical to provide an option of operation creation for the root data structure and its successor associations one level below.

Going beyond this level of nesting would make the UI form excessively complex for the user and difficult for them to comprehend the key specifics of the operations. What's more, operations below this level may also be considered redundant since they tend to be too granular. Having considered these aspects, it may be concluded that supporting operations beyond the second level would complicate the user interface without providing any significant value. Therefore, the decision was made to support operation creation only for the root data structure (for instance – *Tourist destination*) and its successors one level below (based on the running example – *owner* and *contact*). This means that if *Tourist destination* were a flat data structure which contained only attributes and no associations definition of following operations would be possible:

- Create *Tourist destination*
- Retrieve *Tourist destination(s)*
- Update *Tourist destination*
- Delete *Tourist destination*

However, in the case of a more complex data structure like the running example it is possible to define more operations, in particular – all of the operations mentioned above and additionally:

- Create *owner*
- Retrieve *owner(s)*
- Update *owner*
- Delete *owner*

As well as same set of operations for association named *contact*. As mentioned, operation definition below this level cultivates excessive granularity and is not supported.

In conclusion, whilst analyzing the requirements several key decisions were made:

- The extension allows the user to specify request/operation details.
- The extension allows the user to create operations for main data structures as well as its successors – one level below.
- The output is provided in the form of OpenAPI specification.
- The extension allows the user to update (previously saved) configuration.

5.2 Solution Design

As mentioned, Dataspecer data structures provided as inputs may be different, which means that the input data structure may be flat (consisting of only attributes) or more complex – with attributes and associations. Possible visual representations of flat and complex data structures are illustrated by Figure 19 and Figure 20.

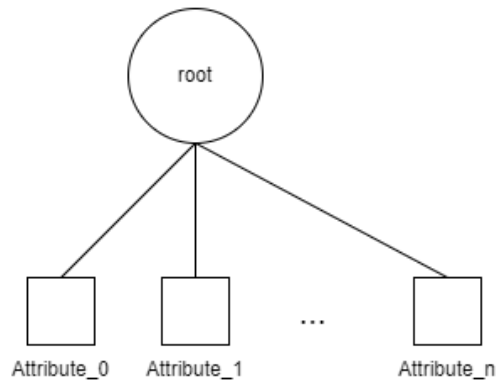


Figure 19 – Flat Data Structure in Dataspecer (*Source: Author*)

As evident, the flat data structure is relatively simple, because it only has two levels. There is the root node at the top level, whereas the second level consists of one or many attributes. The second level is the last level, since attributes in general point to something primitive and cannot have children [31]. Furthermore, Figure 20 represents the illustration for a possible deeper, complex data structure. In this case, not only attributes but also associations are present. The top level holds the root node again. However, the next levels may hold both – attributes and associations. The tree can expand further, because associations may have their own children – (zero or many) attributes and associations [3], [31].

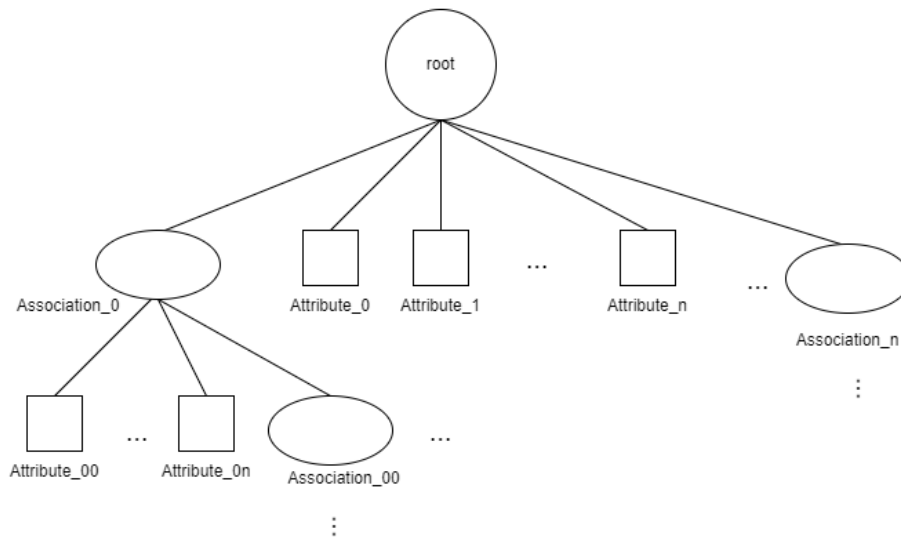


Figure 20 – Complex Data Structure (Source: Author)

As mentioned in subchapter 2.2, REST principles require resources to have a uniform representation in the response sent from the server [13]. This means that the schemas (models) have to be defined for the resources which will be utilized for this purpose. What's more, as discussed in subchapters 3.3 and 3.4 a good (REST) API should be well structured, which means that the structure of the designed models should be modular. Considering the context of this project, from the solution perspective this means that the schemas (models) should be (and are) defined not only for the root data structure, but also for all the associations present within. Despite the fact that the extension only allows operation creation for the root data structure and its successor associations one level below, it is crucial to define models for all associations. This is important for modularity, since the sub-models will be referenced where necessary instead of creating one big, unstructured schema of the whole input data structure. To be more precise, for each input data structure the number of defined *schemas* (in the *components* construct of corresponding OpenAPI specification) will be equal to $1 + \text{total number of associations present in the original input data structure schema}$. More precisely, the *schemas* of the *components* section include: schema for root data structure(s) and schemas for associations. What's more security schema is separately defined in the components construct. As mentioned, this results in the fact that the schema of the original (root) data structure will be able to reference the schemas of its successor associations (one level below). What's more, the schemas of these associations, in turn, reference the schemas of associations below them and so forth.

Particular examples are discussed in the later subchapters. Furthermore, REST principles require each resource model to have an identifier, so that the instances of these resources may be uniquely identified [15]. From the solution perspective of this project, it means that even though the data structure schemas created via the Dataspecer tool do not typically include an identifier, the corresponding schemas in the OpenAPI specification have to have a field *id* that would be utilized for the identification purpose. What's more, because of the fact that resources are identified via URI the program provides suggestions for endpoint (path) structures. These suggestions are based on the input data structures and utilize their names. For example, a suggested path for getting a particular *Tourist destination* would be “/Touristdestinations/{id}”. This gives the user an idea of what a path should look like and guides them in the direction of utilizing appropriate naming conventions. However, when it comes to the definition of paths, user makes the end decision, which means that it is the user's responsibility to provide actual path as input.

Another important aspect worth considering is compatibility of Dataspecer data structure and REST resource. Based on the information provided in 2.2 as well as in 5.1, in order to achieve a greater level of compatibility between the Dataspecer data structure and REST resource, a collection of operations (performing various manipulations on it) need to be defined. What's more, the API has to be stateless and the messages have to be self-descriptive [13]. This means that each request (therefore each operation defined in OpenAPI) has to contain sufficient information for the server to process. Because of this, this extension allows the user to specify most parts of the operation details – path, HTTP method as well as response code. What's more the UI shows short descriptions of the HTTP methods and response codes, which makes it easier for the user to make appropriate decisions when designing operations. The user also specifies for which data structure is this operation intended and if this operation is manipulating a collection or not. However, JSON as the content type, as well as utilization of Bearer tokens as the way of authentication, is set automatically by the program and cannot be changed by the user. The particular way of capturing user input is discussed in the next subchapters. Furthermore, when it comes to REST resources, they may be grouped into collections [15]. The program handles resource collections as well as single resource by utilizing “*manipulate a collection*” switch. This switch determines possible HTTP method options for particular operation. More precisely, if

this switch is on, only GET and POST methods are available. This means that the users can define operations to retrieve a collection of resources and to create a new resource (and add it to the collection). However, if this switch is off, possible options: GET, PUT, PATCH, DELETE refer to single resource instance and allow the user to define CRUD operations for it. It is important to note, that it is user's responsibility to set this switch correctly. As for the response generation, the response depends on the status code chosen by the user. In case of 200 (*OK*) and 201 (*CREATED*) [33] an instance of the resource is sent in the response. This instance conforms to the defined schemas (models) discussed above which again aligns with the REST principles, in particular – the response is sent in its consistent format. If the switch about collections is on, the response of a GET request will result in returning homogeneous collection of resource instances. Last but not least, another switch – “*association mode*” is utilized to state for which data structure (root or association one level below) is the operation defined. Detailed examples are provided in subchapters 5.3 and 5.4. Choosing a correct data structure fosters recommendation of a path structure (suggested path) which utilizes the name(s) of relevant resources. These suggested paths assist the user in determining the paths the API exposes. The program extracts path parameters from the provided endpoint (path) and generates query parameters (for filtering) automatically for GET requests. This means that the distinction between path and query parameters is provided by the program which is one of the characteristics of a good REST API as defined in 3.4. What's more the choice of the data structure determines the schema (model) referenced by the response. Like responses, requests are managed as well, by allowing the user to specify the request body only when needed – in the case of choosing POST or PATCH options.

Based on the requirements analysis provided in 5.1, it can be concluded that the most crucial input needed from the user is details of the operations (requests). Additionally, a way of API specification identification and versioning, needs to exist. Because of this, metadata also represents an essential part of the user-provided input. What's more, a base URL is of vital importance, since it represents a starting point for the paths exposed by the API. Therefore, the **information required as the user input** is:

- Metadata – API title, description and version
- Base URL

- Operation Details – Operation name, operation type, endpoint, request and response bodies as well as description. The user must specify if the operation is intended for the root data structure or its successor (association one level below). The information whether collection operation manipulation is performed or not needs to be provided as well.

As it was decided that this input has to be collected from the user, it is entered into the user interface (UI) and subsequently transferred to other components. To provide a brief overview, the system contains four primary architectural components – *Frontend*, *Dataspecer Backend*, *Fetcher* and *OAS Generator*. The *Frontend* component is specifically designed to receive information from user. The structure of the user-provided input is illustrated by Figure 21 below.

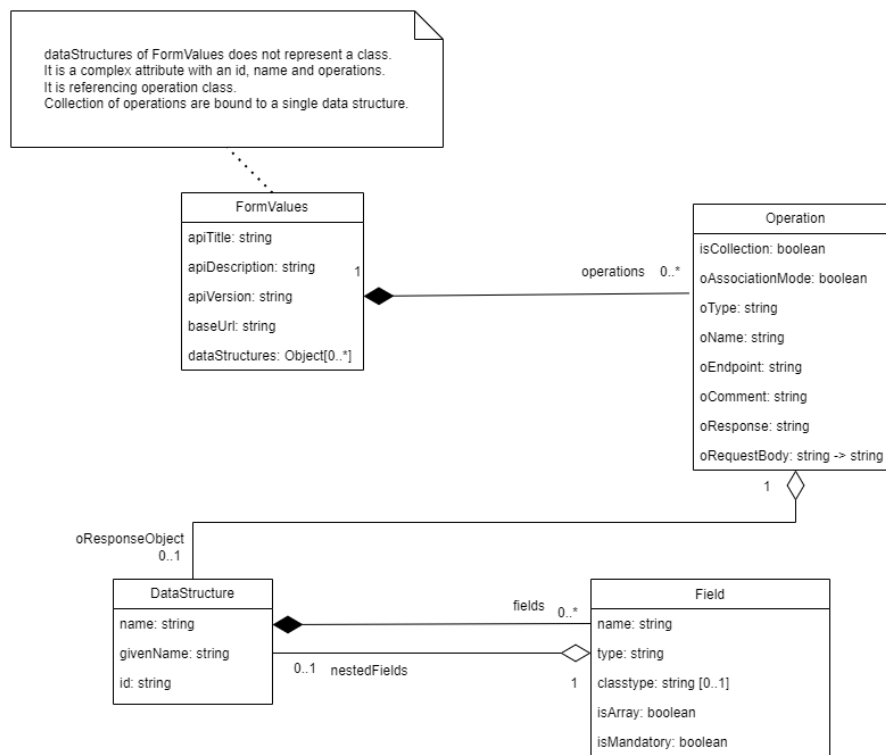


Figure 21 – Structure of User-Provided Input (*Source: Author*)

As evident, the user provided input is defined as form values. It not only contains the metadata – name of the API, description, version and base URL but also collection of simple data structures. These data structures do not represent an instance of some standalone class. It merely is a complex attribute referencing collection of *Operation*. Each data structure chosen by the user may be associated with multiple operations or none at all. More formally, a **collection of operations is bound to a single data**

structure chosen by the user via the UI. It is crucial to note, that these data structures are populated by selecting a desired data structure from a collection of fetched data structures from the *Dataspecer backend*. This means that a **connection** is made **between the user input and data structure from Dataspecer**. To be more precise, in each session of OpenAPI specification generation the user has access to the data structures contained by a single data specification. For instance, if target data specification *Furniture* in Dataspecer contains multiple data structures *Chair*, *Table* and *Sofa*, the user needs to select a data structure in the extension and then define operations for it. For example, if the user chooses *Chair* data structure the operations bound to it would ideally be – *CreateChair*, *DeleteChair*, etc. A sample user input object based on the running example of *Tourist destination* is illustrated by Figure 22 below.

```

apiDescription: "test"
apiTitle: "TouristDestinationsAPI"
apiVersion: "1.0"
baseUrl: "https://test.com"
▼ dataStructures: Array(1)
  ▼ 0:
    id: "d4198a6c-4b99-4893-94ce-345193ccf57e"
    name: "Tourist destination"
    ▼ operations: Array(5)
      ▶ 0: {name: '', isCollection: true, oAssociatonMode: true, oType: 'POST', oName: 'createContact', ...}
      ▶ 1: {name: '', isCollection: true, oAssociatonMode: false, oType: 'GET', oName: 'getTouristDests', ...}
      ▶ 2: {name: '', isCollection: false, oAssociatonMode: true, oType: 'PATCH', oName: 'PatchContact', ...}
      ▶ 3: {name: '', isCollection: false, oAssociatonMode: false, oType: 'PUT', oName: 'PutTouristDest', ...}
      ▶ 4: {name: '', isCollection: false, oAssociatonMode: false, oType: 'DELETE', oName: 'deleteTouristDest', ...}

```

Figure 22 – Sample User Input: Tourist destination (*Source: Author*)

Once operations are defined for the *Chair* data structure, the user can stop at this point and generate corresponding OpenAPI specification or continue in the same manner and choose a second data structure, for example – *Table* and also define operations such as *CreateTable*, *DeleteTable*, etc. for it, and generate the specification after. As discussed, **fetching** data regarding the data structures contained by the target data specification represents another **notable phase** of the whole process. More precisely, one of the main technical challenges is to represent data structures in a form that would be utilized by system components in an effective manner. As discussed, the data structure schemas designed via the Dataspecer tool have multiple layers. The associations present in the root data structure may also have such nested, multi-layered structure. An object of type *DataStructure* illustrated by Figure 23 is able to capture the nature of these data structures and their fields.

