



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

JSON SCHEMA MAKER

APLIKACE PRO DEFINICI JSON SCHÉMAT

BACHELOR'S THESIS

BAKALÁŘSKÁ PRÁCE

AUTHOR

AUTOR PRÁCE

SUPERVISOR

VEDOUCÍ PRÁCE

MARTIN FUJAČEK

Ing. ALEŠ SMRČKA, Ph.D.

BRNO 2020

Bachelor's Thesis Specification



Student: **Fujaček Martin**
Programme: Information Technology
Title: **JSON Schema Maker**
Category: Web

Assignment:

1. Study the JSON format and the draft of JSON schema standard
2. Design a web application for specification of a JSON schema. Web application should include an editor of JSON documents enabling syntax highlight and interactive modification of a JSON code. The output of the application will be the definition of a JSON schema used for modified JSON documents. Optional feature of the application is the validator of JSON documents for different programming languages.
3. Implement the designed application as a single-page application.
4. Demonstrate the functionality of the application on an artificial set of use cases.

Recommended literature:

- Draft standardu IETF pro schéma JSON. Dostupné na URL: <https://tools.ietf.org/html/draft-handrews-json-schema-01>
- Standard JSON. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259.

Requirements for the first semester:

- The first two points.

Detailed formal requirements can be found at <https://www.fit.vut.cz/study/theses/>

Supervisor: **Smrčka Aleš, Ing., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: May 28, 2020
Approval date: May 15, 2020

Abstract

JSON Schema provides a way of controlling how JSON data should look like. The goal of this thesis is to simplify the definition of schemas for existing JSON data. This Bachelor's thesis discusses the design and implementation of a single-page application for generating JSON Schema based on sample JSON documents. The output of this application can help users with defining the skeleton of the schema. The contribution of this thesis resides in the ability to generate a schema for multiple JSON samples, without the need for repetitive usage of the application and subsequent merging of the individual schemas into the resulting schema. On top of that, the implemented tool provides an automatic validation while manipulating either the input or the schema and also providing additional information in case of errors.

Abstrakt

JSON Schema predstavuje spôsob určovania ako majú vyzerat dáta formátu JSON. Táto práca má za cieľ zjednodušiť definovanie schém pre existujúce dáta formátu JSON. Popisuje návrh a implementáciu jednostránkovej aplikácie pre generovanie JSON schém podľa vzoriek JSON dokumentov. Výstup aplikácie môže pomôcť užívateľom s tvorbou kostry schémy. Prínos tejto práce spočíva v možnosti generovania výslednej schémy nad viacerými vzorkami JSON dokumentov bez nutnosti opakovaného používania aplikácie a následného zlučovania jednotlivých schém do výslednej schémy. Okrem toho poskytuje automatickú validáciu pri manipulácii či už so vstupom, alebo schémou, pričom poskytuje dodatočné informácie o prípadných chybách.

Keywords

web, JSON, schema, generator, validation, single-page application

Klíčové slová

web, JSON, schema, generátor, validácia, jednostránková aplikácia

Reference

FUJAČEK, Martin. *JSON Schema Maker*. Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, Ph.D.

Rozšírený abstrakt

Táto práca sa zaoberá tvorbou a manipuláciou s JSON schémami, pričom si kladie za cieľ uľahčiť prácu ich autorom. JSON schéma poskytuje spôsob definovania štruktúry a obsahu dokumentov vo formáte JSON. Účelom práce bolo navrhnúť a implementovať jednoduchú, jednostránkovú, webovú aplikáciu, ktorej úlohou je vygenerovať kosť JSON schémy na základe existujúcich dokumentov formátu JSON, s možnosťou dodatočných, ručných úprav výsledku.

Dôležitou vlastnosťou výsledného riešenia je schopnosť generovať schému z väčšieho počtu vstupných dokumentov bez nutnosti opakovaného použitia pre každý dokument zvlášť. Takisto bol kladený dôraz na poskytovanie spätnej väzby užívateľovi v priebehu manipulácie so vstupnými dokumentmi alebo schémou. V neposlednom rade bolo dôležité zabezpečiť, aby bolo možné s výsledkom ďalej pracovať a dopĺňať ho či už ručne, alebo opakovaným použitím.

Úvod tejto práce je venovaný oboznámeniu s formátom JSON. Najmä však s návrhom štandardu JSON schémy, konkrétne verzie 07, na ktorú je cielený výsledný produkt. Detailne vysvetľuje jej princípy a popisuje jednotlivé kľúčové slová. Ďalej boli analyzované existujúce riešenia s podobným zameraním. Výsledkom tejto analýzy bolo vytipovanie chýbajúcich vlastností a výber použitej externej aplikácie pre realizovanie komponenty zodpovednej za validáciu vstupného dokumentu voči schéme. Nasledovalo stručné predstavenie použitých princípov a technológií, ktoré boli zvolené pre realizáciu riešenia.

Samotné riešenie predstavuje dva základné celky. Prvým z nich je *validátor*. Táto súčasť slúži na overenie, že daný vstupný dokument odpovedá schéme. Inými slovami rozhoduje, že má očakávanú štruktúru a obsah. Výsledkom validácie je okrem samotného verdiktu aj zoznam prípadných nezrovnalostí. Každý z týchto údajov nesie informáciu o tom, ktorá časť dokumentu nezodpovedá definíciám, so stručným slovným vysvetlením a lokalitou definície v schéme pre jednoduchú orientáciu. Samotná implementácia validátora nie je predmetom tejto práce, bola použitá voľne dostupná knižnica tretej strany.

Zaujímavejšou časťou je zložka, ktorej úlohou je tvorba kosť schémy. Iteratívnym spôsobom sú najskôr vygenerované čiastočné schémy pre každý vstupný dokument, ktoré sa následne kombinujú do výslednej schémy. Je dôležité dodať, že vždy sa pracuje s aktuálnym obsahom schémy. To znamená, že je nutné schému vyprázdniť pred ďalším použitím v prípade, že nie je požadované pokračovať v práci s danou schémou.

Obe služby sú realizované formou webového aplikačného programového rozhrania, ktoré využíva klientská časť aplikácie predstavujúca webové užívateľské rozhranie. Jeho najzákladnejšou časťou sú dva editory, ktoré slúžia pre prácu so vstupnými dokumentmi, respektíve schémou. Samotné editory poskytujú užívateľovi bohatú funkcionálnosť ako zvýraznenie, či kontrolu syntaxe. Okrem toho poskytujú možnosť definovať vlastné chybové oznámenie a viazať ich na konkrétne pozície v editore. Týmto spôsobom je realizovaná spätná väzba užívateľovi o chybových oznámeniach z validácie. Ďalšou dôležitou vlastnosťou je možnosť vytvoriť a používať viacero inštancií textových modelov v rámci jedného editoru. To umožňuje realizáciu záložiek a teda súbežne pracovať s viacerými vstupnými dokumentmi.

Pre implementáciu serverovej časti systému bola zvolená technológia .NET Core, ktorá poskytuje vývoj multiplatformných aplikácií, a rovnako aj pre jednoduchú integráciu s externou validačnou knižnicou, ktorá je postavená nad rovnakou technológiou. Klientská časť je realizovaná v jazyku JavaScript pre jeho dominantné postavenie vo webových prehliadačoch.

Správnosť funkcionality aplikácie bola overená formou automatizovaných testov. Ešte pred samotným vývojom serverovej časti bola vytvorená sada jednotkových testov, ktoré definovali očakávaný výstup. To malo za následok rýchlejšiu spätnú väzbu a následne rýchlejší vývoj. Po dokončení minimálnej nutnej funkcionality celej aplikácie boli vytvorené systémové testy. Časti, ktoré nebolo možné otestovať automaticky boli overené ručným testovaním.

JSON Schema Maker

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Aleš Smrčka, Ph.D. The supplementary information was provided by Ing. Jiří Pokorný. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Martin Fujaček
May 21, 2020

Acknowledgements

I would like to express my gratitude towards the supervisor, Ing. Aleš Smrčka, Ph.D. for his bits of advice and professional guiding during the creation of this thesis. I also appreciate countless remarks and tips from Ing. Jiří Pokorný.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 5 |
| 2 | JSON Schema and Used Technologies | 6 |
| 2.1 | Introduction to JSON Schema | 6 |
| 2.1.1 | JSON Document and its Structure | 6 |
| 2.1.2 | JSON Schema | 7 |
| 2.2 | Existing JSON Schema Generators | 12 |
| 2.3 | Existing JSON Schema Validators | 17 |
| 2.4 | Used Technologies and Principles | 20 |
| 2.4.1 | Hyper Text Markup Language & Cascading Style Sheets | 20 |
| 2.4.2 | JavaScript | 20 |
| 2.4.3 | C# and .NET Core | 21 |
| 2.4.4 | Client/Server Model | 21 |
| 2.4.5 | Hypertext Transfer Protocol | 22 |
| 2.4.6 | Web Application Programming Interface | 23 |
| 3 | Design of JSON Schema Maker | 24 |
| 3.1 | Outlining the Final Product | 24 |
| 3.2 | JSON Schema Maker Requirements Specification | 24 |
| 3.3 | Designing Graphical User Interface | 25 |
| 3.4 | Architectural Overview of the Application | 26 |
| 3.4.1 | Server-Side Architecture | 26 |
| 3.4.2 | Client-Side Architecture | 28 |
| 4 | Implementation Details of JSON Schema Maker | 32 |
| 4.1 | Third-Party Frameworks and Libraries | 32 |
| 4.2 | Implementing the Generator Part | 33 |
| 4.2.1 | Generator's Configuration Options | 35 |
| 4.2.2 | Code Snippets for JSON Validators | 35 |
| 4.3 | Component for Validating Against a Schema | 36 |
| 4.4 | Client-Side Implementation | 36 |
| 5 | Evaluation of Implemented Solution | 41 |
| 5.1 | Unit Tests | 41 |
| 5.2 | End-To-End Tests | 42 |
| 5.3 | Exploratory Testing | 42 |
| 5.3.1 | Demonstration of the Application's Functionality | 43 |
| 5.4 | Compatibility Testing | 43 |

| | |
|---|-----------|
| 6 Conclusion | 44 |
| 6.1 Unfinished Functionality | 44 |
| Bibliography | 45 |
| A Contents of the Included Storage Media | 47 |
| B Screenshots of the Web Application | 48 |

List of Figures

| | | |
|-----|--|----|
| 2.1 | Visual appearance of the Liquid Technologies web application | 13 |
| 2.2 | Example output generated by the Liquid Technologies | 14 |
| 2.3 | JsonSchema.net’s user interface for generating a JSON Schema | 15 |
| 2.4 | Edit dialog for manually changing the definitions of the generated schema by JsonSchema.net | 15 |
| 2.5 | Main part of quicktype’s user interface | 17 |
| 2.6 | Client/server model | 21 |
| | | |
| 3.1 | The initial layout of the web page | 25 |
| 3.2 | The modal dialog containing configuration options. | 26 |
| 3.3 | States of the application and transitions between them | 30 |
| | | |
| 4.1 | Class diagram describing different type generators. | 34 |
| 4.2 | Class diagram of the client-side implementation. | 36 |
| 4.3 | Preview of the EditorService class. | 37 |
| 4.4 | Preview of the TabService class. | 38 |
| 4.5 | Preview of the FileService class. | 38 |
| 4.6 | Preview of the Validator class. | 39 |
| 4.7 | Preview of the Utils class. | 39 |
| 4.8 | Preview of the App class. | 40 |
| | | |
| B.1 | Sample details of validation errors | 48 |
| B.2 | Final layout of the application | 49 |
| B.3 | Preview of the validation snippet modal dialog for Python. | 50 |
| B.4 | Example of a notification toast message. | 50 |
| B.5 | Preview of the unfinished schema store support. | 50 |
| B.6 | Modal dialog with configuration options. | 51 |

Listings

| | | |
|-----|--|----|
| 2.1 | Example of a JSON document | 6 |
| 2.2 | Example of a list validation | 9 |
| 2.3 | Example of a tuple validation with additionalItems | 9 |
| 2.4 | Example of a property dependency | 10 |
| 2.5 | Example of a schema dependency | 11 |
| 2.6 | Example of conditional branching in a JSON Schema | 11 |
| 2.7 | Simple HTTP GET request | 22 |
| 2.8 | Sample HTTP response | 22 |
| 4.1 | Example usage of the Json .NET Schema validator | 36 |
| 5.1 | Writing unit test with NUnit 3. | 41 |

Chapter 1

Introduction

Many systems depend on external data, provided by a user or another system, to fulfill their purpose. Every application expects a certain structure in its inputs. There are many formats in which the data can be represented. JSON is an interchange format widely used as a way of providing data to applications. It is easily readable for humans and also for machines to parse. However, the application must be aware of the fact that the data provided can have a different structure from what it expects.

That is when JSON Schema comes into play. JSON Schema is a piece of metadata for JSON documents. It contains various information about the structure, content, values, types, properties, and a couple of other useful definitions that describe how a JSON document should look like. It is not a standard yet, although there are multiple draft specifications already, the 2019-09 version being the most recent in the time of writing this paper.

Writing a schema manually for a semi-complex JSON could be very tedious as the resulting schema is typically considerably larger than the data it describes. Imagine, a JSON of 15 lines in length, its schema could be over a hundred lines long. This is the motivation for creating a tool that can automatically generate a schema for your existing JSON sample, and even multiple samples at once!

This paper introduces a web application with a code name *plexSON* whose purpose is to simplify the creation of the JSON schemas.

Chapter 2 presents the core technologies that this thesis works with—a JSON document and a JSON Schema. Then the currently available implementations for generating JSON schemas are discussed. It talks about the individual characteristics of each implementation and compares the advantages and the disadvantages of these tools. In the last section, the author presents the chosen technologies, programming languages, and protocols. Chapter 3 contains the design of the application. The architecture is documented in this part as well as the communications between the modules. On top of that, this chapter also includes a list of requirements and the design of the visual side. The description of the implementation details is in Chapter 4. It contains separate parts focusing on the back-end of the application as well as its front-end. Chapter 5 discusses how the application is tested and how the results were evaluated. Its purpose is to perform verification and validation of the final product. Lastly, Chapter 6 is the conclusion and it deals with the next steps, possible extensions, and opens space for further development.