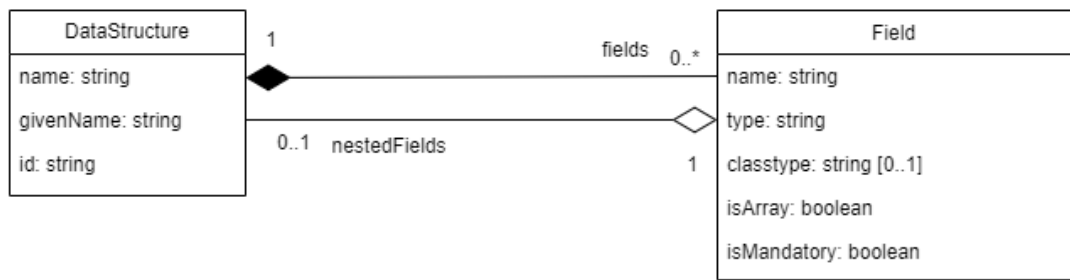


Figure 23 – DataStructure and Field Models (Source: Author)

As evident, there exist two associations between classes *DataStructure* and *Field*. On one hand, data structure may have zero or many fields. In particular, this is a case of a restricted aggregation since the object of type *Field* cannot exist without its container – object of type *DataStructure*. On the other hand, a field may have zero or one data structure. Because of their multi-layer structure, the fields may also be considered as data structures if they are marked as associations in the Dataspecer tool. If this is the case, *Field.type* is populated with the value – “Object” and *Field.classType* with the actual type. Furthermore, the entire nested structure is considered by populating *Field.nestedFields*. The representation of tourist destination data structure is illustrated by Figure 24 below.


```

▼ fields: Array(3)
  ▶ 0: {isMandatory: true, isArray: false, name: 'capacity', type: 'integer'}
  ▼ 1:
    classType: "human_or_person"
    isArray: true
    isMandatory: false
    name: "owner"
    ▼ nestedFields:
      ▶ fields: []
      givenName: undefined
      id: "ae0ac7d8-6361-41a9-be...-15a2f40985ae"
      name: "human_or_person"
      ▶ [[Prototype]]: Object
      type: "Object"
      ▶ [[Prototype]]: Object
    ▼ 2:
      classType: "contact"
      isArray: false
      isMandatory: true
      name: "contact"
      ▼ nestedFields:
        ▼ fields: Array(4)
          ▶ 0: {isMandatory: false, isArray: true, name: 'e-mail', type: 'string'}
          ▶ 1: {isMandatory: false, isArray: false, name: 'url', type: 'string'}
          ▶ 2: {isMandatory: false, isArray: false, name: 'phone_number', type: 'decimal'}
          ▶ 3: {isMandatory: false, isArray: true, name: 'has_contact', type: 'Object', classType: 'workplace', ...}
          length: 4
          ▶ [[Prototype]]: Array(0)
          givenName: undefined
          id: "996ca3f3-7563-4672-a661-16a5dbdalc5a"
          name: "contact"
          ▶ [[Prototype]]: Object
          type: "Object"
          ▶ [[Prototype]]: Object
          length: 3
          ▶ [[Prototype]]: Array(0)
          givenName: "Tourist destination"
          id: "b0b6e181-780b-46d5-b62e-005010be6b26"
          name: "tourist_destination"

```

Figure 24 – Example of DataStructure Object - Tourist destination (Source: Author)

As demonstrated by Figure 24, tourist destination represents the root data structure. Root data structure’s attributes and associations are represented by the *fields* property. When it comes to an attribute (in this case *capacity*), its type (integer) is directly written in the type property. *Contact* and *owner* represent associations of the root data structure. Because of the fact that they represent associations, meaning that they may also have nested, multi-layered structures (for instance, *contact*), their type property holds value “*Object*” while the *classType* is populated with the actual type. As for their actual nested structure, it is represented by *Field.nestedFields* property. This is highlighted by the yellow arrows on Figure 24.

As for the actual OpenAPI specification generation process, there are two parameters needed. The first parameter represents input from the user, illustrated by Figure 21, whereas the second parameter represents the collection of data structures retrieved from the *Dataspecer backend* illustrated by Figure 23. The generator consolidates this information and generates corresponding OpenAPI specification as illustrated by Figure 25 below.

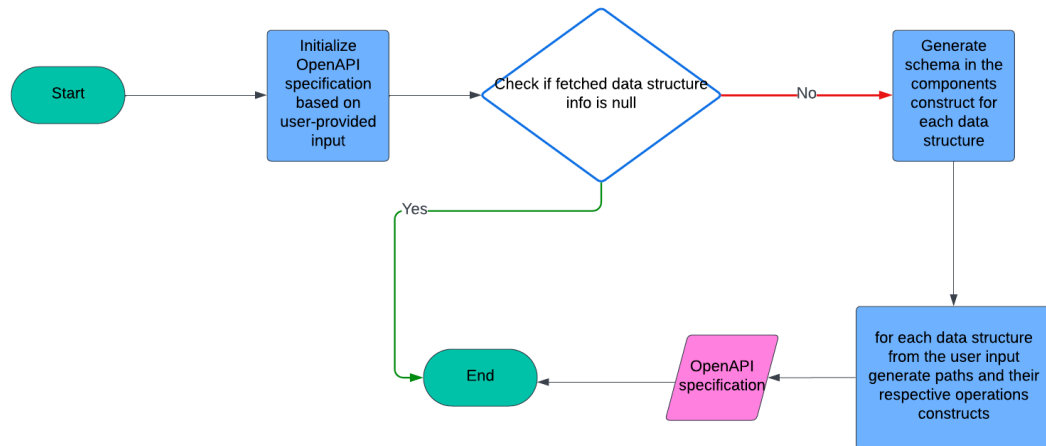


Figure 25 – High-Level Flow Diagram: OAS Generation (Source: Author)

As illustrated by Figure 25, the OAS generation algorithm firstly initializes the OpenAPI specification. This means that all the necessary constructs described in 2.4 are initialized. During the initialization process the values for the simpler constructs such as – metadata, base URL as well as security are set. More complex constructs such as *components* and *paths* are initialized as empty constructs. Next the program checks if the collection of fetched data structures is not empty. If the collection is not empty, the schemas for the data structures are generated which means that the value for the components construct is set. Next the program creates paths and corresponding operation(s) constructs. Having completed this step, the OpenAPI specification generation is finalized and is displayed to the user. More technical details regarding the operation of the components – *Fetcher*, *Frontend* as well as *OAS Generator* are discussed in chapters 5.5 – Architecture and 5.6 – Implementation.

5.3 Extension Demonstration

The aim of this subchapter is to consider UI and in **particular discuss and analyze the process of capturing user input (discussed in 5.3.1)**. Dataspecer tool is currently in the process of migration. A new component Dataspecer Manager is introduced which will represent a starting point for all features of the tool. The extension is accessible through Dataspecer Manager. The information for accessing this extension on production environment, as well as, build instruction for the local environment are provided in the Attachments section – A.1 and A.2. The source code is attached electronically and A.3 showcases the structure of the electronic attachment.

Upon the initial access of the extension, the user is presented with a page, where the fields of the form are empty by default. The interface is divided into two parts. The left side is dedicated for the form and its inputs whereas the right side serves the purpose of displaying the generated output OpenAPI specification corresponding to the form data as illustrated by Figure 26.

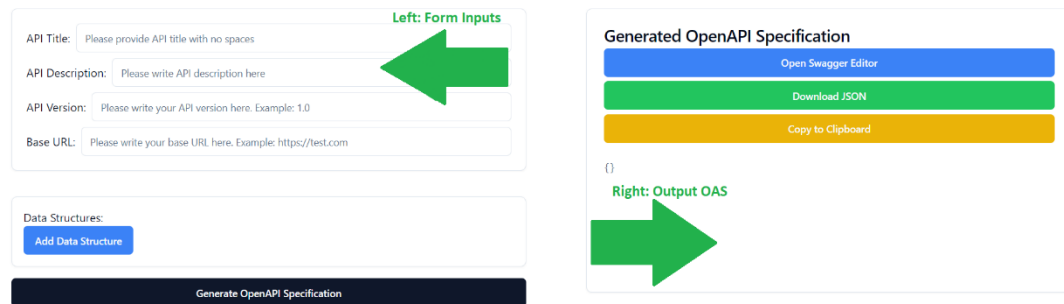


Figure 26 – Dataspecer Extension: Initial View (Source: Author)

As evident, metadata such as title, description, version and base URL have to be provided and data structure has to be added by clicking “Add Data Structure” button. Once this is done, the user is able to choose a data structure for which in the next stage the operations are defined by clicking “Add Operation” button as exemplified by Figure 27.

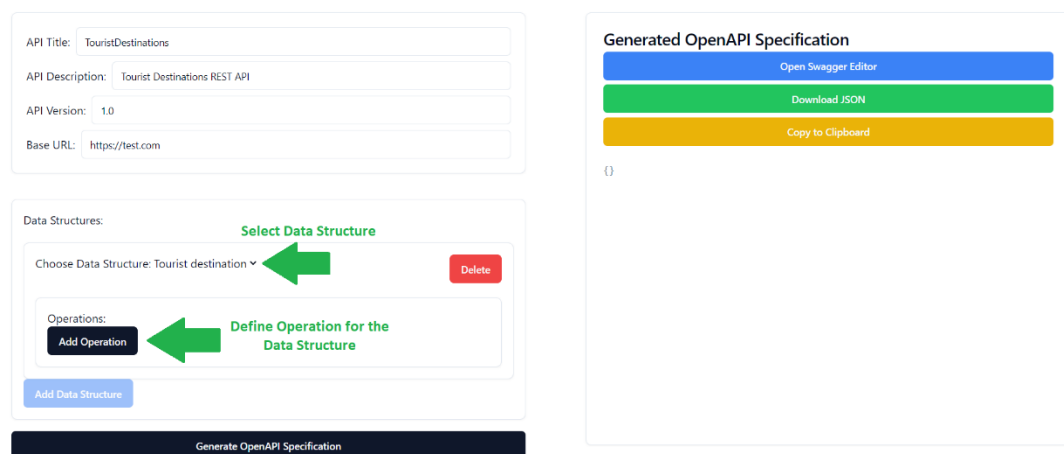


Figure 27 – Dataspecer Extension: Select Data Structure and Start Defining Operations (Source: Author)

Once all of the operations are defined “Generate OpenAPI Specification Button” needs to be clicked, which results in the OpenAPI specification generation. At this point the user-provided input is saved as well and will be displayed to the user on the next visit.

As illustrated by Figure 28, this specification is displayed on the right side of the page and the user is then able to either download its JSON representation or copy it and continue its exploration in the swagger editor.

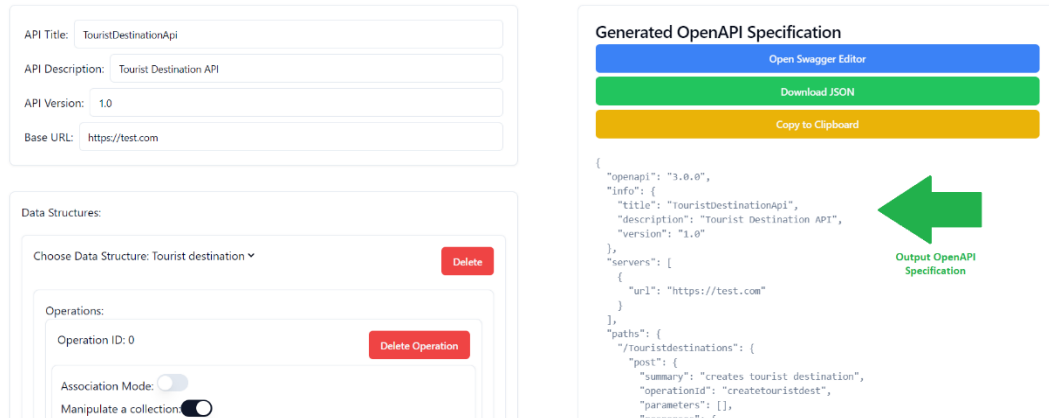


Figure 28 – Dataspecer Extension: Output (Source: Author)

5.3.1 Capturing User Input and its Conceptual Alignment

As mentioned prior, during the requirements analysis (in subchapter 5.1) there exists a gap that needs to be filled – in particular, the information regarding the operations performing different manipulations on the Dataspecer data structures needs to be obtained. As noted in 5.1, after careful consideration of different options, it was concluded to allow the users to specify the operation (request) details. This information is captured through the form in the user interface. *OperationCard* component (displayed on Figure 29) appears once “Add Operation” button is clicked and is utilized for this purpose.

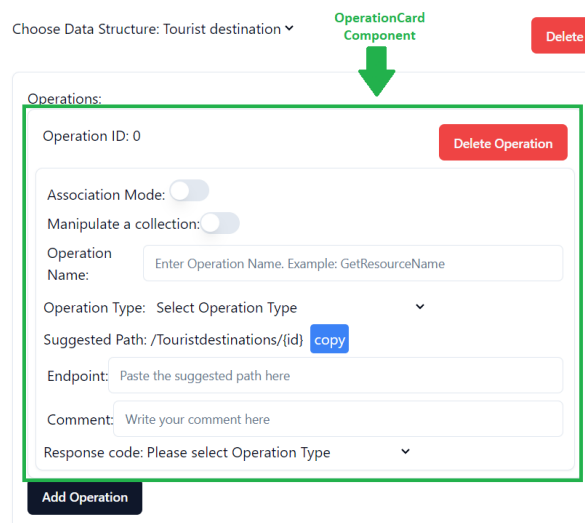


Figure 29 – Dataspecer Extension: OperationCard Component (Source: Author)

As evident, this component consists of multiple sub-components such as: association mode, collection mode (manipulate a collection), operation name, operation type, suggested path, endpoint, comment as well as a response code. Each of these sub-components serve a different purpose. Table 8 provides the details below.

OperationCard Sub-Component Name	Purpose
Association Mode	Specifies if the operation is intended for the root data structure (for instance, <i>Tourist destination</i>) or its successor one level below (in the context of running example – <i>owner</i> or <i>contact</i>).
Collection Mode	Specifies if the operation is performing a collection or a single resource manipulation.
Operation Name	Specifies the name of the operation.
Operation Type	Specifies the type of the operation. Possible options are: GET, POST, PUT, PATCH and DELETE.
Suggested Path	Suggests the structure of the operation path (endpoint) to the user.
Endpoint	Specifies the path (endpoint) of the operation. In most cases it is recommended to paste the value of the suggested path in this field.
Comment	Specifies the summary of the operation.
Response Code	Specifies the response code for the operation. Possible options are: 200, 201, 204, 400, 401, 500.

Table 8 – OperationCard Sub-Components (*Source: Author*)

As mentioned, these type of card needs to be created for each operation that needs to be present in the resulting OpenAPI specification. The contents of operation card are mapped to the operations subconstruct as well as the path construct of the OAS mentioned in 2.4.1. This section will examine two examples of user-defined operations

(requests) and their alignment with the output OpenAPI specification in order to enhance the reader's understanding.

The first operation (request) aims to retrieve a collection of *Tourist destinations*. Since this operation is intended for the root data structure, the association mode switch is off. However, because of the fact that a collection of resources is being retrieved, collection mode switch is on. Next a name for the operation is provided as well as an endpoint specifying a path exposed by the API. What's more, the program also generates a suggested path which suggests the path structure. Furthermore, it is recommended to **utilize the structure of this path for the endpoint for the target request**. Lastly an optional comment is added to the operation and a desired response code is assigned. This is initially reflected by the UI when filling out the operation card component for this operation, and later after generating the API specification, this operation is reflected in the paths construct under one of the *operation* subconstructs. This can be observed in Figure 30 (UI) as well as Figure 31 (OAS) below.

Operation ID: 1 Delete Operation

Association Mode: ←

Manipulate a collection:

Operation Name: ←

Operation Type: GET - Retrieve a collection of resources ← ▾

Suggested Path: /Touristdestinations copy

Endpoint:

Comment: ←

Response code: 200 - successful request ← ▾

Figure 30 – Retrieve Tourist Destinations: UI representation (*Source: Author*)

```
    "get": {
      "summary": "retrieves a collection of tourist destinations",
      "operationId": "gettouristdests",
      "parameters": [ ...
    ],
    "responses": {
      "200": {
        "description": "200 - successful request",
        "content": {
          "application/json": {
            "schema": {
              "type": "array",
              "items": {
                "$ref": "#/components/schemas/tourist_destination"
              }
            }
          }
        }
      }
    }
  },
}
```

Figure 31 – Retrieve Tourist Destinations in OAS (Source: Author)

The second operation serves the purpose of creating new contact in the context of tourist destinations. What’s more, the logic of the second operation proceeds in the similar manner as the first. The key difference is that the second operation is not intended for the root data structure – *Tourist destination*, but its successor (one level below) – *contact*. Because of this, association mode switch is on which enables the user to choose a new (successor) data structure. In this case collection mode switch is on and POST is chosen as the HTTP method. This means that a new resource is created and added to the collection. It is notable that the suggested path has evolved, indicating that this resource is connected to another resource – the root, *Tourist destination*. This is illustrated by Figures 32 (UI) and 33 (OAS) below.