Chapter 2

JSON Schema and Used Technologies

After reading this chapter, you will understand the basic purpose of schemas, particularly the JSON Schema and its format. As the name suggests, it is related to the JSON format—at first, because it describes JSON documents and secondly, it is also a JSON document itself. It is mandatory to understand what JSON is, to understand the JSON Schema. The explanation of the most frequently used terms in this thesis is here.

2.1 Introduction to JSON Schema

The building blocks of this project are JSON, and more importantly, JSON Schema. It is only right to start with the introduction to these terms.

2.1.1 JSON Document and its Structure

JSON stands for JavaScript Object Notation. It is a lightweight, text-based, language-independent data interchange format. It was derived from the ECMAScript Programming Language Standard. JSON defines a small set of formatting rules for the portable representation of structured data [3].

Its purpose is to serialize data, usually structures and/or collections, in a format similar to the JavaScript objects, which are key-value pairs. The format is human-readable and easy to parse by machines. The keys are always strings and according to the specification, they must be strictly double-quoted. This is different from the JavaScript's notation, where the keys can be single-quoted as well, or not quoted at all. The value can be an object, an array of values, a number, a string, or one of three literals: `true`, `false`, or `null`. For a comprehensive explanation and restrictions on the format, refer to the RFC specification [3]. Listing 2.1 denotes a simple JSON object.

```
{
  "checked": false,
  "dimensions": {
    "width": 5,
    "height": 10
  },
  "id": 1,
```

```
"name": "A green door",
"price": 12.5,
"tags": [
  "home",
  "green"
]
}
```

Listing 2.1: A sample JSON document (taken from [18]).

2.1.2 JSON Schema

A JSON value describing a certain person can have many forms. Imagine an object containing properties with the person's name, address as a single string with a street, a number and a city, and his or her phone number and work number. The same data can be, however, provided in a completely different way. For example, the same person can be described by two properties for the first name and the last name, an object property for the address, and an array of phone contacts. Now you want to make sure that your application receives the person's information in an expected structure. That is the primary purpose of any schema, JSON Schema not being an exception.

JSON Schema defines the media type `application/schema+json`, which is itself in the JSON format, for describing the structure of JSON data. JSON Schema asserts how a JSON document must look like, ways to extract information from it, and how to interact with it. To distinguish the data documents serialized in the JSON format from the schema documents, this thesis will use terms *JSON instance*, *JSON document*, and *JSON Schema document* as defined by the draft specification[19]:

- **JSON document** - an information resource (series of octets) described by the `application/json` media type.
- **JSON instance** - A JSON document which a schema is applied to.
- **JSON Schema document**, or simply a schema—a JSON document used to describe an instance.

Validation

At its core, JSON Schema provides validation keywords and annotations. The latter category does not have an influence on the validation result but provides additional information about the instance or the schema. A brief overview of the JSON Schema content follows [5].

Most of the time, JSON Schema is an object, whose properties serve to describe a JSON instance. However, some properties can contain one or more sub-schemas. Sometimes it can come handy to either accept or refuse everything, particularly in these sub-schemas. For this purpose, the JSON Schema can simply be a boolean value, where `true` always passes and `false` always fails the validation.

If the schema is an object, it typically contains a property `$schema`, which is used to distinguish JSON Schema from an arbitrary JSON data. Its value defines the schema version by the URI associated with the schema draft version, e.g.:

`http://json-schema.org/draft-07/schema#`.

Every schema can—and should—have its own unique identifier. This is achieved by setting the `$id` property to an absolute URI. This property also has another purpose. Mainly in conjunction with the `$ref` property, it provides a way of structuring complex schemas. In that case, it provides a base URL for relative `$ref` references in the same file without the need to use JSON pointers. More on the `$ref` later.

Since comments in JSON are forbidden, the `$comment` is used to bypass this, providing a way for the authors to include comments inside the schema. This keyword has no effect on the validation.

There are multiple ways of defining the structure of the instance. The most restrictive one is by specifying the exact value by the `const` property. It passes validation if and only if the value is exactly the same as defined by the keyword. The `enum` keyword works similarly, it provides an array of allowed values. The last option is restricting the type of value, which can be accomplished by the `type` keyword. It can be either a string stating which type should be allowed, or an array of such strings if we want to accept more types. Every type can further be restricted by some type-specific keywords.

Numeric types

The `number` and `integer` types can restrict the range of accepted values by the `maximum` and `minimum` keywords. Both have also an exclusive variant, `exclusiveMaximum` and `exclusiveMinimum` respectively. Additionally, the value can be restricted by `multipleOf`, which accepts numbers divisible by the specified number. All of these keywords' values have to be a number.

There are two numeric types defined by the JSON Schema since most of the programming languages distinguish integers and floating-point numbers. However, the JSON itself does not have distinct types for these two categories, and thus it is recommended to use additional checking on the mathematical value. To avoid situations where in some programming languages `1.0` would be accepted as an integer and in some it would not. The `multipleOf` keyword can be used to overcome this problem.

String type

The length of the string can be limited by the `minLength` and `maxLength` keywords, both numeric. The value itself can be limited by a regular expression, specified by the `pattern` property.

Despite the existence of a keyword that can be used for semantic validation—`format`—it serves as an assertion and an annotation. The specification [20] does not force the validators to treat it as an assertion influencing the validation outcome. Most of the validators do not provide complete support for all of the possible values. Those are:

- email, IDN-email,
- hostname, IDN-hostname, URI, URI-reference, IRI, IRI-reference, URI-template,
- IPv4, IPv6,
- JSON-Pointer, Relative-JSON-Pointer,
- date, time, date-time, and
- regex.

The internationalized variants (starting with *iri* and *idn*) are not implemented as assertions in the library which this project is using to simplify the manipulation with JSON data and JSON Schemas. The values are case-insensitive.

Array type

Arrays can be restricted in length (numeric `minItems`, `maxItems`), uniqueness of the items (boolean `uniqueItems`), and a check for presence of a specific item (the `contains` schema).

Furthermore, it is possible to define a schema for every item in the array. There are two ways possible to accomplish this with the `items` keyword: if it is a schema, then every item in the array must pass the validation against that schema—in this thesis, it will be referred to as a *list validation*. If it is an array of schemas, then every item is validated against the schema that corresponds to its position in the array (e.g. the first item is validated against the first schema in the `items` array, the second item against the second schema and so on). From now on, the term *tuple validation* will be used for this case. The difference between these two types is visible in Listings 2.2 and 2.3.

Beware, however, if not further limited by other keywords, an array **does not have to** contain as many elements as the `items` keyword specify. It can contain even more elements, as long as the `items` keyword validates successfully.

In case of a tuple validation, there is a possibility to define a schema for the items of the instance's array, whose position is greater than the `items` array. This is done by specifying the `additionalItems`. Note that in case of a list validation (i.e. when `items` is a schema, not an array of schemas) this keyword has no effect and it does not make sense at all.

```
{
  "type": "array",
  "items": {
    "type": "number"
  }
}
```

Listing 2.2: An example of a *list validation*. Every item in a given array must validate against the `items` sub-schema. In this case, it must be an array of numbers (taken from [5]).

```
{
  "type": "array",
  "items": [
    {
      "type": "number"
    },
    {
      "type": "string"
    },
    {
      "type": "string",
      "enum": ["Street", "Avenue", "Boulevard"]
    },
    {
      "type": "string",

```

```

    "enum": ["NW", "NE", "SW", "SE"]
  }
],
  "additionalItems": { "type": "string" }
}

```

Listing 2.3: An example of a *tuple validation* with `additionalItems`, limiting the fifth and higher elements of the array to be strings. In order to forbid more than four elements, that are described by the `items` schema, we could use `false` in `additionalItems`, or specify `maxItems` explicitly (taken from [5]).

Object type

The object type is the most interesting as for the validation possibilities. The number of properties can be limited by the `minProperties` and `maxProperties` numeric values.

If some properties should be always present, their keys can be included in the `required` array.

The main logic for validating object resides inside the `properties` keyword. It is an object containing a schema for every property name of the object in the tested instance. It is possible to specify a schema for properties without the need to know the exact name of the property. One way is to use `patternProperties`. It works the same as `properties`, with the exception that the property names are expressed as regular expressions rather than the whole names. Another way is using the `additionalProperties` which will be used for all of the properties that were matched by neither `properties` nor `patternProperties`. The names of the object's properties can also be tested against a schema, the `propertyNames` is used for this purpose.

The last keyword—`dependencies`—is probably the most complex one. It serves two purposes based on the presence of a certain property in the validated instance. The value of the keyword is an object, whose keys are property names and the values of those properties can be either an array of strings representing the property names of the object being validated, or it can be a schema.

If it is an array of strings, it means that if the specified property is present, then all of the properties in that array must be present in the object as well. This is not bidirectional, though. So when a property—whose name is specified in the array—is present in the object, it does not mean that there must also be present the property which is the key of the `dependencies` keyword. This type is known as a *property* dependency.

If it is a schema, then it extends the original schema to have other constraints. This is called a *schema* dependency. Listings 2.4 and 2.5 contain examples of both types.

```

{
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "credit_card": { "type": "number" },
    "billing_address": { "type": "string" }
  },
  "required": ["name"],
  "dependencies": {
    "credit_card": ["billing_address"]
  }
}

```



```
}  
}
```

Listing 2.4: An example of a *property dependency*. If an object contains a `credit_card`, it must contain `billing_address` as well. However, only `billing_address` or neither of the two, is valid (taken from [5]).

```
{  
  "type": "object",  
  "properties": {  
    "name": { "type": "string" },  
    "credit_card": { "type": "number" }  
  },  
  "required": ["name"],  
  "dependencies": {  
    "credit_card": {  
      "properties": {  
        "billing_address": { "type": "string" }  
      },  
      "required": ["billing_address"]  
    }  
  }  
}
```

Listing 2.5: An example of a *schema dependency*. This is another way of defining the same as Listing 2.4 (taken from [5]).

Combining sub-schemas

It is also possible to use conditional branching inside the JSON Schema. The keywords providing this functionality are `if`, `then`, and `else`. All accept a schema as their value. Any of the keywords can be omitted, since `then` and `else` are ignored in case the `if` is not present. The `if` schema does not have a direct impact on the validation result, it just controls which conditional branch will be used to further validation, the same semantics as programming languages use. An example can be seen in Listing 2.6.

Do not forget that JSON (and thus JSON Schema) should not have duplicate properties with the same key. This would mean that only one `if` could be present in every schema. This is true—to get around this, a cascade of conditions can be formed inside of an `allOf` array of schemas, see below.

There is also a keyword named `not`. The value for it must be a schema as well and it produces a negated result of that schema's result.

For combining sub-schemas, there are three keywords that can be used. All of them are arrays of schemas. The first one is `allOf`. The validation of this keyword passes if the instance is valid against *every* schema in the array. The second is `anyOf`. Here, the validation will succeed as soon as the instance is valid against *at least one* of the schemas in the array. The third keyword is `oneOf`. This time, the validation will pass if and only if the instance is valid against *exactly one* schema in the array, no matter which one it is.

```
{
```

```

"type": "object",
"properties": {
  "street_address": {
    "type": "string"
  },
  "country": {
    "enum": ["United States of America", "Canada"]
  }
},
"if": {
  "properties": { "country": { "const": "United States of America" } }
},
"then": {
  "properties": { "postal_code": { "pattern": "[0-9]{5}(-[0-9]{4})?" } }
},
"else": {
  "properties": { "postal_code": {
    "pattern": "[A-Z] [0-9] [A-Z] [0-9] [A-Z] [0-9]" } }
}
}

```

Listing 2.6: An example of conditional branching in a JSON Schema. Note that a schema can only contain one such conditional. To extend this example for other countries, we would need to wrap triples of `if`, `then`, and `else` inside of an `allOf` for further scaling (taken from [5]).

Annotations

Annotations are used for explanation purposes for humans and have informational character. They do not influence on the validation result. The `title` and `description` properties contain string annotations representing short and longer explanation of a given schema. The value of the `default` property can be of any type, as it denotes the default value of the described token. The `examples` property is an array whose items present how accepted values could look like. The values can be marked read-only and/or write-only by the `readOnly` and `writeOnly` boolean properties.

2.2 Existing JSON Schema Generators

Similar software solutions—that take a sample JSON instance and try to generate a JSON Schema which would describe it—already exist. Some are useful, some are barely usable. The reason for the creation of another tool is simple—none of the listed below meets all of the requirements listed in Section 3.2 completely. Here are just a few of them compared with a description of the main advantages and disadvantages of each one. Table 2.1 on the end of the section summarizes the current situation.

Liquid Technologies

Liquid Technologies provide a free converter of a JSON document to a JSON Schema¹. However, only the input is editable, the generated output can not be manually edited, only copied to the clipboard and edited elsewhere. The bigger disadvantage though, is that it generates only draft version 4 of the JSON Schema, which lacks the latest features, like *if-then-else* keywords and others. Similarly to many tools, this one accepts unquoted property names as a valid JSON, even though it is not valid according to the JSON specifications. Only one input can be entered and processed, which makes it impossible to update the schema directly. Figures 2.1 and 2.2 show how the input and the output of the application look like. Liquid Technologies provide a complementary tool to convert a JSON Schema to a JSON instance as well².