Operation ID: 2 Delete Operation

Association Mode: ← → Target Datastructure: contact ▾

Manipulate a collection:

Operation Name: ←

Operation Type: POST - Create a new resource within the collection ▾ ←

Request Body ←

e-mail

url

phone_number

has_contact

Suggested Path: /Touristdestinations/{id}/contacts copy ←

Endpoint: ←

Comment: ←

Response code: 201 - new resource created ← ▾

Figure 32 – Create Contact: UI Representation (Source: Author)


```
1  "/Touristdestinations/{id}/contacts": {
2    "post": {
3      "summary": "creates contact",
4      "operationId": "createcontact",
5      "parameters": [
6        {
7          "name": "id",
8          "in": "path",
9          "required": true,
10         "schema": {
11           "type": "string"
12         }
13       }
14     ],
15     "responses": {
16       "201": {
17         "description": "201 - new resource created",
18         "content": {
19           "application/json": {
20             "schema": {
21               "$ref": "#/components/schemas/contact"
22             }
23           }
24         }
25       }
26     },
27     "requestBody": {
28       "required": true,
29       "content": {
30         "application/json": {
31           "schema": {
32             "type": "object",
33             "properties": {
34               "url": {
35                 "type": "string"
36               },
37               "phone_number": {
38                 "type": "number",
39                 "format": "double"
40               },
41               "has_contact": {
42                 "$ref": "#/components/schemas/workplace"
43               }
44             }
45           }
46         }
47       }
48     }
49   }
50 }
```

Figure 33 – Create Contact in OAS (Source: Author)

To sum up, the contents of the *OperationCard* components are utilized in order to create paths and its respective operation sub-constructs in the output OpenAPI specification. To be more precise, endpoint which in most cases is the same as the suggested path, represents the key to the *path* object of *paths* collection in the OpenAPI specification. Operation type determines the *operation type* inside the path object. The rest of the fields are utilized for populating the sub-sections of the *operation* sub-

construct. For instance, if appropriate, the operation in the OAS contains the request body component specified in the UI. What's more, the combination of the association mode, chosen data structure (either root or successor) as well as the response are utilized for constructing *responses* section. Lastly, the optional comment populates the *summary* of the operation in the OpenAPI specification.

5.4 Output OAS

Previous chapter demonstrated only a portion of the output OAS – mainly the examples of path constructs. This chapter will focus on the output OpenAPI specification as a whole. More specifically, this chapter considers a scenario where **first version** of an OpenAPI specification needs to be created according to the following description – API specification with the title “**TouristDestinationsAPI**” aims to manage tourist destinations. The base URL for initiating all requests of the API is: “**https://test.com**” The API (and therefore the specification) has to support following requests (operations):

- Retrieve collection of *Tourist destinations*
- Create an instance of *Tourist destination* with *capacity* and *contact* fields (and add to a collection of *Tourist destinations*)
- Update instance of *Tourist destination* fully (entire instance of a particular *Tourist destination* has to be updated.)
- Delete a particular instance of *Tourist destination*
- Create contact information (*contact*) with *phone number*, *email* and *has_contact* fields for particular *Tourist destination*
- Update contact information (*email* and *phone number* of a *contact*) of a particular tourist destination

This chapter considers output OpenAPI specification which was produced via this extension of the Dataspecer tool. In particular, running example of *Tourist destination* was utilized as the root data structure as shown in 5.3 to produce desired OpenAPI specification. As demonstrated by Figure 34, output OAS contains following constructs: *openapi*, *info*, *servers*, *paths*, *components* and *security*.

```

1  {
2  "openapi": "3.0.0",
3  > "info": { ...
7  },
8  > "servers": [ ...
12 ],
13 > "paths": { ...
269 },
270 > "components": { ...
350 },
351 > "security": [ ...
355 ]
356 }

```

Figure 34 – Structure of Tourist Destination OAS (Source: Author)

As indicated, the *openapi* construct shows the version of the OpenAPI itself. The *info* and *servers* constructs are populated according to the description as illustrated by Figure 35. It is important to note, that from the perspective of the extension program this information is provided by the user.

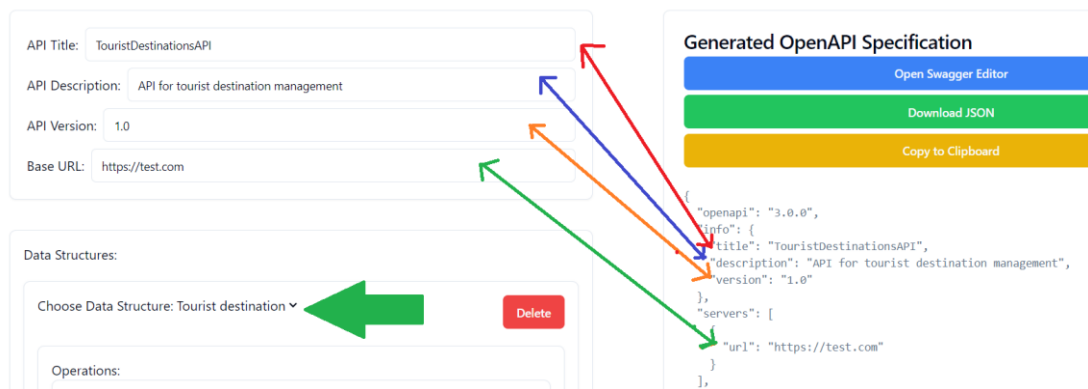


Figure 35 – Mapping of Metadata between UI and OAS (Source: Author)

5.4.1 Paths and Operations

Figure 35 also shows that *Tourist destination* was chosen as the data structure for which the desired operations are defined. Operation definition follows the process described in 5.3.1 which means that for each *OperationCard* filled in by the user corresponding operation sub-construct is created inside their respective paths construct. For instance, the operations for *Tourist destinations* retrieval and its creation, represent examples of collection manipulation. The operation retrieving *Tourist destinations*, retrieves a collection of objects of *Tourist destination* whereas the request creating a new instance, creates a new *Tourist destination* and adds it to

the collection of tourist destinations. Because of this, the suggested path for both of these operations would be *"/Touristdestinations"*. This means that an object in the paths construct with a key *"/Touristdestinations"* would be a parent for both of these operations which is indeed the case as shown by Figure 36 and Figure 37 below. Figure 36 illustrates operation sub-construct for *Tourist destination* retrieval (GET *Tourist destinations*). The fields *summary*, *operationId* as well as the *responses* are filled in according to the user-provided input. More precisely, the field *summary* corresponds to the comment of the *OperationCard* component and intends to provide documentation for the operation which in turn is a characteristic of a good (REST) API. The *operationId* corresponds to name of the operation. What's more, responses object is constructed according to the status code set in the UI (200 in this case). Query parameters of the operation-sub construct are generated automatically according to the fields of the data structure for which this request (operation) is intended. This also reflects intension of designing a good API since a distinction between path and query parameters are provided by the program. This particular GET operation is intended for the root data structure – *Tourist destinations* that has three fields – *capacity*, *owner* and *contact* on its own. The field *id* is appended by the program automatically since usually the data structures designed via the Dataspecer tool lack a unique identifier which is an essential part of a REST resource (the process of adding *id* field to the data structures is described in more detail in 5.6 Implementation). The request allows the user to filter the collection of retrieved resources based on its fields – in this case: *capacity*, *owner*, *contact* and *id*.

```

1  paths": {
2    "/Touristdestinations": {
3      "get": {
4        "summary": "retrieves a collection of tourist destinations",
5        "operationId": "gettouristdests",
6        "parameters": [
7          {
8            "name": "capacity",
9            "in": "query",
10           "description": "Filter results based on capacity",
11           "schema": {
12             "type": "integer"
13           }
14         },
15         {
16           "name": "owner",
17           "in": "query",
18           "description": "Filter results based on owner",
19           "schema": {
20             "type": "array",
21             "items": {
22               "$ref": "#/components/schemas/human_or_person"
23             }
24           }
25         },
26         {
27           "name": "contact",
28           "in": "query",
29           "description": "Filter results based on contact",
30           "schema": {
31             "$ref": "#/components/schemas/contact"
32           }
33         },
34         {
35           "name": "id",
36           "in": "query",
37           "description": "Filter results based on id",
38           "schema": {
39             "type": "string"
40           }
41         }
42       ],
43       "responses": {
44         "200": {
45           "description": "200 - successful request",
46           "content": {
47             "application/json": {
48               "schema": {
49                 "type": "array",
50                 "items": {
51                   "$ref": "#/components/schemas/tourist_destination"
52                 }
53               }
54             }
55           }
56         }
57       }
58     },

```

Figure 36 – GET Tourist destinations: Sample Operation (Source: Author)

As for the creation of *Tourist destination*, this POST request/operation belongs to the same path and follows a similar structure as previously discussed GET request. *Summary* and *operationId* are populated according to the user-input. According to the description, a tourist destination is created according to two parameters – *capacity* and *contact*. This is reflected in the request body. As shown, it is specified in the OAS that this request body is required and its content holds a request body schema with two parameters – *capacity* (a simple integer) and a *contact*. The *contact* represents an association in the original Dataspecer data structure and has its own multi-layered schema. Because of this, it has its own dedicated schema in the resulting OAS which will be discussed later in this chapter. Therefore, instead of merely stating its type, the request body references the schema, ensuring clarity of the OpenAPI specification. This was done with the intension of appealing to the good (REST) API characteristics and improve structure and usability of the output OAS.

```
59     "post": {
60       "summary": "creates tourist destinations",
61       "operationId": "createtouristdest",
62       "parameters": [],
63       "responses": {
64         "201": {
65           "description": "201 - new resource created",
66           "content": {
67             "application/json": {
68               "schema": {
69                 "$ref": "#/components/schemas/tourist_destination"
70               }
71             }
72           }
73         }
74       },
75       "requestBody": {
76         "required": true,
77         "content": {
78           "application/json": {
79             "schema": {
80               "type": "object",
81               "properties": {
82                 "capacity": {
83                   "type": "integer"
84                 },
85                 "contact": {
86                   "$ref": "#/components/schemas/contact"
87                 }
88               }
89             }
90           }
91         }
92       }
93     }
94   },
```

Figure 37 – POST Tourist destination: Sample Operation (Source: Author)

The rest of the operations specified in the description follow the same structure. The difference is that these operations are children of different object within the paths construct. The operations of (full) update of *Tourist destination* and its deletion are bound to a particular tourist destination. Which means that suggested path would be *"/Touristdestinations/{id}"* indicating that the request is intended for a specific resource. As for the last two operations, they are intended for the successor of the root data structure – *contact*. Figures 38 and 39 show the corresponding operations constructs of tourist destination full update (PUT) and its deletion. As shown, the request body of the PUT request references the schema of tourist destination from the components construct. Since the request is handling full update of the object, the user does not need to specify the request body – it is generated automatically by the program.

```
"/Touristdestinations/{id}": {
  "put": {
    "summary": "Full update of tourist destination",
    "operationId": "fullupdatetouristdest",
    "parameters": [ ...
  ],
  "responses": {
    "204": { ...
  }
},
"requestBody": {
  "required": true,
  "content": {
    "application/json": {
      "schema": {
        "$ref": "#/components/schemas/tourist_destination"
      }
    }
  }
}
},
"delete": { ...
```

Figure 38 – PUT Tourist destination: Sample Operation (Source: Author)

```

"/Touristdestinations/{id}": {
  "put": {...
  },
  "delete": {
    "summary": "deletes particular tourist destination",
    "operationId": "deletetouristdest",
    "parameters": [
      {
        "name": "id",
        "in": "path",
        "required": true,
        "schema": {
          "type": "string"
        }
      }
    ],
    "responses": {
      "204": {
        "description": "204 - success with no content to return",
        "content": {
          "application/json": {
            "schema": {}
          }
        }
      }
    }
  }
}

```

Figure 39 – DELETE Tourist destination: Sample Operation (*Source: Author*)

The operations intended for the successor (one level below) – follow in the same manner as the previous ones described above. Figure 40 illustrates creation of contact. As shown by Figure 40, the request body satisfies the initial description by including three desired parameters – *phone number*, *email* and *has_contact*. The response body reflects the nature of the chosen response code 201. It states that created instance has to be returned with the 201 response code. Operation *summary*, *operationId* as well as the endpoint (*path*) are filled based on the values provided via the UI.


```

1  "/Touristdestinations/{id}/contacts": {
2    "post": {
3      "summary": "creates contact for tourist destination",
4      "operationId": "createcontact",
5      "parameters": [
6        {
7          "name": "id",
8          "in": "path",
9          "required": true,
10         "schema": {
11           "type": "string"
12         }
13       }
14     ],
15     "responses": {
16       "201": {
17         "description": "201 - new resource created",
18         "content": {
19           "application/json": {
20             "schema": {
21               "$ref": "#/components/schemas/contact"
22             }
23           }
24         }
25       }
26     },
27     "requestBody": {
28       "required": true,
29       "content": {
30         "application/json": {
31           "schema": {
32             "type": "object",
33             "properties": {
34               "e-mail": {
35                 "type": "array",
36                 "items": {
37                   "type": "string"
38                 }
39             },
40             "phone_number": {
41               "type": "number",
42               "format": "double"
43             },
44             "has_contact": {
45               "$ref": "#/components/schemas/workplace"
46             }
47           }
48         }
49       }
50     }
51   }
52 }
53 }

```

Figure 40 – POST contact: Sample Operation (Source: Author)

5.4.2 Components

Components construct is build based on the data structures located in the target data specification. Discussed data specification contains one data structure – *Tourist destination* (running example) which means that the *components* construct reflects its

structure. Moreover, as illustrated by Figure 41 this construct contains security schema of the API.

```
"components": {  
  "schemas": {  
    "human_or_person": { ...  
  },  
  "workplace": { ...  
  },  
  "contact": { ...  
  },  
  "tourist_destination": { ...  
  }  
},  
  "securitySchemes": {  
    "bearerAuth": {  
      "type": "http",  
      "scheme": "bearer",  
      "bearerFormat": "JWT"  
    }  
  }  
}
```

Figure 41 – Security Schema in OAS (Source: Author)

As shown, the *components* construct contains schemas for all data structures – the root data structure as well as all associations defined within (regardless of the level). What’s more, this construct holds the information regarding authentication. The program utilizes Bearer Authentication, which means that only the bearers of an access token are able to access the API [34]. *Tourist_destination* represents the schema of the root data structure – *Tourist destination*. This schema is constructed based on the data structure designed via the Dataspecer tool. As shown by Figure 42, like Dataspecer data structure *Tourist destination*, the corresponding schema has following fields (properties) – *capacity*, *owner* (of type *human_or_person*) and *contact* (of type *contact*). Additionally, it contains field *id* which ensures that this resource is identifiable. Since *owner* and *contact* represent associations in the running example, the schemas are also defined for their respective types (classes) – *human_or_person* and *contact*. Figure 43 illustrates these schemas and their connection to the Dataspecer data structure. It is important to note, that there is no structure defined for the *owner*. Because of this, the description of its corresponding schema notifies the user that this component needs to be filled in. This enhances documentation of the OAS.