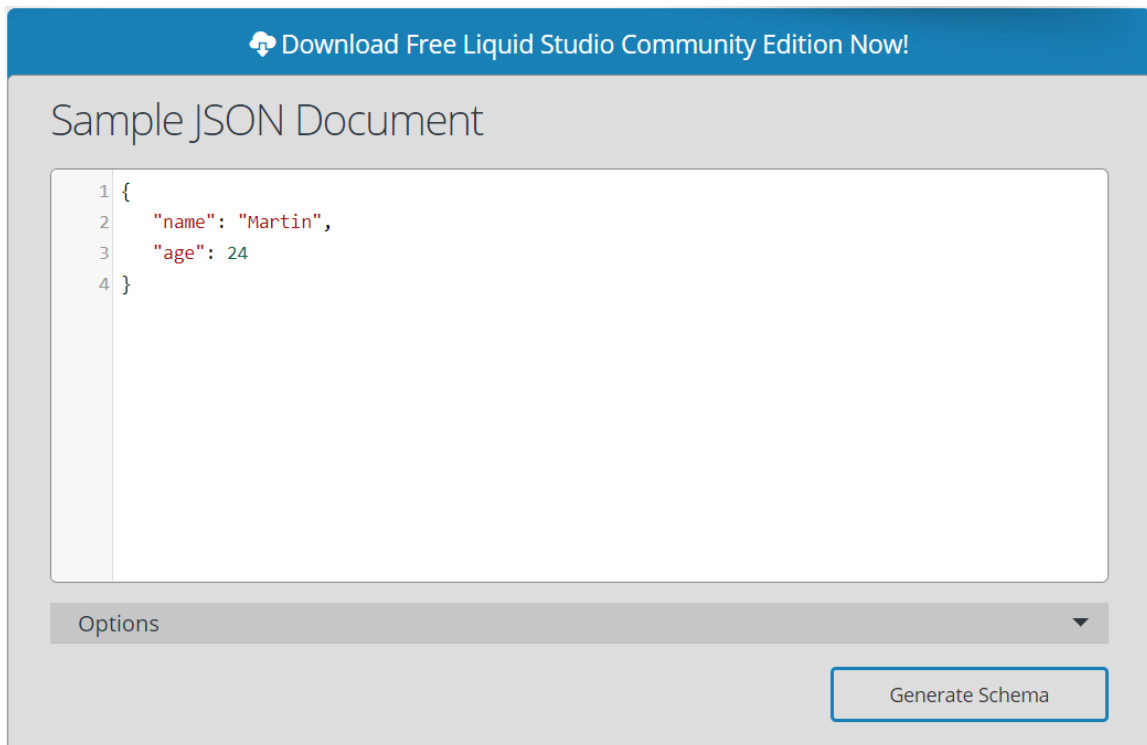


Figure 2.1: A part of the visual appearance of the Liquid Technologies web application.

Generate schema

Another tool is an open-source project called Generate Schema³. It provides a command-line interface to transform JSON objects to many different forms of schemas, like those for MySQL, Mongoose, Google BigQuery, and more, including JSON Schema. It comes as a Node.js⁴ package and is licensed under the MIT license. However, the disadvantages outweigh as the project's latest version was released in April 2018, with the last contribution to the repository dating July that year. It only supports JSON Schema draft version 4 and

¹<https://www.liquid-technologies.com/online-json-to-schema-converter>

²<https://www.liquid-technologies.com/online-schema-to-json-converter>

³<https://github.com/nijikokun/generate-schema>

⁴<https://nodejs.org/en/>



```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    },
    "age": {
      "type": "integer"
    }
  },
  "required": [
    "name",
    "age"
  ]
}
```

Figure 2.2: Example output for the input from Figure 2.1 generated by Liquid Technologies.

the generation can not be controlled by any configuration or user options apart from setting a title to the resulting schema. It can only generate the schema from a single input JSON which must be an object or an array.

JSONSchema.Net

The next tool is a web application again, it is called JSONSchema.Net⁵. The users can paste their JSON document and edit the following settings:

- select a draft version—available are 4, 6, and 7,
- select an identifier type—specifies the format of the \$id keyword, available are JSON pointer, plain name, hybrid, base URI, and none,
- select an array validation type—available are list validation, tuple validation, and allow anything,
- specify the absolute URI—fills the \$id of the root schema,
- pick annotations—specify which annotations will be present in the schema, e.g. title, description, default, and examples of each sub-schema,
- pick the restriction level—available are type, enum, and const,
- make all properties required for object types, and
- use only JSON numeric types.

There is also a possibility of using a *verbose* mode, where all of the keywords are present in the output, which is not directly editable manually. However, the definitions can be changed interactively by clicking one after switching to a tree view. An example of such edit dialog is presented in Figure 2.4. Apart from the JSON, the user can also select other formats of the output schema—YAML⁶ and XML⁷, which are other common serialization data formats.

⁵<https://jsonschema.net/home>

⁶YAML—YAML Ain't Markup Language

⁷XML—eXtensible Markup Language

Even though the project started in 2017, it is under active development. The last update to the application was in March 2020. As seen in Figure 2.3, the application does not provide a syntax highlighting except for emphasizing the keys of the properties. It is possible to provide only a single input JSON document, so for multiple input instances, the application would need to be used repetitively, and the resulting schema created manually.

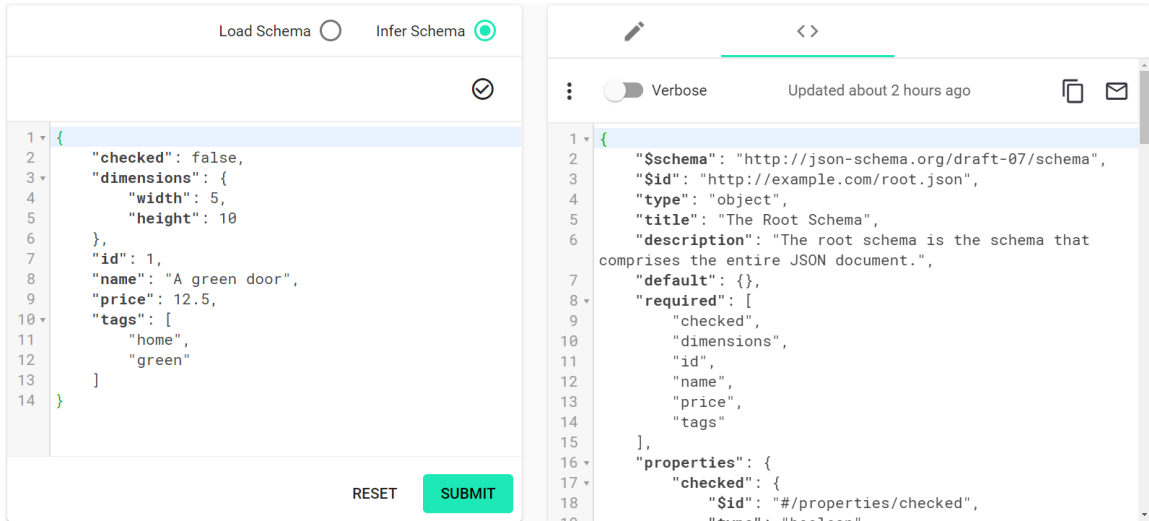


Figure 2.3: JsonSchema.net’s user interface for generating a JSON Schema.

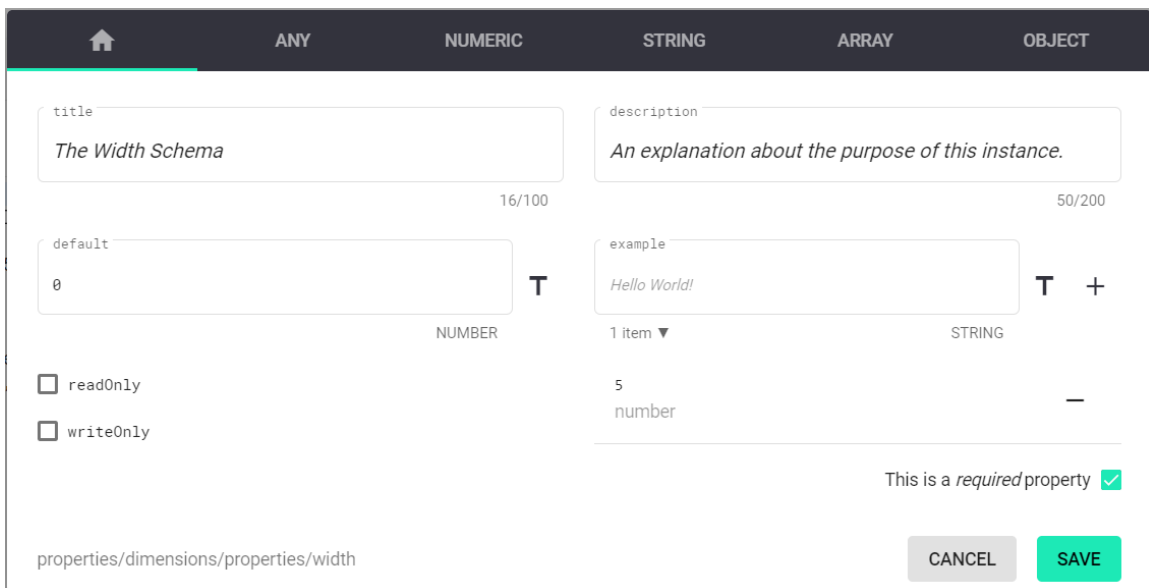


Figure 2.4: An edit dialog for manually changing the `width` definitions of the generated schema by the JsonSchema.net tool.

Schema Guru

Schema Guru is another command-line application developed by Snowplow Analytics Ltd. The strong advantage of Schema Guru is that it works with an unlimited set of input JSON

instances at once, which provides more precise results immediately, without the need to re-run with every input document. Unfortunately, that is all for its strong sides. The project is dead for years, as the latest version (0.6.2) was released back in April 2016 and it supports JSON Schema draft version 6. The project’s GitHub repository⁸ states that it comes with a demo web user interface, yet the link is not working anymore. The application is written in Scala language and is licensed under the Apache-2.0 license.

Quicktype

The last candidate is a very powerful tool, which comes in multiple forms—it is a command-line application with a web user interface and even as an extension to Integrated Development Environments (IDEs) like Visual Studio and VS Code⁹ by Microsoft or Xcode¹⁰ by Apple. The project is written in TypeScript language and is licensed under Apache-2.0. The latest release tag was added in December 2017 however, it is under active development. Contributions to the project’s GitHub repository¹¹ are almost daily. Generating a JSON Schema is only one of the application’s features. It can also be used for generating types in different programming languages such as JavaScript, C#, Python, Java, Go, C++, Kotlin, and even more. It supports different input types, too—JSON, JSON Schema, GraphQL queries, or TypeScript. Most importantly, it can handle multiple JSON inputs to provide the schema. The web application¹² provides an intuitive and easy-to-use interface. Figure 2.5 shows the web application. Unlike many of the web-based generators, this solution allows the user to edit the generated output directly in place. Both editors provide a nice syntax highlighting, which simplifies the usage even more. The only, yet quite significant downfall is that it still does not support at least JSON Schema draft version 7.

| Tool | License | Online | Latest release | Draft version | Syntax highlight | Multiple JSON inputs | Editable output |
|-------------------------------------|--------------------|------------|----------------|---------------|------------------|----------------------|-----------------|
| Liquid Technologies Generate Schema | proprietary | YES | N/A | 4 | YES | NO | NO |
| JsonSchema.net | MIT | NO | 07/18 | 4 | NO | NO | NO |
| Schema Guru | Apache 2 | YES | 03/20 | 7 | NO | NO | YES |
| Quicktype | Apache 2 | NO | 05/16 | 6 | NO | YES | NO |
| PlexSON | Apache 2 | YES | 12/17 | 6 | YES | YES | YES |
| | CC BY-NC-SA | YES | 05/20 | 7 | YES | YES | YES |

Table 2.1: The existing solutions compared with the product of this thesis, based on the selected features.

⁸<https://github.com/snowplow/schema-guru>

⁹<https://visualstudio.microsoft.com/>

¹⁰<https://developer.apple.com/xcode/>

¹¹<https://github.com/quicktype/quicktype>

¹²<https://app.quicktype.io/#l=schema>

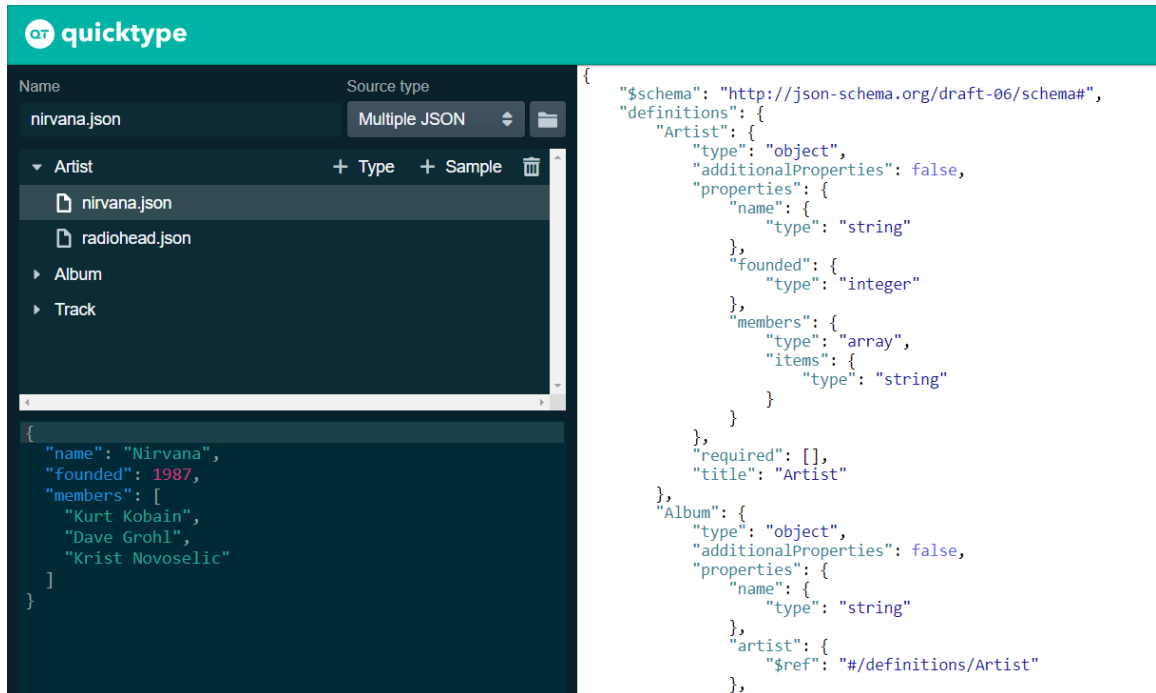


Figure 2.5: The main part of quicktype’s user interface showing how to generate a schema from multiple inputs.

2.3 Existing JSON Schema Validators

Instead of implementing the JSON Schema validator from scratch, I decided to use one from a long list of already existing validators¹³. When picking the most suitable one, the following criteria were considered, with assigned importance (weight) of their error feedback:

- supports at least draft version-07,
- finds all errors (4), and
- the validation error contains
 - a user-friendly message (2),
 - the value that failed the validation (2),
 - a path to the value in the instance, line number and position (1+1), and
 - a path to the violated definition in the schema, line number and position (1+1)

Following is a shortlist of the validators compared based on the given criteria. All mentioned validators support at least JSON Schema draft version 7. Table 2.2 sums up the comparisons with the assigned scores.