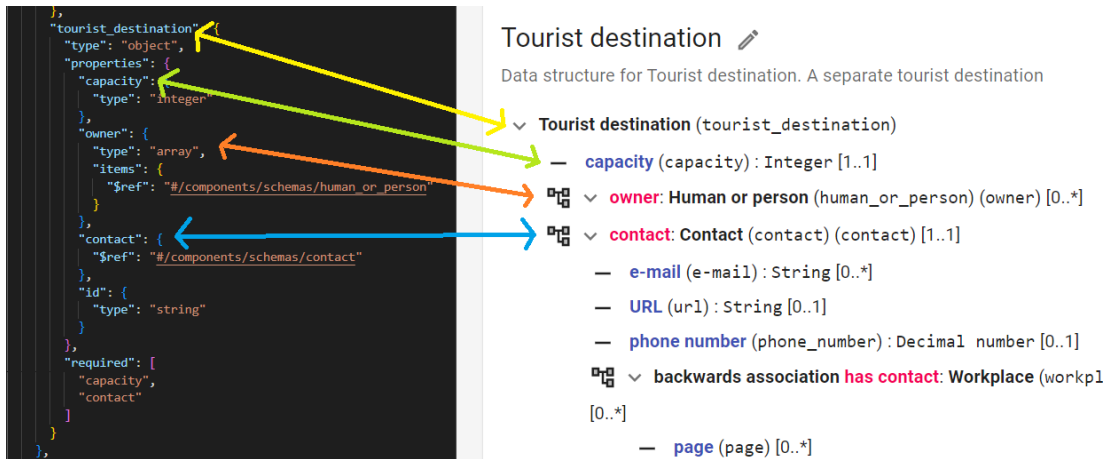


Figure 42 – Tourist Destination Schema in Dataspecer and in OAS (Source: Author)

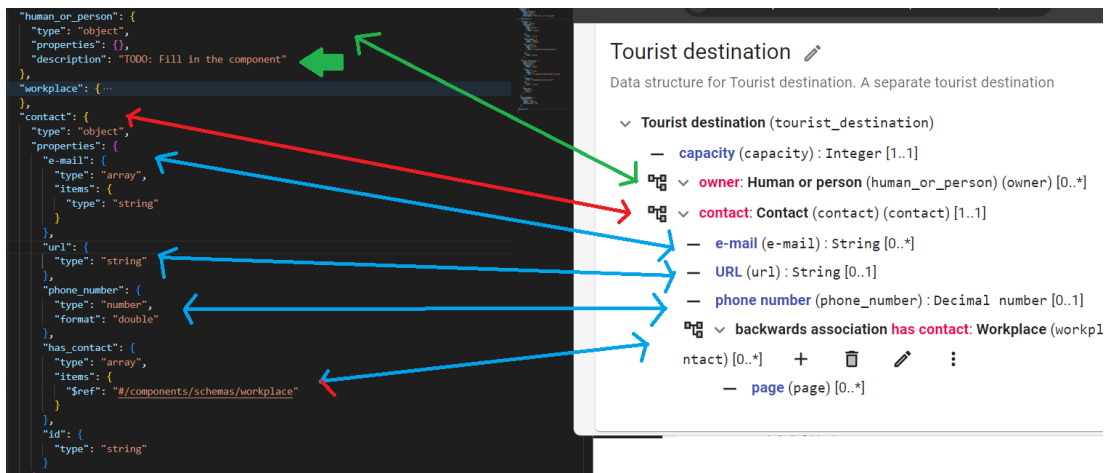


Figure 43 – Association Schemas in Dataspecer and in OAS (Source: Author)

What’s more, it has to be acknowledged that if the data specification included multiple data structures, all of their schemas would be reflected in the components construct in the same way as it is done for the running example.

5.4.3 Supported Constructs in the Output OAS

As demonstrated by the examples provided in the previous sub-chapters (5.4.1 and 5.4.2), output OAS supports basic constructs of the OpenAPI specification described in sub-chapter 2.4. To be more precise, the output OAS contains metadata as well as base URL of API requests. More complex constructs contained by the output are *paths* with their corresponding *operation* sub-constructs and *components*. Last but not least, the output contains *security* construct specifying the authentication approach of the API. Supported constructs [9] as well as explanations of what their support entails is encompassed by Table 9 below. It the format of output OAS is JSON. Generation of YAML is not supported.

Construct of OpenAPI Specification	General Description	Support in the generated OAS
Metadata: openapi	Specifies OpenAPI version	<ul style="list-style-type: none"> • The version is automatically set by the program to 3.0.0 • The users are not able to change the version
Metadata: info	Specifies metadata – title, description and version of the API	<ul style="list-style-type: none"> • The user is able to set title, description and version of the API
Servers	Specifies the base URL of the API server. Generally, OpenAPI supports having multiple servers.	<ul style="list-style-type: none"> • The user is allowed to set (only one) base URL. • Multiple servers are not supported.
Paths	Specify individual paths (endpoints) and operations defined via HTTP methods for these endpoints.	<ul style="list-style-type: none"> • Paths and their corresponding operation constructs and sub-constructs are created based on the user input. • Extraction of path parameters is supported. The user has to specify path parameters. • Automatic generation of query parameters is supported for GET requests. • Parameter types other than path and query are not supported. • Request (where applicable) as well as response bodies are supported for each operation in each path construct.

		<ul style="list-style-type: none"> • Multiple request bodies/response bodies are not supported. • Only following response codes are supported: 200, 201, 204, 400, 401 and 500. • Content-type is set to JSON. The user is not able to modify it.
Input and Output Models	Specify common definitions of the schemas utilized across the OAS	<ul style="list-style-type: none"> • Components construct and their corresponding schemas are generated based on the data structures designed via the Dataspecer tool. • Multi-layer nested structures are represented by referencing necessary schemas.
Authentication	Specifies authentication methods utilized in the API	<ul style="list-style-type: none"> • Authentication is automatically set to Bearer token authentication. • Changing authentication method is not supported.

Table 9 – Supported Constructs of the Output OAS (Source: Adapted from [9])

5.5 Architecture

The aim of this section is to discuss the technical aspects such as project architecture as well as implementation details. The source code of this project may be found in the electronic attachment (A.3). The source code located in the *src* directory is organized into several subdirectories each of which serve a different purpose. These subdirectories are: *components*, custom components (*customComponents*), *models* as well as *props*. The *components* directory represents a repository for the components that were imported from shadcn-ui library. As for the *custom components*, this is where bespoke components reside. To be more precise, they achieve the goal of meeting the distinct needs of the project. Custom components combine not only pre-imported but also handcrafted HTML components which ensure seamless form operation. As the name suggests *models* represents a repository where the typescript

types as well as interfaces utilized across the whole codebase reside. As for the *props*, this folder holds the data about the properties which are being passed to the aforementioned custom components. Furthermore, multiple notable files are located directly in the *src* directory. *DataStructureFetcher.tsx*, *DataTypeConverter.tsx*, *FormValidationSchema.tsx*, *OApiGenerator.tsx* and *MainForm.tsx*. hold the logic of the key architectural components and will be discussed in a detailed manner in the later parts of this section.

Having discussed the general organization, architectural perspective of the project may be considered. At first, the placement of this extension will be considered in the context of the overall Dataspecer tool architecture. As mentioned above, Dataspecer represents a complex tool. This means that it consists of many architectural components. However, there are three key components which are relevant in the context of this projects scope. These components are: Dataspecer Backend, Structure Editor (discussed in 4.2) and Dataspecer Manager. Each of these architectural components have their distinctive purpose:

- **Dataspecer Backend** – is a critical component in the overall Dataspecer architecture. It represents a central hub when it comes to communication in the Dataspecer tool ecosystem. Dataspecer Backend component communicates with other components and facilitates data exchange.
- **Structure Editor** – serves the purpose of creation and management of data structures.
- **Dataspecer Manager** – represents a primary interface for initiating various functionalities of the Dataspecer tool.

Component API-Specification Generator (this extension project generating OpenAPI specifications) is located on the same level as these three components. It also communicates with the Dataspecer Backend in order to store and receive data. The high-level representation of these components and their relationship with each other is illustrated by Figure 44 below.

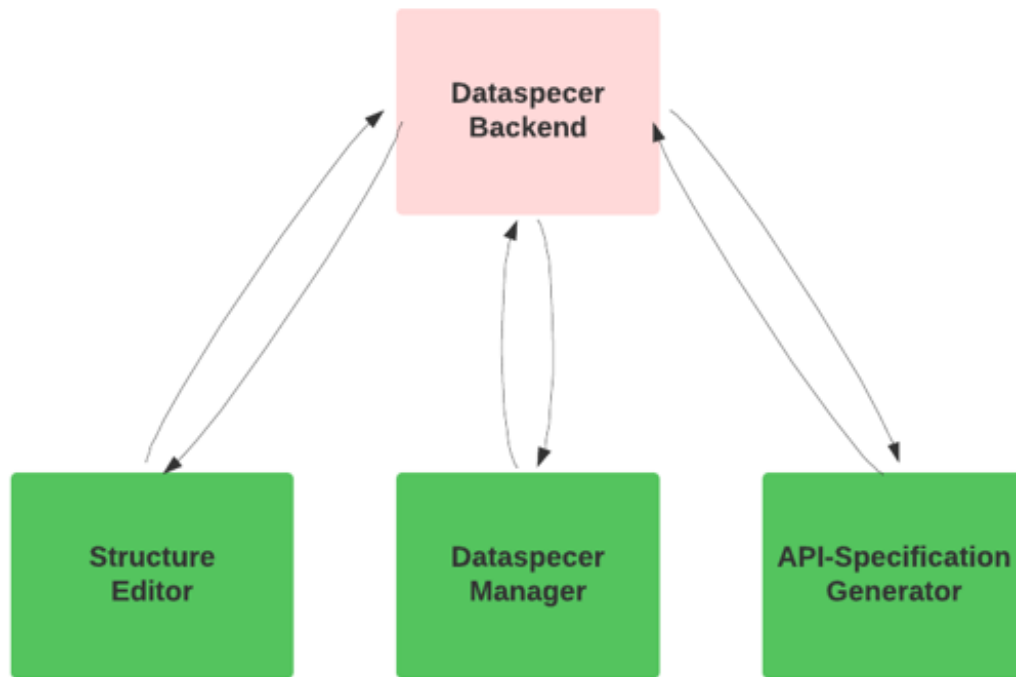


Figure 44 – High-Level Architecture: Integration with Dataspecer *(Source: Author)*

Having considered the architectural perspective within the broader context, the architecture of API-Specification Generator component may be discussed. When it comes to API-Specification Generator the main components are:

- **Fetcher** – Fetches and processes information regarding the data structures in the target data specification from the Dataspecer Backend.
- **Frontend**
 - Gets data (metadata and operation/request details) from the user.
 - Receives the information about the data structures fetched from the Fetcher component.
 - Receives (if available) pre-saved configuration (operation details) from the Dataspecer Backend.
 - Sends all of the information mentioned to the Generator.
 - Sends the configuration – metadata and request (operation) details to the Dataspecer Backend for future maintenance.
- **OAS Generator** – consolidates data sent by the Frontend component and generates corresponding OpenAPI specification.

The high-level interpretation of the flow is illustrated by Figure 45 below.

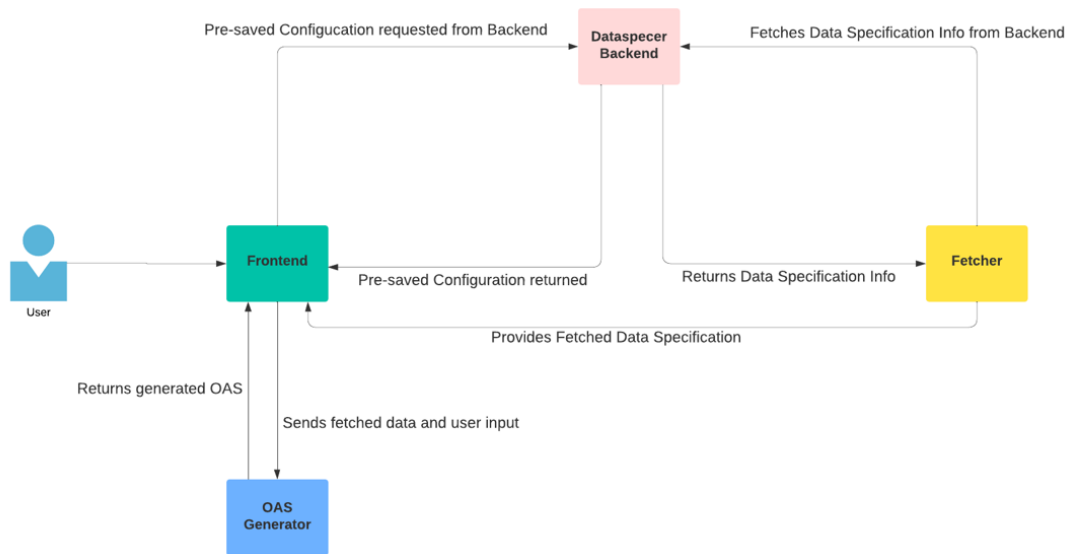


Figure 45 – API-Specification Generator: High-Level Flow Diagram (Source: Author)

5.6 Implementation

As mentioned in 5.4, there are several architectural components to this project. One of the most crucial components is the **Fetcher**. The logic of this component is located in *DataStructureFetcher.tsx*. The Fetcher component is responsible for the **integration of this extension within the Dataspecer tool**. More precisely, it serves the purpose of retrieving data regarding desired data structures from the Dataspecer backend and conforming them to the form illustrated by Figure 23 from 5.2. Fetching data structure information represents the initial step when it comes to program operation. In particular, the URL of the current window of the browser holds an id which identifies target data specification. The fetcher utilizes this id and retrieves information regarding target data specification. This data contains unique identifiers for the data structures defined in the data specification. Therefore, as the next step the information about each individual data structure is obtained, resulting in a collection of unprocessed data structures in their raw form. Having retrieved a collection of unprocessed data structures, the processing may start. Once the processing phase is complete, a collection of the processed data structures is produced. Figure 46 illustrates the high-level flow of the Fetcher component.

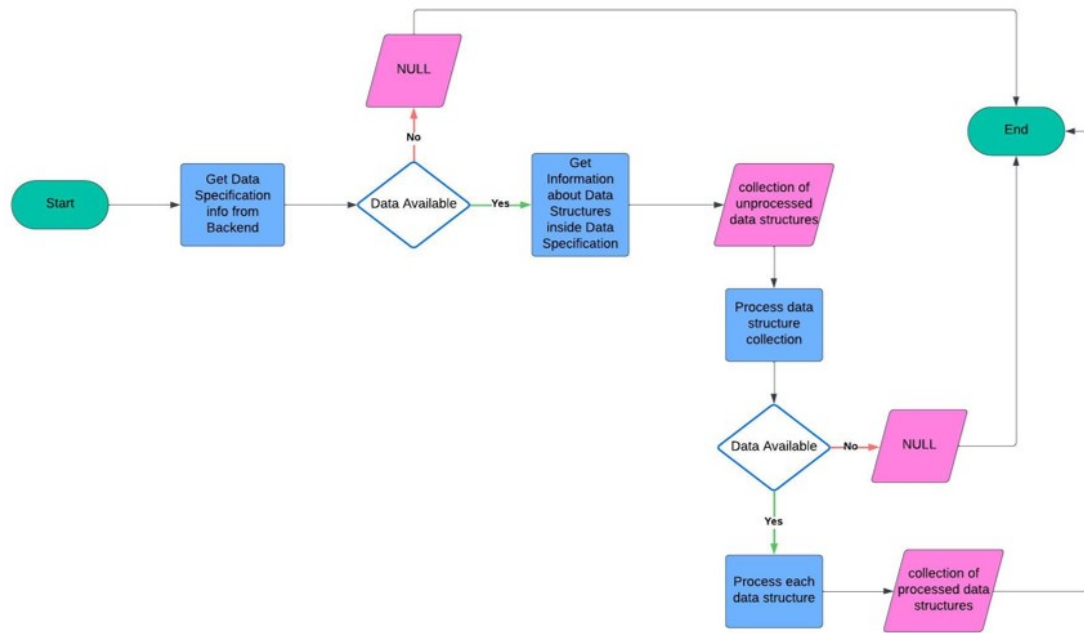


Figure 46 – Fetcher: High-Level Flow Diagram (Source: Author)

Complex, multi-level structure of data structure schemas in Dataspecer make their processing challenging. The information stored on the Dataspecer backend regarding individual data structure holds various properties. More precisely, the name as well as details regarding its attributes and associations are stored. Given their complexity and multi-level structure, the associations may also be treated as data structures themselves. This means that when it comes converting the collection of unprocessed data structures into a collection of processed data structures, the associations are processed recursively in order to consider whole representation of the root data structure. As illustrated by Figure 47, association fields are processed recursively until they are reduced to simple structures – attributes. Once primitivity is reached, the processing phase is complete. An example of processed data structure is illustrated by Figure 24 in 5.2.