Json .NET Schema

The Json .NET Schema¹⁴ library is written in .NET Core. It comes with several license tiers, but luckily, one of those is a free AGPL-3.0 license. This edition is limited to 1000 validations

¹³<https://json-schema.org/implementations.html#validators>

¹⁴<https://github.com/JamesNK/Newtonsoft.Json.Schema>

per hour, which is sufficient for this project's needs. This library can be used to parse a JSON string into an internal representation, manipulate it, validate an instance against the schema, serialize, and de-serialize .NET types. It provides detailed validation output, from the set of the requested information, the following are present: path in the instance, line number and position in the instance, path in the schema, message, and value. The validation finds all errors. Even though the latest version is released a couple of months already, it is still under active development.

Manatee.Json

Another validator is Manatee.Json¹⁵, which is very similar to Json .NET Schema. It is also written in .NET Core and provides similar features. It differs in the license—which is MIT—and the way it reports validation errors. The user can choose from different forms of structuring the error outputs, from a flat list of errors to a nested hierarchy following the structure of the schema. The most detailed one contains these from the set of requested information: path in the instance, path in the schema, message, and value. The validation finds all errors. The latest release is tagged from 2015, but the development has not stopped since.

AJV

Another JSON Schema Validator¹⁶ (AJV) is a JavaScript implementation for Node.js and browser. It can also be used as a command-line interface tool. One of its unique features is the ability to define custom keywords. By default, the validation returns results after the first error is encountered, but this behaviour can be turned off, to continue and find all of the errors. From the set of the requested information, the error records have the following structure: path in the instance, a path in the schema, and a message about the error. In a verbose mode, the error record additionally contains the validated data.

Networknt

The next candidate is JSON Schema Validator¹⁷ by Networknt. It is written in Java and is licensed under Apache-2.0 License. The latest version was released in April 2020. It supports the latest JSON Schema draft version and the result of validation is a set of validation errors, where from the set of the requested information, each (if any) contains a message and a path in the instance.

Fast JSON Schema

Another implementation is called Fast JSON Schema for Python¹⁸. As the previous sentence suggests, it is written in the Python language, but only supports version 3.3 and higher of this language. It has very good contribution rate since almost every month a new contribution is submitted to the repository. It is licensed under BSD-3-Clause. In case of a failed validation, it raises an exception which, from the set of the requested information, contains: message, value, path in the instance document. The disadvantage is that it reports the first error and exits, so it does not find all errors.

¹⁵<https://github.com/gregsdennis/Manatee.Json>

¹⁶<https://github.com/epoberezkin/ajv>

¹⁷<https://github.com/networknt/json-schema-validator>

¹⁸<https://github.com/horejsek/python-fastjsonschema>

Jsonschema

Jsonschema¹⁹ is another Python implementation of a JSON Schema validator. This one works with Python version 2.7 and higher. The author contributes very often. The application can be also used from the command-line. Its performance is not as fast as that of the Fast JSON Schema, but it finds all errors. The validation errors contain plenty of information, but from the requested ones, it provides: message, path in the instance, and path in the schema.

Swaggest

The next one is a JSON Schema implementation for PHP²⁰. Similarly to many others, this project is licensed under the MIT license. With multiple versions every quarter, it is no doubt that the development is alive and perpetual. Even though the error message contains much information about what went wrong during the validation, it is just one string, which—in some cases—can be an indented multi-line string. From the set of the requested information, it contains a brief message and a path in the schema which triggered the validation error. However, the validation finds only the first error and ends. On the other hand, *Swaggest* allows creating JSON Schemas programmatically in an Object-Oriented manner.

Opis JSON Schema

Another PHP implementation is called Opis JSON Schema²¹. This time, it is licensed under Apache-2.0 license and requires PHP 7 or later. In addition to all the keywords defined by the JSON Schema specification, this library introduces a set of custom keywords like `$var`, `$map`, and a few more. The result of a validation process is an object with properties representing the success and a complement signalling whether there were errors. In such case, from the set of the requested information, every error contains the data that did not correspond and a path to that data in the instance. It now has been some time since the latest version as the current release dates back to August last year.

Qri

The last candidate is a module of a solution called Qri²², which comes as a desktop application as well as a command-line interface. It is implemented in the Go language and the JSON Schema module is MIT licensed. The latest contribution was made in the fall last year. The default validator produces error records with the following from the set of requested information: message, value, path in the instance, and path in the schema.

¹⁹<https://github.com/Julian/jsonschema>

²⁰<https://github.com/swaggest/php-json-schema>

²¹<https://github.com/opis/json-schema>

²²<https://github.com/qri-io/jsonschema>

| Tool | Draft version | Error feedback | Contribution | License |
|------------------|---------------|----------------|--------------|--------------|
| Json .NET Schema | 7 | 11 | 05/20 | AGPL-3.0 |
| Manatee.Json | 2019-09 | 10 | 05/20 | MIT |
| AJV | 7 | 10 | 05/20 | MIT |
| Networknt | 2019-09 | 7 | 05/20 | Apache 2 |
| FastJsonSchema | 7 | 5 | 03/20 | BSD-3-Clause |
| Jschema | 7 | 8 | 05/20 | MIT |
| Swaggest | 7 | 6 | 04/20 | MIT |
| Opis | 7 | 7 | 08/19 | Apache 2 |
| Qri.io | 7 | 10 | 05/20 | MIT |

Table 2.2: Shortlist of the compared validators.

As a result, Json .NET Schema was chosen as a library for the back-end of the application based mainly on the level of detail which it provides as part of the error feedback and also because it is a complete tool-set for manipulation with JSON Schema, such as creating and/or updating the definitions of the schema. On top of that, the fact that it can be referenced from a .NET Core application allows for the consuming project to be multi-platform.

2.4 Used Technologies and Principles

This section lists and briefly describes the technologies and principles that this thesis and mainly the implementation are based on.

2.4.1 Hyper Text Markup Language & Cascading Style Sheets

HTML is the most basic building block of the Web. It was first published as an Internet draft in 1993. The language was created by Tim Berners-Lee, and today it is maintained by W3C, WHATWG, and IETF organizations [13]. The latest version is HTML5, which is used in this project.

CSS is a stylesheet language used to describe the presentation of a document written in HTML or XML (including XML dialects such as SVG, MathML, or XHTML). CSS describes how elements should be rendered. It was developed by Håkon Wium Lie [12] and is maintained by the W3C organization. The latest version is CSS3, also used in this project.

2.4.2 JavaScript

JavaScript is a dynamic, lightweight, interpreted, object-oriented language with first-class functions and is best known as the scripting language for Web pages [15]. It is the most popular and used technology for client-side development. It was created by Brendan Eich at Netscape in 1995, originally called Mocha and LiveScript [17]. Netscape submitted the language for standardization to the European Computer Manufacturer’s Association (ECMA), and because of trademark issues, the official name of JavaScript is “ECMAScript”. For the same trademark reasons, Microsoft’s version of the language is formally known as *JScript* [8]. The latest version is ECMAScript 2019 (the 10th Edition). This project is using version ECMAScript 2016 (the 7th edition).