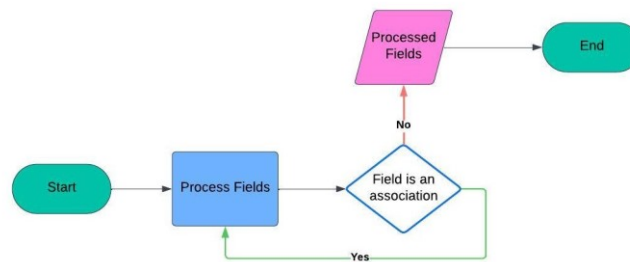


Figure 47 – Fetcher: High-Level Flow Diagram – Processing Fields (Source: Author)

As shown by Figure 45 in 5.5, **Frontend** (*MainForm.tsx*) represents the focal point of the system. It communicates with all other components. It not only receives data from the user, but also from the *Dataspecer Backend* and the *Fetcher* components. The user directly interacts with the frontend by providing metadata as well as operation details in order to generate output OAS. However, in order to do so, data structure needs to be chosen for which the operations will be defined. Because of this, the frontend receives collection of processed data structures from the fetcher component. This allows the user to select their target data structure and define corresponding operations. Because of the fact, that the program also supports maintenance of the API specifications, the frontend component receives data from the *Dataspecer Backend* (if available). This data is pre-saved configuration – operation details and metadata entered and saved by the user, whilst generating the OpenAPI specification previously. What happens is that upon submission of the form details, the configuration – metadata and operation details, are saved on the *Dataspecer Backend* for future reference. Moreover, the output OpenAPI specification is generated and displayed on the right side of the page as illustrated on Figure 28 in subchapter 5.3. What's more, the frontend component utilizes **validation schema** (*FormValidationSchema.tsx*) in order to ensure the validity of the output OAS and appeal to the characteristics of a good (REST) API. The most important restrictions defined in the validation schema are:

- Operation names must be unique.
- The combination of operation type (HTTP method) and path must be unique.

To be more precise, operation name is translated as `operationId` in the resulting OpenAPI specification. Since it represents an identifier of the operation inside the path construct, duplicates are unwanted and OpenAPI standard does not allow operations

with the same identifier (operation name). The second restriction states that each path has to contain operations with different operation types (HTTP methods). This means that, one path construct cannot contain two operations having DELETE (for example) as operation type. If these constraints are not met during the submission, the form is not submitted and corresponding error messages are displayed as illustrated by Figures 48 and 49. These restrictions ensure that the resulting OAS is concise and free from redundant operations.

Error: Each combination of endpoint and operation type must be unique within each data structure.

Figure 48 – Error message: combination of path and HTTP method has to be unique
(Source: Author)

Error: Operation name must be unique.

Figure 49 – Error Message: operation name must be unique (Source: Author)

As for the **OAS generator** (located in *OApiGenerator.tsx*), it receives all essential information for the production of OpenAPI specification from the frontend component. To be more precise, two types of inputs are received – user-provided configuration as well as data structure information. User-provided configuration holds the metadata, base URL as well as operation details which were provided by the user via the user interface. As for the data structure information, this is the collection of data structures fetched and processed by the *Fetcher* component. The OAS generator component is structured in multiple methods. The method ***generateOpenAPISpecification*** represents a central point invoking its helper methods – *handlePathOperations* and *createComponentSchema*. The method *handlePathOperations* serves the purpose of generating *paths* and their respective *operations* constructs whereas the method *createComponentSchema* aims to generate *components* construct containing schemas of the components. These helper methods call their respective sub-helper methods to handle smaller sub-construct generation. The high-level flow of the OAS generator is already considered by Figure 25 in 5.2, however, there are some notable aspects that need to be discussed in more detail. Figure 50 illustrates the process of component schema creation. Component schema is created for each data structure fetched by the fetcher component, which means that component schema creation method is called for each data structure in this collection.

The method firstly validates passed data structure. In case it is not null it proceeds with formatting its name according to the OpenAPI rules – any non-accepted character is converted to underscore. Next the method checks if a schema with such name already exists in order to prevent duplication. If not, a properties sub-construct is created based on the fields of the passed data structure. The last step of this method is setting the values for the fields (type, description, properties and required) of the schema object.

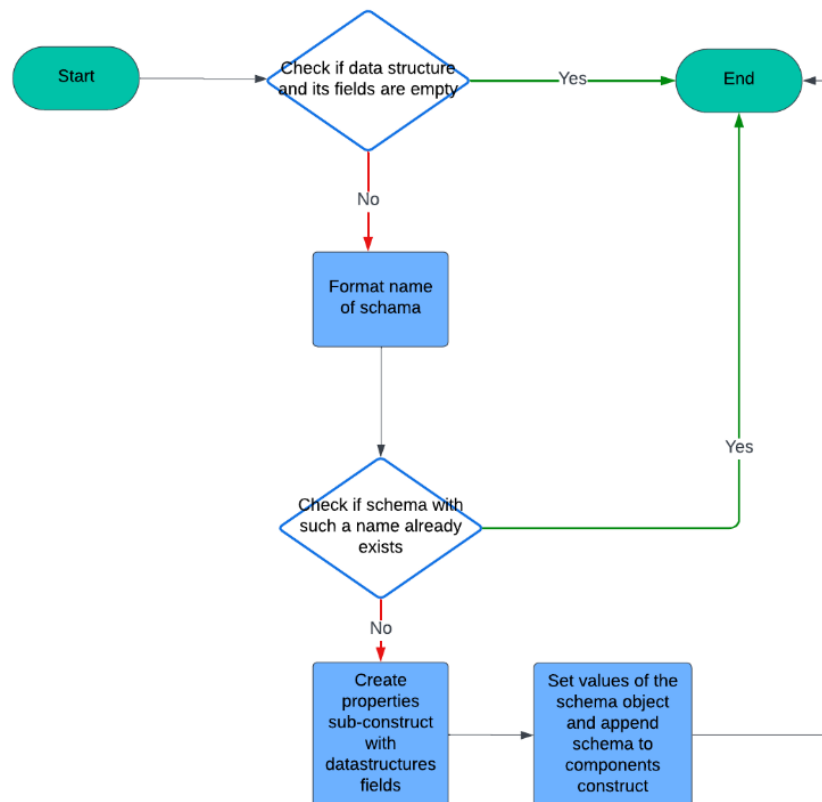


Figure 50 – Flow Diagram: Creating Schemas in Components Construct (Source: Author)

When it comes to populating the schema object there are two possibilities. If the passed data structure has properties (fields) configured in the Dataspecer, then the values of the schema are set according to these fields. If not, a schema with empty values is created with the name of the data structure and a description is set notifying the user that this component needs to be filled in. This is illustrated by Figure 51 below.

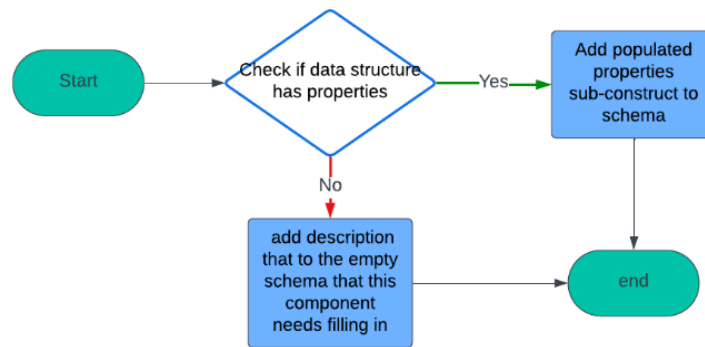


Figure 51 – Flow Diagram: Populating Schema Object (*Source: Author*)

What’s more, creation of properties sub-construct for a particular schema is also a challenge. The method responsible for property sub-construct creation (*createProperties*) is invoked by schema creation function. Two parameters are passed to property creation method – OAS and fields of the current data structure. The properties of the component schemas are managed in a nested structure. The program examines each property (*field*). In case of handling an association, the method *createComponentSchema* is invoked on the *field* which means that a component schema is generated for the nested data structures (associations) as well. To be more precise, when it comes to property creation, the function firstly initializes this sub-construct and then validates each field of the data structure. If the field is not null the method checks if the field is an attribute or association. In case it is an attribute, the program determines if this attribute represents a collection or not. If it represents a collection, the openAPI specification will mark this property as an array. Next, it checks whether the field should be marked as required. If so, it appends this field to the required sub-construct of the schema construct. The same process is performed in case of associations, however in this case firstly a schema is created for the association itself too. This is because an association, like the root data structure may have a multi-level structure and therefore may be considered as a data structure too. Lastly, the program checks if the properties construct of a data structure schema has a property named *id*. If not, the program appends *id* to the properties so that the resource is

identifiable. The process described above is illustrated by Figure 52 below.

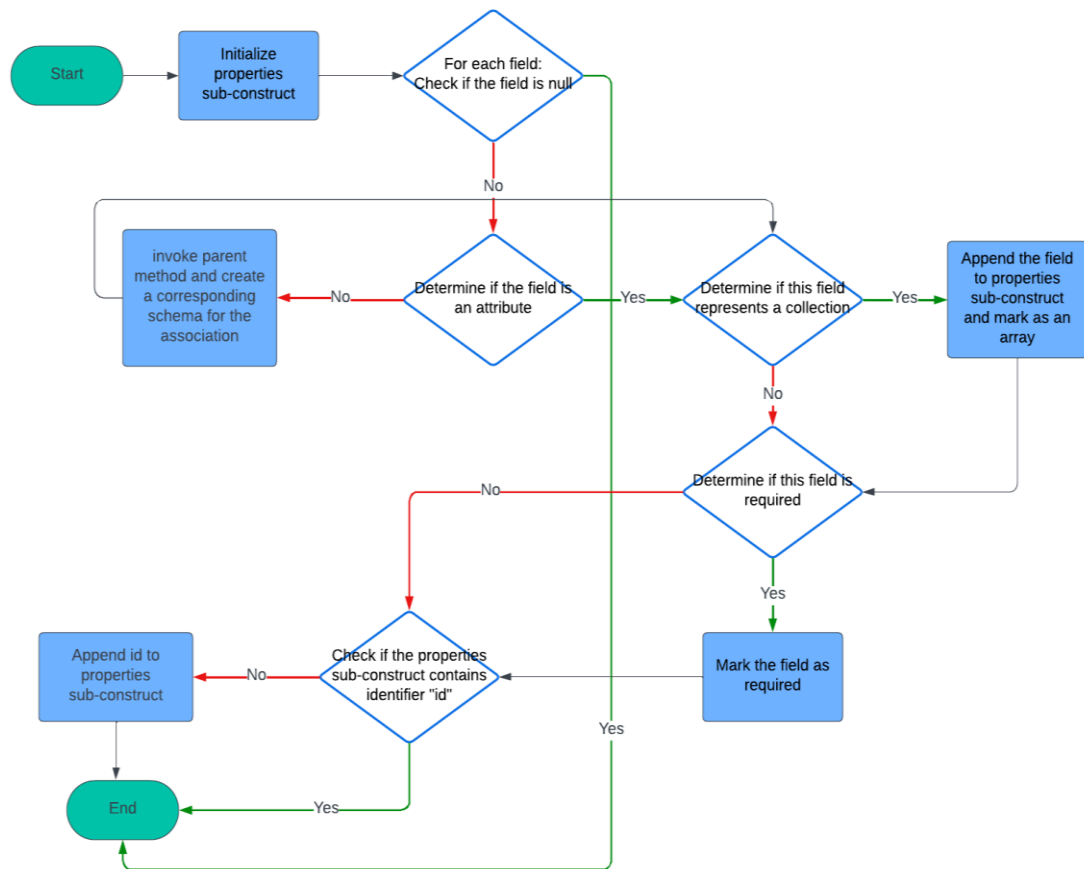


Figure 52 – Flow Diagram: Schema Properties Construction (Source: Author)

Another important challenge is to generate paths and their respective operations constructs. The method for handling paths and operations constructs is invoked for each operation of each data structure from the user-provided input. It receives the fetched data structures collection, data structure of the current iteration, the operation of the current iteration as well as OAS (initialized at the beginning of the OAS generation process) as parameters. Initially, the method checks if such path already exists and if not, a path construct is initialized. This path construct represents parent construct for its corresponding operation sub-constructs. Next the method proceeds with extraction of path parameters. Then it creates operation construct. In case of GET operation, query parameters are generated automatically based on the fields of the target data structure. Lastly created operation object is added to its parent path construct. Figure 53 below illustrates this process.

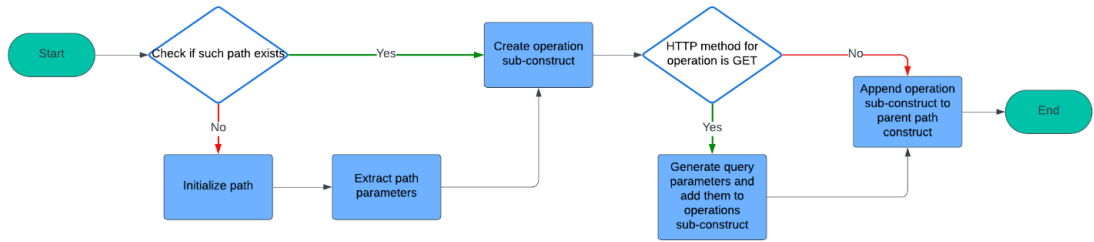


Figure 53 – Flow Diagram: Creating Path and Operation(s) constructs (Source: Author)

Creation of individual operation sub-constructs is another notable aspect to consider. The sub-helper method dedicated for *operation* sub-construct creation (*createOperationObject*) is invoked by the function (*handlePathOperations*) described above. As illustrated by Figure 54, it sets operation sub-construct based on the user-provided input provided via *OperationCard* component discussed in 5.2.1. The creation of response body is handled via a helper method and is based on the status code provided via *OperationCard* component from the UI. The generation of request body sub-construct is handled in this method as well. In case of POST or PATCH methods, the request body is generated based on the passed fields via the UI. Because of the fact that PUT response represents full update of the response, in this case the request body references corresponding data structure schema from the *components* construct.

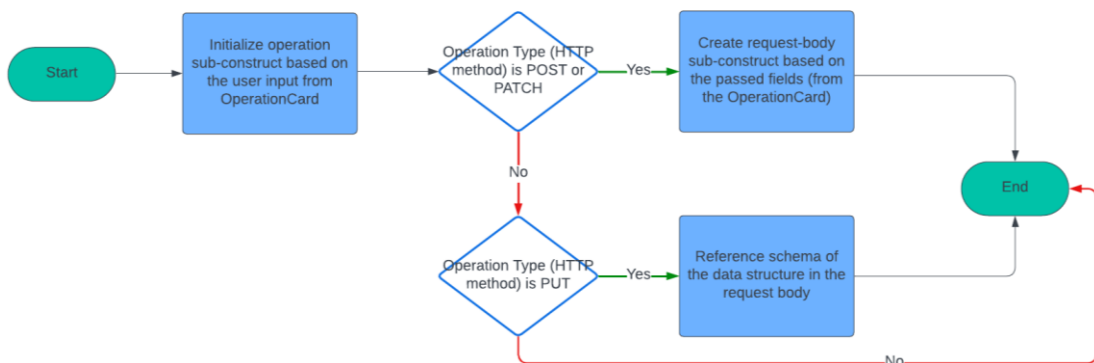


Figure 54 – Flow Diagram: Operation Sub-Construct Construction (Source: Author)

In conclusion, the generation of OpenAPI specification relies on the collaboration between aforementioned components. *Dataspecer Backend* stores pre-saved configuration as well as information about the data structures defined within the data specification. The *Fetcher* retrieves these data structures and processes them. The

Frontend receives necessary information from the *Dataspecer Backend, Fetcher* as well as the user and passes it to the OAS generator. The generator consolidates this information and provides resulting OpenAPI specification.

6. Evaluation

The definition of a good REST API has been provided in 3.4. A good REST API allows the developers to have a robust knowledge of the API. What's more, a good REST API is characterized as usable, structured, stateless and documented. Furthermore, it provides enhanced developer user experience. The exact explanations of these characterizations may be found in Table 3 of subchapter 3.4. The aim of this subchapter is to **determine whether the extension facilitates generation of API specifications for good REST APIs.**

Robust API Knowledge

The **extension allows the user/developer to have robust API knowledge of current API** via generated OpenAPI specification. In the scope of this project two aspects of the robust API knowledge are considered – domain concepts and execution facts. As mentioned in sub-chapter 3.1, the domain concepts represent abstract ideas that the API tries to model along with the corresponding terminology [23]. As for the execution facts, they focus on the expectations of the API, which means that this concept focuses on aspects like types (classes), as well as possible inputs and outputs [23]. Running example *Tourist destination* is modeled via meaningful attributes and associations. What's more these fields are named properly. To be more precise, generally a tourist destination may have an owner, it may also have a contact information and a capacity. Contact information may in turn consist of email, phone number and other relevant attributes. This means that domain concepts and part of the execution facts is supported by the Dataspecer tool itself, because it gives the user ability to design data structures based on conceptual models. While the user is designing a data structure, they are somewhat encouraged in designing meaningful data structures. For example, if the user is designing a *chair* data structure, choosing *material* as an attribute will be an easy choice because it will be provided in the possible options in the Dataspecer tool. What's more, the process of designing data structure schemas in the Dataspecer tool allows the user to set the types of the fields as well as their cardinalities. Understanding what are the types (classes) of the attributes and associations belongs to the execution facts aspect. The extension generating corresponding OpenAPI specification strengthens robust API knowledge. Firstly, the program appends field *id* which serves the purpose of resource identification. Next, the program allows the user to specify the operation details which is part of the execution facts. More precisely, the essential parts

needed for successful request creation are listed on the *OperationCard* component. And this information needs to be provided by the user. This means that in case of meaningful user input, the developer will have understanding through generated OAS (as seen in 5.4) of following:

- For which data structure is operation intended (root data structure or successor association)
- If a collection of resources is manipulated or not
- What is the name of the request/operation
- What parameters (if any) need to be passed in the request body
- What kind of response is expected and what is its media-type
- If and how (with what query parameters) can the result be filtered
- What is the description of the request/operation
- What is the endpoint exposed by the API for sending the request

Usability

Another key characteristic of a good REST API is usability. The **extension facilitates creation of a usable REST API specification in OpenAPI format**. While considering the implementation of the project (subchapter 5.6), it was mentioned that the program restricts the user of creating operations with the same name for a given data structure. For instance, in the context of the running example, this would mean that, there cannot exist two operations named “*createTouristDest*”. What’s more, the program restricts same HTTP methods for a particular path. For example, the user cannot define two delete requests (deleting tourist destination) for the same path. This ensures that the generated OpenAPI specification is as clean as possible from the redundant requests which in turn increases the likelihood that the output API specification and therefore its corresponding API is easy to use and hard to misuse.

Another important aspect connected to the usability is naming conventions. Because of the fact that the user is responsible for providing values of the operation details, it is users job to follow the rules of naming conventions and choose meaningful names for various fields. This means that, if the user disobeys the rules of naming conventions on purpose and provides unmeaningful data, the output will not be elegant. Despite this fact, the program tries to facilitate this aspect and provides suggested paths to the user. The suggested path specifies suggested structure for the endpoint. Each suggested

path structure is based on the target data structure – either root or its association one level below, which means that each suggested path (structure) is based on their names. In the context of the running example, some of the sample suggested paths could be: “/Touristdestinations”, “Touristdestinations/{id}/owners”. The user is able to set a different endpoint, however it is possible and, in most cases, recommended to use suggested paths directly as the operation endpoints. Because of the fact that the program appeals to a broad target audience with different experience, the user is provided with the description of possible HTTP methods and response codes as illustrated by Figure 55.

Operation ID: 4 Delete Operation

Association Mode:

Manipulate a collection:

Operation Name:

Operation Type: DELETE - Remove the specific entity ▼

Suggested Path:

Endpoint:

Comment:

Response code: 204 - success with no content to return ▼

- Select Operation Type
- GET - Retrieve the specific entity
- PUT - Full replacement of the entity
- PATCH - Partially update existing entity
- DELETE - Remove the specific entity

Figure 55 – OperationCard: Demonstration of HTTP Method Descriptions (Source: Author)

Comprehending the general meaning of the HTTP method (operation type) as well as expected response code gives the user opportunity to understand the essence of the request/operation. This increases the likelihood that meaningful names will be utilized for operation name and operation type fields. Furthermore, having information about the meaning of each possible HTTP method increases the likelihood that the user chooses appropriate option. For example, it is easy to make a mistake whilst creating a request that aims to update a resource. Considering a scenario when the user wants to update only a particular property of tourist destination. If there were no descriptions provided, a user with a limited experience would not know which HTTP method to use – PUT or PATCH. Since each option is appended with a description, it is easier to make a correct choice.

Last but not least, requests and responses are managed effectively. The extension allows the user to specify the request body only in the cases where it is needed – POST and PATCH requests. This means that resulting OpenAPI specification is free of redundant request-body sub-constructs ensuring clarity and accuracy of structure. As for the responses, in case of success (response codes 200 and 201) either a homogeneous collection of resources is returned or a single instance, depending on the operation nature. This conforms to the REST principles since REST requires resources to have a uniform representation in the response [13]. Figure 56 exemplifies single instance resource whereas Figure 57 exemplifies collection response.

```
"responses": {
  "201": {
    "description": "201 - new resource created",
    "content": {
      "application/json": {
        "schema": {
          "$ref": "#/components/schemas/tourist_destination"
        }
      }
    }
  }
}
```



Figure 56 – Sample Response: Single Instance (Source: Author)

```
"responses": {
  "200": {
    "description": "200 - successful request",
    "content": {
      "application/json": {
        "schema": {
          "type": "array",
          "items": {
            "$ref": "#/components/schemas/tourist_destination"
          }
        }
      }
    }
  }
}
```



Figure 57 – Sample Response: Collection (Source: Author)

Structured

Because of the fact that OpenAPI was chosen as the format of generated API specification, the output OAS is structured into the constructs that are specific to the OpenAPI standard. These constructs are: *openapi* and *info* (forming the metadata), *servers*, *paths*, *components* (input and output models) and *security* (authentication). This means that the decision of choosing OpenAPI standard as the output format ensures the resulting API specification is structured. However, OpenAPI specifications can be structured poorly as well. If the components construct was not modularized and the schemas of sub-components were placed directly in the parent schema, it would

violate the principle of being well-structured. This is not the case when it comes to the output of this extension. More particularly, when it comes to the *schemas* inside *components*, the parent schema references its child schemas fostering modularity and clarity. Figure 58 below illustrates example of bad structure and Figure 59 exemplifies a good, modular structure. Positive example was generated via this extension of Dataspecer. As evident, in the negative example (Figure 58) modularity is being neglected and everything is placed in one schema. This screenshot is only a fraction, since this way the schema gets too long and not manageable. On the other hand, the schema generated via the Dataspecer extension is clear and concise since it references other schemas. Referenced schemas are defined separately.

```

tourist_destination:
  type: object
  properties:
    capacity:
      type: integer
    contact:
      type: object
      properties:
        e-mail:
          type: array
          items:
            type: string
        url:
          type: string
        phone_number:
          type: number
          format: double
        has_contact:
          type: array
    }
}

tourist_destination:
  type: object
  properties:
    capacity:
      type: integer
    owner:
      type: array
      items:
        $ref: '#/components/schemas/human_or_person'
    contact:
      $ref: '#/components/schemas/contact'
    id:
      type: string
  required:
    - capacity
    - contact
  
```

Figure 58 – Example of a bad, unstructured Schema (Source: Author)
 Figure 59 – Example of a good, structured schema (Source: Author)

As mentioned, prior, the program provides distinction between path and query parameters which as defined in 3.4 is another characteristic of a good REST API. This enhances the structured quality of the specification, since the developer is able to distinguish path and query parameters clearly. Figure 60 illustrates query parameter whereas Figure 61 showcases path parameter.

```

"parameters": [
  {
    "name": "capacity",
    "in": "query",
    "description": "Filter results based on capacity",
    "schema": {
      "type": "integer"
    }
  }
]
  
```

Figure 60 – Query Parameter in OAS (Source: Author)

```
"parameters": [  
  {  
    "name": "id",  
    "in": "path",  
    "required": true,  
    "schema": {  
      "type": "string"  
    }  
  }  
]
```

Figure 61 – Path Parameter in OAS (Source: Author)

Documented

Another characteristic of a good API which needs to be reflected in the API specification is being (well) documented. First and foremost, it has to be noted that API specification by its essence already somewhat represents a form of documentation for the actual API implementation. As noted in subchapter 2.3, API specification is a formal document which holds the information regarding the elements that the API has to contain [4]. However, API specification and in this particular case OpenAPI specification needs to be documented too. Having a documented API specification supports the teams working on the API by allowing them to understand the essence of the API as well as work with it in an effective manner. Developed extension of the Dataspecer tool facilitates specification documentation by allowing the user to provide descriptions for various constructs. First, the user is prompted to provide API description in the metadata section. Moreover, the extension gives the user possibility to document each operation by providing a meaningful description. This again is the responsibility of the user. Furthermore, empty schemas are automatically appended with a description that this component needs to be filled in, which draws the user's/developer's attention to it. The examples of these descriptions are illustrated by Figures 62, 63 and 64. What's more, automatically generated (filtering) query parameters also include descriptions as illustrated by Figure 60.

```
"openapi": "3.0.0",  
"info": {  
  "title": "TouristDestinationsAPI",  
  "description": "API for tourist destination management",  
  "version": "1.0"  
},
```

Figure 62 – API Description (Source: Author)

```

"paths": {
  "/Touristdestinations": {
    "get": { ...
  },
  "post": {
    "summary": "creates tourist destinations",

```




Figure 63 – Operation (Request) Summary (Source: Author)

```

"schemas": {
  "human_or_person": {
    "type": "object",
    "properties": {},
    "description": "TODO: Fill in the component"
  },

```



Figure 64 – Empty Schema Description (Source: Author)

Stateless

Being stateless is another important characteristic of a good REST API which needs to be reflected in the corresponding OpenAPI specification. This is achieved by imposing a dedicated component in the frontend – *OperationCard*. As mentioned, throughout this thesis, each API request in the UI has a corresponding *OperationCard* component that gathers essential information for interaction with the server. The program is able to recognize path parameters and extract them. It also supports request body sub-construct generation, which means that all the necessary parameters can be gathered. The content-type is automatically set to JSON and does not need user interference. As mentioned in the previous chapters, the user has to specify exact parameters by choosing desired fields in the request body and specifying the path of the operation is user's responsibility as well. Suggested paths may be utilized for path endpoints as mentioned before. As for the authentication, utilization of JWT Bearer tokens is chosen automatically by the program. This enhances the statelessness of the OpenAPI, because the token is a self-contained entity [35]. This means that since the token contains necessary information for request authentication, there is no need for the server to maintain user's state [35]. Due to the fact that necessary information can be gathered by the *OperationCard* component and content-type and authentication approach are chosen automatically, each request defined in the output is populated so that, the server does not need to rely on the stored context while processing this request.

Enhanced Developer Experience

The concept of enhanced developer experience encompasses ensuring that the developers have a positive interaction with the API. This is achieved through API's usability. As discussed above, this extension of the Dataspecer tool facilitates generation of usable APIs. This means that when OpenAPI specifications are easy to use, developers are more likely to have a positive experience whilst interacting with the API. Another important factor facilitating enhanced developer experience is exposure of clear API paths. In the context of this extension this is achieved by defining a base path and additional paths with corresponding operations. Each path acts as an entry point. For example, the path *"/Touristdestinations"* represents an entry point for the collection of tourist destinations, while the path *"/Touristdestinations/{id}"* allows accessing particular instance of tourist destination. As for the HTTP methods (GET, POST, etc.) they provide ways for clients to create, read, update and delete (CRUD) API resources. In conclusion, generated OpenAPI specification offers enhanced developer experience by its usability as well as by the utilization of clearly exposed relative paths. Despite the fact that user is responsible for providing paths, this job is simplified by suggested path structures which in most cases can be directly utilized as the paths (endpoints) for the operation.

The electronic attachment (structure described in A.3) of this thesis contains two sample OpenAPI specifications which were generated via this extension of the Dataspecer tool. The first OpenAPI specification originates from a data specification which contains only one data structure – *Tourist destination* (running example). The title of the OpenAPI specification is *TouristDestinationsAPI*. As mentioned, it is possible from this extension to copy the output OpenAPI specification and open it in the swagger editor. Figure 65 illustrates this process and shows exposed paths and supported operations by this API. As evident, no errors are displayed by the Swagger editor when opening *TouristDestinationsAPI* OpenAPI Specification.

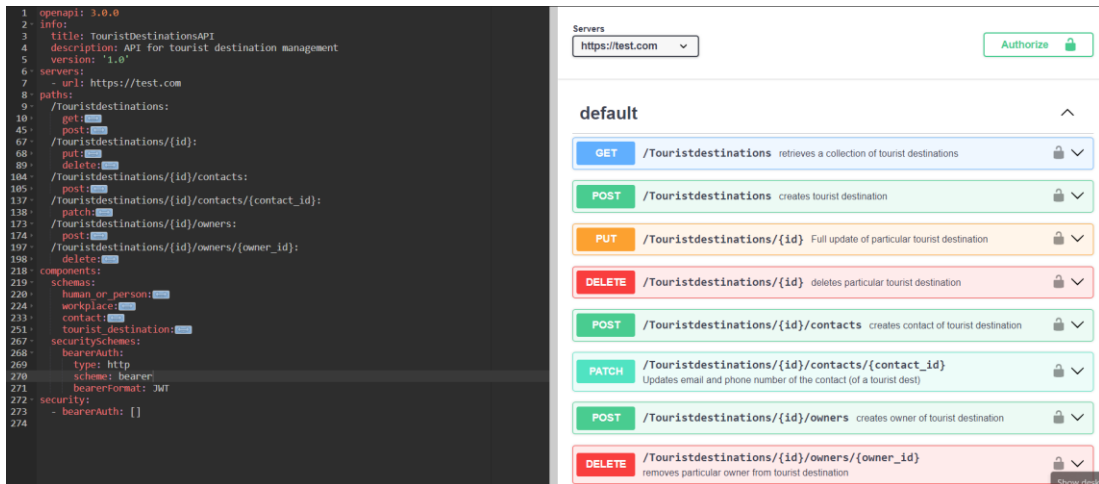


Figure 65 – TouristDestinationsAPI in Swagger Editor (Source: Author)

On the other hand, the second OpenAPI specification originates from a data specification containing two data structures – *album* and *concert*. Data structures album and concert are illustrated by Figures 66 and 67 below.

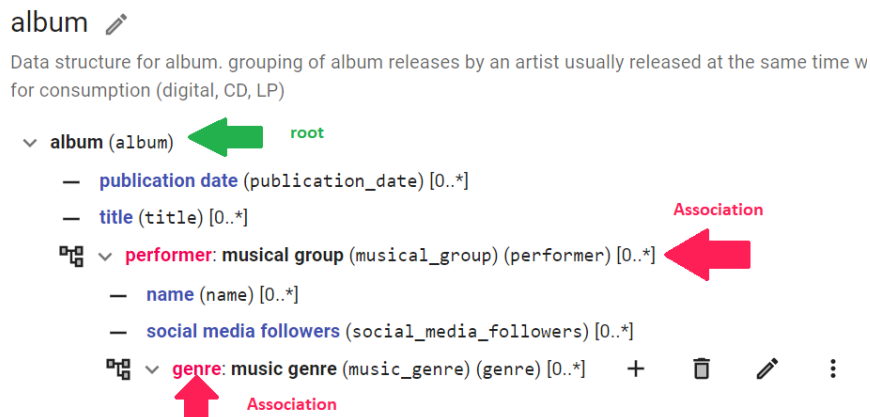


Figure 66 – Data Structure: album (Source: Adapted from [32])

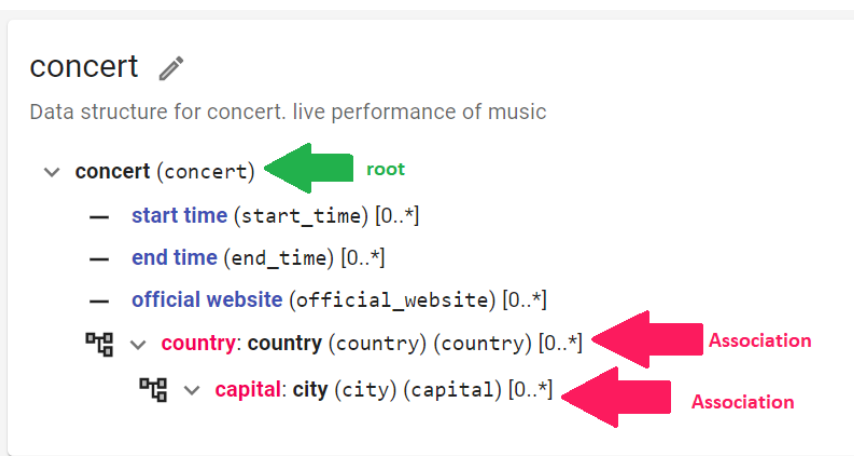


Figure 67 – Data Structure: concert (Source: Adapted from [32])

As evident none of the input data structures are flat, on the contrary they also include associations. Both *album* and *concert* contain associations one level below the root – *performer* and *country* which means that operations may be defined for them as well. An OpenAPI specification named *MusicManagementAPI* was generated for this data specification utilizing these data structures as well. The specification includes CRUD operations for the root data structure as well as associations one level below. The exposed paths and their respective operations are illustrated by Figure 68. It also showcases that there are no errors present when validating this OpenAPI specification with Swagger editor.

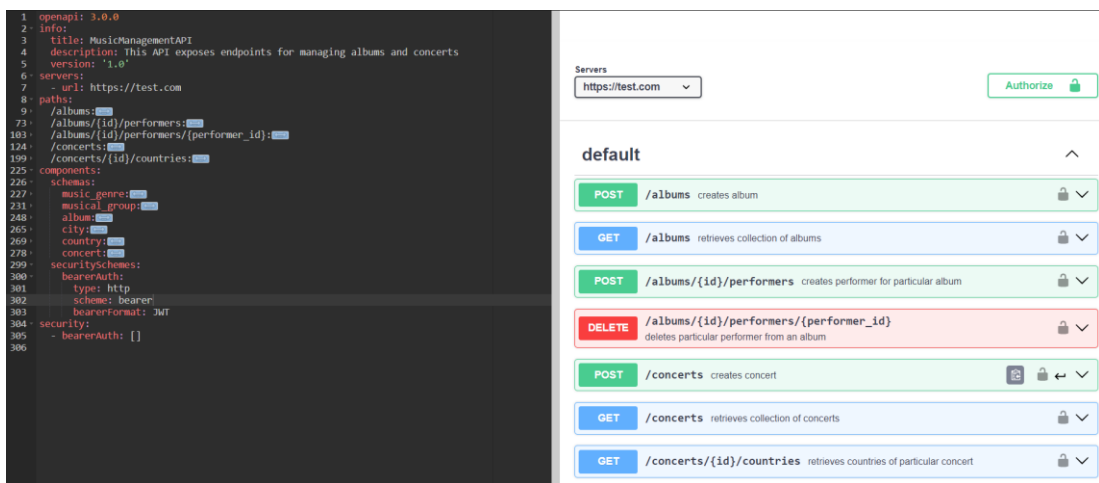


Figure 68 – MusicManagementAPI in Swagger Editor (Source: Author)

Conclusion

To sum up, while the original version of the Dataspecer tool did not have support for OpenAPI specification generation, this extension expands its capabilities and allows the users to generate OpenAPI specifications for the Dataspecer data structures. Resulting OpenAPI specifications are tailored to users' particular needs. In order to achieve this goal, firstly the Dataspecer tool was explored and its data structures were examined. Work carried out for this thesis includes the state of the art of relevant concepts such as – APIs, REST APIs, API specifications, as well as OpenAPI specifications. What's more, this thesis defines characteristics of a good REST API with respect to the API specifications. The solution focuses on these characteristics along with necessary user-provided input in order to provide resulting OpenAPI specification. In particular, the construction of the output OAS depends on the input data structure(s) and user-provided information representing operation details. What's more the tool supports API maintenance, which means that the user-provided configuration is saved and it is possible to update it at a later time. Since the field of technology including OpenAPI standard is ever-evolving, further research into this subject is warranted in order to maintain compatibility with the current trends. While the future work related to this topic would entail additional advancements such as determining if and how it is possible to change output OAS manually and reflect these changes back in the Dataspecer environment, this paper should be utilized as a testament to the potential of OpenAPI specifications and as a foundation of this separate future work.

Bibliography

- [1] Contentful, "What is an API? How APIs work, simply explained," 23 July 2023. [Online]. Available: <https://www.contentful.com/api/#what-is-an-ap>. [Accessed 2024].
- [2] O. Hämäläinen, "API-First Design with Modern Tools (Bachelor's Thesis)," May 2019. [Online]. Available: https://www.theseus.fi/bitstream/handle/10024/226493/Hamalainen_Oona.pdf?sequence=2&isAllowed=y. [Accessed 2024].
- [3] Dataspecer, "The problematics behind data modeling," [Online]. Available: <https://dataspecer.com/docs/tutorial/data-modeling-problematics/>. [Accessed 2024].
- [4] Davor, "API Documentation vs. Specification vs. Definition: What's the Difference?," 2024. [Online]. Available: <https://www.archbee.com/blog/api-documentation-specification-definition-difference>. [Accessed 2024].
- [5] Jeremy Whitlock et.al, "OpenAPI Specification v3.0.0," 2017. [Online]. Available: <https://spec.openapis.org/oas/v3.0.0>. [Accessed 2024].
- [6] Cisco and/or its affiliates, "The Internet of Everything," 2013. [Online]. Available: https://www.cisco.com/c/dam/en_us/about/business-insights/docs/ioe-value-at-stake-public-sector-analysis-faq.pdf. [Accessed 2024].
- [7] N. Kiesler and D. Schiffner, "What is a Good API? A Survey on the Use and," in *EAI IoECon 2023 - The Second EAI International Conference on the Internet of Everything*, Guimarães, Portugal, 2023.
- [8] F. Kilcommins, "https://nordicapis.com/," 14 July 2022. [Online]. Available: <https://nordicapis.com/the-benefits-of-using-api-specifications/>. [Accessed 2024].
- [9] A. Akhvlediani, "Expanding the Dataspecer Tool for API Creation and Management," 2024. [Online]. Available: <https://dataspecer.com/docs/projects/api/>. [Accessed 2024].
- [10] M. Goodwin, "What is an API (application programming interface)?," IBM, 09 April 2024. [Online]. Available: <https://www.ibm.com/topics/api>. [Accessed 2024].
- [11] TechTarget, "client-server," [Online]. Available: <https://www.techtarget.com/searchnetworking/definition/client-server>. [Accessed 2024].
- [12] "Client-Server Model," [Online]. Available: <https://www.geeksforgeeks.org/client-server-model/>. [Accessed 2024].
- [13] L. Gupta, "What is REST?," 12 December 2023. [Online]. Available: <https://restfulapi.net/>. [Accessed 2024].

- [14] M. Ekuan, C. Kittel, R. Downer, A. Buck, J. Bouska and M. Alberts, "RESTful web API design," 03 August 2023. [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design>. [Accessed 2024].
- [15] G. Jansen, "Thoughts on RESTful API Design - Resources," 2012. [Online]. Available: <https://restful-api-design.readthedocs.io/en/latest/resources.html>. [Accessed 2024].
- [16] R. Fadatare, "Identify Resources in RESTful API Design," [Online]. Available: <https://www.javaguides.net/2018/06/how-to-identify-rest-resources.html>. [Accessed 2024].
- [17] "Principles of Software Engineering," [Online]. Available: <https://www.d.umn.edu/~gshute/softeng/principles.html>. [Accessed 2024].
- [18] A. Helton, "Seriously, Write Your API Spec First," 27 September 2023. [Online]. Available: <https://www.readyssetcloud.io/blog/allen.helton/seriously-write-your-spec-first/>. [Accessed 2024].
- [19] Swagger, "OpenAPI Guide - Basic Structure," [Online]. Available: <https://swagger.io/docs/specification/basic-structure/>. [Accessed 2024].
- [20] N. M. E. G. P. Aikaterini Karavisileiou, "Ontology for OpenAPI REST Services," 2020. [Online]. Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9288198>.
- [21] K. Vasudevan, "The Benefits of OpenAPI-Driven API Development," 2018. [Online]. Available: <https://swagger.io/blog/api-strategy/benefits-of-openapi-api-development/>. [Accessed 2024].
- [22] M. Mendoza, "How Businesses Can Benefit from OpenAPI Specification," 2020. [Online]. Available: <https://www.altexsoft.com/blog/openapi-specification/>. [Accessed 2024].
- [23] K. Thayer, S. E. Chasin and A. J. Ko, "A Theory of Robust API Knowledge," 2021. [Online]. Available: <https://dl.acm.org/doi/10.1145/3444945>. [Accessed 2024].
- [24] Swagger, "Best Practices in API Design," [Online]. Available: <https://swagger.io/resources/articles/best-practices-in-api-design/#:~:text=In%20general%2C%20an%20effective%20API,who%20work%20with%20it%20constantly..> [Accessed 2024].
- [25] A. Redko, "The Hypermedia APIs support in JAX-RS and OpenAPI: a long way to go," 2019. [Online]. Available: <https://redko1.rssing.com/channel-8348961/article64.html?nocache=0>. [Accessed 2024].
- [26] A. Redko, "RESTful services with HATEOAS. Documenting Hypermedia APIs," 15 May 2020. [Online]. Available: <https://www.javacodegeeks.com/restful-services-with-hateoas-documenting-hypermedia-apis.html#hyperschema>. [Accessed 2024].

- [27] "7 REST API Best Practices for Designing Robust APIs," [Online]. Available: <https://www.getambassador.io/blog/7-rest-api-design-best-practices>. [Accessed 2024].
- [28] S. Rivera, "3 examples of conceptual data models for data and analytics," 2023. [Online]. Available: <https://www.thoughtspot.com/data-trends/data-modeling/conceptual-data-model-examples>. [Accessed 2024].
- [29] P. Asman, "Attributes and Associations in Object Modeling," 1999. [Online]. Available: <https://www.hillside.net/plop/plop99/proceedings/asman/Attributes.pdf>. [Accessed 2024].
- [30] IBM, "Attributes versus Aggregate and Composite Associations in Rose RealTime," 2018. [Online]. Available: <https://www.ibm.com/support/pages/attributes-versus-aggregate-and-composite-associations-rose-realtime>. [Accessed 2024].
- [31] I. J. G. B. James Rumbaugh, "The Unified Modeling Language Reference Manual," 1999. [Online]. Available: https://idsi.md/files/file/referinte_utili_studenti/The%20Unified%20Modeling%20Language%20Reference%20Manual.pdf. [Accessed 2024].
- [32] Dataspecer, "Dataspecer specification manager," [Online]. Available: <https://tool.dataspecer.com/>. [Accessed 2024].
- [33] w3schools, "HTTP Status Messages," [Online]. Available: https://www.w3schools.com/tags/ref_httpmessages.asp. [Accessed 2024].
- [34] Swagger, "Bearer Authentication," [Online]. Available: <https://swagger.io/docs/specification/authentication/bearer-authentication/>. [Accessed 2024].
- [35] Secret Double Octopus, "Stateless Authentication," [Online]. Available: <https://doubleoctopus.com/security-wiki/network-architecture/stateless-authentication/>. [Accessed 2024].
- [36] Dataspecer, "Dataspecer packages," [Online]. Available: <https://tool.dataspecer.com/manager/>. [Accessed 2024].
- [37] Š. Stenclák et.al, "Dataspecer", GitHub. [Online] Available: <https://github.com/mff-uk/dataspecer/>. [Accessed 2024].

List of Figures

Figure 1 – Research Workflow (<i>Source: Author</i>)	4
Figure 2 – Client-Server Communication (<i>Source: Author</i>).....	6
Figure 3 – Sample OpenAPI Specification for Tourist Destinations Management API (<i>Source: Author</i>).....	11
Figure 4 – Example of API Usage Patterns (<i>Adapted from [23]</i>).....	15
Figure 5 – Transactional Conceptual Data Model (Entities and Relationships only) (<i>Source: Adapted from [28]</i>).....	22
Figure 6 – Sample Data structures based on the Conceptual Model of Figure 5 (<i>Source: Author</i>)	22
Figure 7 – Dataspecer: Specification Manager (<i>Source: Adapted from [32]</i>)	24
Figure 8 – Dataspecer: Create Data Specification (<i>Source: Adapted from [32]</i>)	24
Figure 9 – Dataspecer: Data Structure Creation (<i>Source: Adapted from [32]</i>)	25
Figure 10 – Dataspecer: Vocabulary Sources Selection (<i>Source: Adapted from [32]</i>)	25
Figure 11 – Dataspecer: Data Structure Editor (<i>Source: Adapted from [32]</i>)	25
Figure 12 – Dataspecer: Setting Root Element in Structure Editor (<i>Source: Adapted from [32]</i>).....	26
Figure 13 – Dataspecer: Adding fields to the Root Data Structure (<i>Source: Adapted from [32]</i>).....	26
Figure 14 – Dataspecer: Choosing the fields of the Root Data Structure (<i>Source: Adapted from [32]</i>)	26
Figure 15 – Dataspecer: Tourist Destination, Resulting Data Structure (<i>Source: Adapted from [32]</i>)	27
Figure 16 – Running Example: Attributes (<i>Source: Adapted from [32]</i>)	28
Figure 17 – Running Example: Associations (<i>Source: Adapted from [32]</i>).....	28
Figure 18 – Resources in REST and Data Structures in Dataspecer (<i>Source: Author</i>)	32
Figure 19 – Flat Data Structure in Dataspecer (<i>Source: Author</i>).....	36
Figure 20 – Complex Data Structure (<i>Source: Author</i>)	37
Figure 21 – Structure of User-Provided Input (<i>Source: Author</i>)	40
Figure 22 – Sample User Input: Tourist destination (<i>Source: Author</i>).....	41
Figure 23 – DataStructure and Field Models (<i>Source: Author</i>).....	42

Figure 24 – Example of DataStructure Object - Tourist destination (<i>Source: Author</i>)	43
.....	
Figure 25 – High-Level Flow Diagram: OAS Generation (<i>Source: Author</i>).....	44
Figure 26 – Dataspecer Extension: Initial View (<i>Source: Author</i>)	45
Figure 27 – Dataspecer Extension: Select Data Structure and Start Defining Operations (<i>Source: Author</i>).....	45
Figure 28 – Dataspecer Extension: Output (<i>Source: Author</i>)	46
Figure 29 – Dataspecer Extension: OperationCard Component (<i>Source: Author</i>)....	46
Figure 30 – Retrieve Tourist Destinations: UI representation (<i>Source: Author</i>)	48
Figure 31 – Retrieve Tourist Destinations in OAS (<i>Source: Author</i>)	49
Figure 32 – Create Contact: UI Representation (<i>Source: Author</i>)	50
Figure 33 – Create Contact in OAS (<i>Source: Author</i>)	51
Figure 34 – Structure of Tourist Destination OAS (<i>Source: Author</i>)	53
Figure 35 – Mapping of Metadata between UI and OAS (<i>Source: Author</i>)	53
Figure 36 – GET Tourist destinations: Sample Operation (<i>Source: Author</i>).....	55
Figure 37 – POST Tourist destination: Sample Operation (<i>Source: Author</i>)	56
Figure 38 – PUT Tourist destination: Sample Operation (<i>Source: Author</i>)	57
Figure 39 – DELETE Tourist destination: Sample Operation (<i>Source: Author</i>).....	58
Figure 40 – POST contact: Sample Operation (<i>Source: Author</i>).....	59
Figure 41 – Security Schema in OAS (<i>Source: Author</i>)	60
Figure 42 – Tourist Destination Schema in Dataspecer and in OAS (<i>Source: Author</i>)	61
.....	
Figure 43 – Association Schemas in Dataspecer and in OAS (<i>Source: Author</i>)	61
Figure 44 – High-Level Architecture: Integration with Dataspecer (<i>Source: Author</i>)	65
.....	
Figure 45 – API-Specification Generator: High-Level Flow Diagram (<i>Source: Author</i>)	66
.....	
Figure 46 – Fetcher: High-Level Flow Diagram (<i>Source: Author</i>)	67
Figure 47 – Fetcher: High-Level Flow Diagram – Processing Fields (<i>Source: Author</i>)	68
.....	
Figure 48 – Error message: combination of path and HTTP method has to be unique (<i>Source: Author</i>).....	69
Figure 49 – Error Message: operation name must be unique (<i>Source: Author</i>)	69

Figure 50 – Flow Diagram: Creating Schemas in Components Construct (<i>Source: Author</i>)	70
Figure 51 – Flow Diagram: Populating Schema Object (<i>Source: Author</i>)	71
Figure 52 – Flow Diagram: Schema Properties Construction (<i>Source: Author</i>)	72
Figure 53 – Flow Diagram: Creating Path and Operation(s) constructs (<i>Source: Author</i>)	73
Figure 54 – Flow Diagram: Operation Sub-Construct Construction (<i>Source: Author</i>)	73
Figure 55 – OperationCard: Demonstration of HTTP Method Descriptions (<i>Source: Author</i>)	77
Figure 56 – Sample Response: Single Instance (<i>Source: Author</i>)	78
Figure 57 – Sample Response: Collection (<i>Source: Author</i>)	78
Figure 58 – Example of a bad, unstructured Schema (<i>Source: Author</i>)	79
Figure 59 – Example of a good, structured schema (<i>Source: Author</i>)	79
Figure 60 – Query Parameter in OAS (<i>Source: Author</i>)	79
Figure 61 – Path Parameter in OAS (<i>Source: Author</i>)	80
Figure 62 – API Description (<i>Source: Author</i>)	80
Figure 63 – Operation (Request) Summary (<i>Source: Author</i>)	81
Figure 64 – Empty Schema Description (<i>Source: Author</i>)	81
Figure 65 – TouristDestinationsAPI in Swagger Editor (<i>Source: Author</i>)	83
Figure 66 – Data Structure: album (<i>Source: Adapted from [32]</i>)	83
Figure 67 – Data Structure: concert (<i>Source: Adapted from [32]</i>)	83
Figure 68 – MusicManagementAPI in Swagger Editor (<i>Source: Author</i>)	84
Figure 69 – Dataspecer Manager: List of Data Specifications (<i>Source: Adapted from [36]</i>)	93
Figure 70 – Dataspecer Manager: Create new OpenAPI Specification (<i>Source: Adapted from [36]</i>)	93
Figure 71 – Dataspecer Manager: Accessing Extension (<i>Source: Adapted from [36]</i>)	94
Figure 72 – Necessary part of URL (<i>Source: Author</i>)	95
Figure 73 – Electronic Attachment Structure (<i>Source: Author</i>)	96

List of Tables

Table 1 – Characteristics of Good API (<i>Source: [7]</i>)	17
Table 2 – REST API: notable Best Practices (<i>Source: [27]</i>).....	18
Table 3 – Characteristics of a good REST API (with respect to API specification) (<i>Source: [13], [27]</i>)	20
Table 4 – Running Example: Tourist destinations - Field Details (<i>Source: Author</i>).	29
Table 5 – Resources in REST and Data Structures in Dataspecer (<i>Source: [15]</i>).....	31
Table 6 – Advantages and Disadvantages of User-Specified Details (<i>Source: Author</i>)	34
Table 7 –Advantages and Disadvantages of Automatic Generation (<i>Source: Author</i>)	34
Table 8 – OperationCard Sub-Components (<i>Source: Author</i>).....	47
Table 9 – Supported Constructs of the Output OAS (<i>Source: Adapted from [9]</i>).....	63

A Attachments

A.1 Accessing the Extension Application

The aim of this attachment is to provide information how to access the Dataspecer tool extension on production environment. As mentioned, Dataspecer is in the process of migration which means that the Dataspecer Manager will represent starting point for various features of the tool. This extension is accessible via Dataspecer Manager as well.

First the user needs to navigate to the manager via following URL: <https://tool.dataspecer.com/manager/>. Having accessed this link, the user will be able to view the data specifications created prior as illustrated by Figure 69.

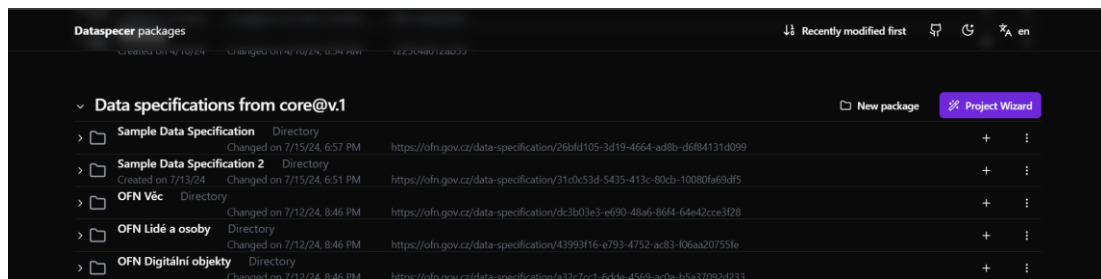


Figure 69 – Dataspecer Manager: List of Data Specifications (Source: Adapted from [36])

For the purpose of generating OAS the user needs to navigate to the desired data specification and click the plus button. Once this button is clicked, the option OpenAPI needs to be chosen as showcased by Figure 70.

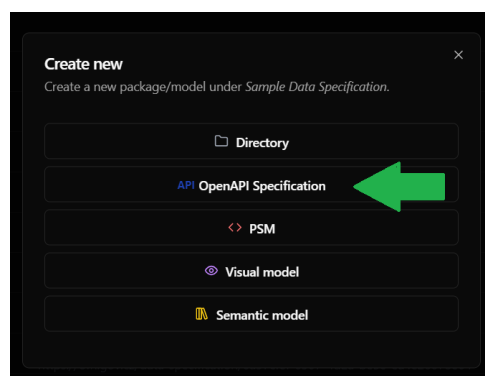


Figure 70 – Dataspecer Manager: Create new OpenAPI Specification (Source: Adapted from [36])

Once OpenAPI Specification is chosen, the user needs to provide name and description and save changes. Once the initialization is complete, the user will see the newly created package/model under desired data specification as illustrated below by Figure

71. Once the user clicks “open” pointed by the green arrow, the user is redirected to the extension application and can start designing their API.

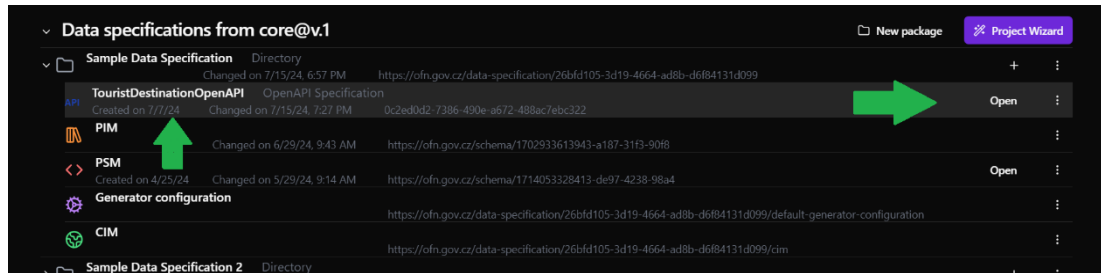


Figure 71 – Dataspecer Manager: Accessing Extension (Source: Adapted from [36])
Sample Data Specification and *Sample Data Specifcaiton 2* were utilized in order to generate electronically attached OpenAPI specifications – *MusicManagementOpenAPI.json* and *TouristDestinationOpenAPI.json*.

A.2 Build Instructions

Following steps [9] need to be taken in order to build the application:

1. Firstly, whole mono repository [37] needs to be cloned via *git clone ...*
2. Next, local environment needs to be set by firstly creating file *.env.local* in following directory: *dataspecer/applications/api-specification*. In this file a local environment variable called *VITE_BACKEND* has to be defined and its value needs to be set to the backend URL: <https://backend.dataspecer.com>.
3. Running *npm install* in the root of the repository represents the next step which is responsible for installing all necessary packages.
4. Next *npm run build* needs to be run in order to build the dependencies of this package.

It is important to note that, in case building only this application (the extension) is required, *npm run build* can be run from the current directory (*dataspecer/applications/api-specification*). Running the live server is also possible via *npm run dev* (from the current directory as well). However, in case the user wants to run this application locally, he/she has to manually update the URL in the browser. The part of URL can be copied from the production environment as illustrated by Figure 72 below.

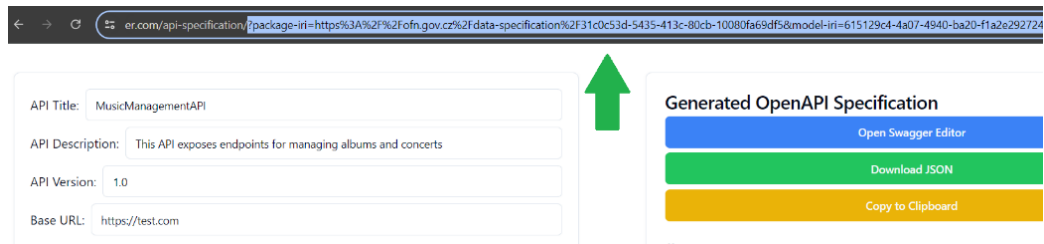


Figure 72 – Necessary part of URL (*Source: Author*)

In this case the user's URL would resemble: `http://localhost:PORT_NUM /?package-iri=https%3A%2F%2Fofn.gov.cz%2Fdata-specification%2F26bfd105-3d19-4664-ad8b-d6f84131d099&model-iri=0c2ed0d2-7386-490e-a672-488ac7ebc322`.

It is also important to note that, for the full experience on the local environment aside from the current application, other components of Dataspecer need to be run. These components are:

- Structure editor (directory: *dataspecer/applications/client*)
- Dataspecer Manager (directory: *dataspecer/applications/manager*)
- Backend (directory: *dataspecer/services/backend*)

A.3 Electronic Attachments

This electronic attachment contains the source code and OpenAPI specifications generated via this extension and is included as an archive. Figure 73 below illustrates the structure of the archive (only notable files are demonstrated).

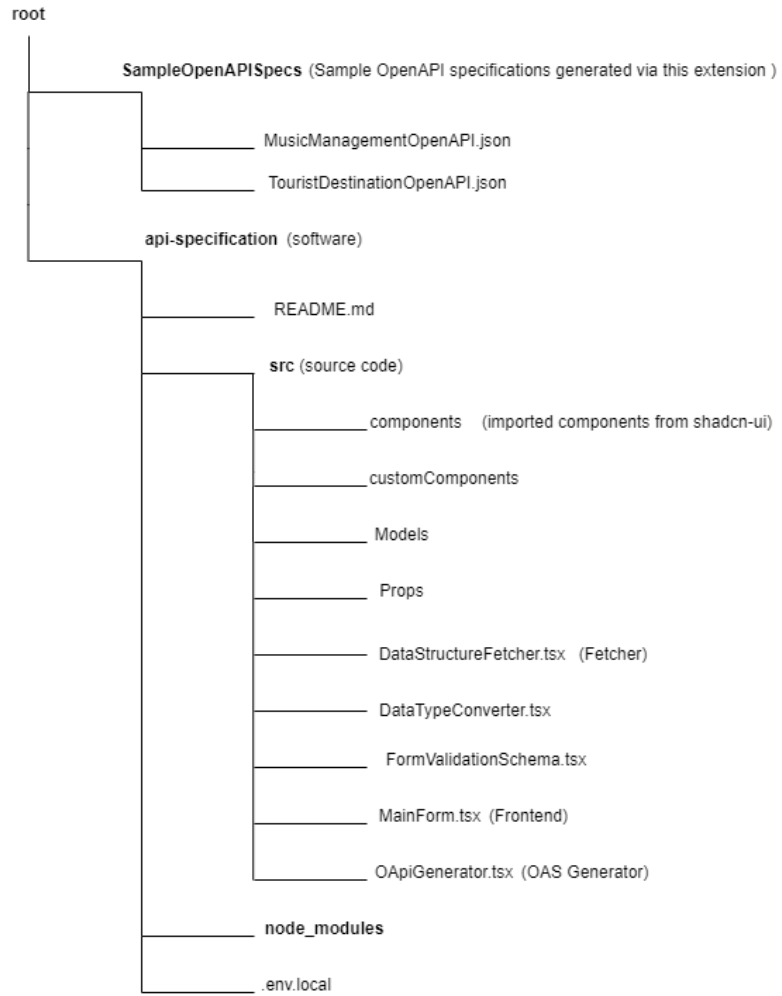


Figure 73 – Electronic Attachment Structure (Source: Author)