2.4.3 C# and .NET Core

C# is a general-purpose, type-safe, object-oriented programming language. The chief architect of the language since its first version is Anders Hejlsberg (the creator of Turbo Pascal and architect of Delphi). The C# language is platform-neutral and works with a range of platform-specific compilers and frameworks, most notably the Microsoft .NET Framework for Windows [1]. The disadvantage of the .NET Framework is that it runs only on Windows OS.

This issue is solved by .NET Core. It is a cross-platform (**R_Multiplatform**), open-source, and modular .NET platform for creating modern web apps, micro-services, libraries, and console applications [10]. It is licensed under the MIT License. This thesis uses .NET Core version 3.1 which is the latest version available.

The next release is scheduled to be released on November 2020 and it will be named .NET 5 [11].

2.4.4 Client/Server Model

Client/server is an architectural pattern, which consists of two independent, computational systems. Each of these two systems has its own role in their relationship. The communication is always initiated by the client machine, which makes a request to the server. The latter processes the request and optionally evaluates eventual input data. After the operation, it responds with either the result or, in case of a problem, an error back to the client.

One server can listen to and serve multiple clients. This model is depicted in Figure 2.6. Typically, the communication is network-based and uses the Hypertext Transfer Protocol, described in the next section. The client is usually equipped with a user interface to interact with the user.

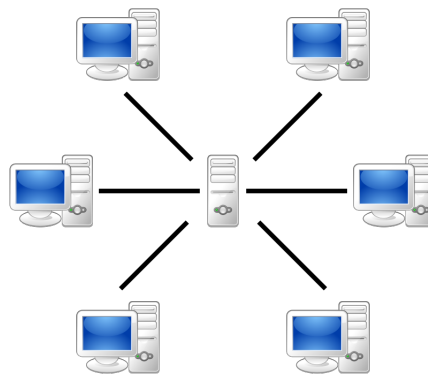


Figure 2.6: A visual representation of a client/server model (taken from [4]).

2.4.5 Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is used as an underlying technology for communication between the client-side and the web application programming interface. HTTP is a stateless application-level protocol for distributed, collaborative, hypertext information systems [7]. The communication is divided into requests and responses. HTTP defines several methods, each with a different use. The most commonly used ones are summarised in Table 2.3, along with their intended use.

Every HTTP request has a target, called a *resource*, which is uniquely identified by a Unified Resource Identifier (URI). Along with the resource, each request must also include a *method*, the version, *headers*, and optionally *payload* data separated by an empty line. Listing 2.7 presents a simple HTTP request.

| HTTP method | Meaning |
|-------------|---------------------|
| GET | Retrieve a resource |
| PUT | Upload a resource |
| POST | Send data |
| PATCH | Update a resource |
| DELETE | Remove a resource |

Table 2.3: The most common HTTP methods and their usage.

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.71 zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
```

Listing 2.7: A simple HTTP GET request (taken from [6]).

The responses have a slightly different structure. It starts with the version, a numeric *status code*, and a textual *reason phrase*. The latter double indicates the outcome of the operation. It is followed by *headers* ended with an empty line, and the *payload* data come last. Listing 2.8 shows how such a response could look like.

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

```
Hello World!
```

Listing 2.8: A sample HTTP response to a request from Listing 2.7 (taken from [6]).

2.4.6 Web Application Programming Interface

An application programming interface (API) can be perceived as a contract. This contract then makes the connection between the author developer and the consumer (typically also a developer) much more efficient since the interfaces are documented, consistent, and predictable [9].

Web Application Programming Interface (Web API) is a pattern used for communication between arbitrary computer programs, mainly (but not limited to) web browsers controlled by users to access a website or a service on a remote machine. This communication is realized by the HTTP requests and responses [2].

One of the most common and used types of an API is a RESTful API, which conforms to the REST²³ protocol. RESTful APIs can be used in situations where CRUD²⁴ operations on resources are required. It suits best with database-based applications or other data-oriented ones. However, this project does not require all these actions, it only provides a few actions like validating JSON documents against a schema and/or generating a schema based on the input instance. In Chapter 4, you will find more details on how it has been implemented.

²³REST—Representational State Transfer

²⁴CRUD—Create, Retrieve, Update, Delete

Chapter 3

Design of JSON Schema Maker

The analysis and design are an integral part of any software development cycle. Understanding and defining the requirements precisely allows for faster development and higher quality of the end product. This chapter intends to introduce the reader with the motivation for creating a new application and explain its architecture.

3.1 Outlining the Final Product

The core of the system will run as a web service on a server, providing a simple web API. This service will be bound to a web user interface, and it will provide two basic operations: generating a schema and validating an input JSON against a schema. The former will be user-initiated, the latter will be triggered regularly, as needed, mostly by changing either input(s) or the schema. This way, the usage of the application will be simple, without the need of downloading and setting up any sources or libraries. The only prerequisites will be a browser¹ and an internet connection.

3.2 JSON Schema Maker Requirements Specification

The primary goal of this application is to easily validate JSON documents. For every system that comes into contact with some external data, the best practice should be to validate them. This should not be an exception with JSON data to preserve safety, even though the JSON Schema is not yet fully standardized. Furthermore, creating a schema by hand can be really time-consuming, error-prone, and very ineffective.

That is the second goal that this application addresses. The users will benefit from generating the schema. All they need to do is to provide some samples of JSON documents and the skeleton for their schema will be ready in one click. It is vital to point out the word *skeleton* from the previous sentence. It is still a machine-generated piece of information which, in many cases, can not cover all of the semantics of the original data. So the user is encouraged to review the output and change the details to fit his needs. Many of the current generators lack this important feature.

What about another scenario, when a user already has a schema, but maybe wants to include new data. Let us say that a policy has changed, and now his system does not require certain properties, but their presence is not prohibited either, or include new fields. Instead

¹The client-side implementation has a few requirements for all the features to function properly, described in Section 5.1.

of manually adding and deleting definitions in the schema, updating it will be as easy as pasting it in the editor and specifying new samples. The application will then return an updated schema which will include the new definitions.

Since the very beginning of the project, some requirements emerged naturally, based on the assignment itself. Others arose during a deeper analysis. The following is a list of requirements posed on the resulting product. Table 3.1 contains the requirements with a description and also references to the sections where they are discussed.

After exploring the existing solutions (see Section 2.2) for use-cases defined in this section, there is no really a candidate that would satisfy all those conditions.

3.3 Designing Graphical User Interface

Creating a good, simple, and intuitive user interface can be a challenging task. As the first impression can really make a difference, it is important to pay enough attention to its design.

As depicted by Figure 3.1, the layout of the web page is divided into two main parts that occupy most of the page, since those are the most important areas. On the left-hand side is a place for the editor for input documents. This is where users will probably spend the most time at. The black rectangles represent the editors, which the users will interact with. The editor on the right half is for the schema. Just under both editors, there are two file pickers, one for every editor. At the bottom is a palette with control buttons. The left-most button opens up a dialog for configuration options. The dialog itself is shown in Figure 3.2. There are buttons for clearing the contents of the editors. These buttons are visually distinguished from the other buttons, they are red to express attention. The main control button, which starts the generating process, is also emphasized by a darker color. Remaining buttons are in light grey color.

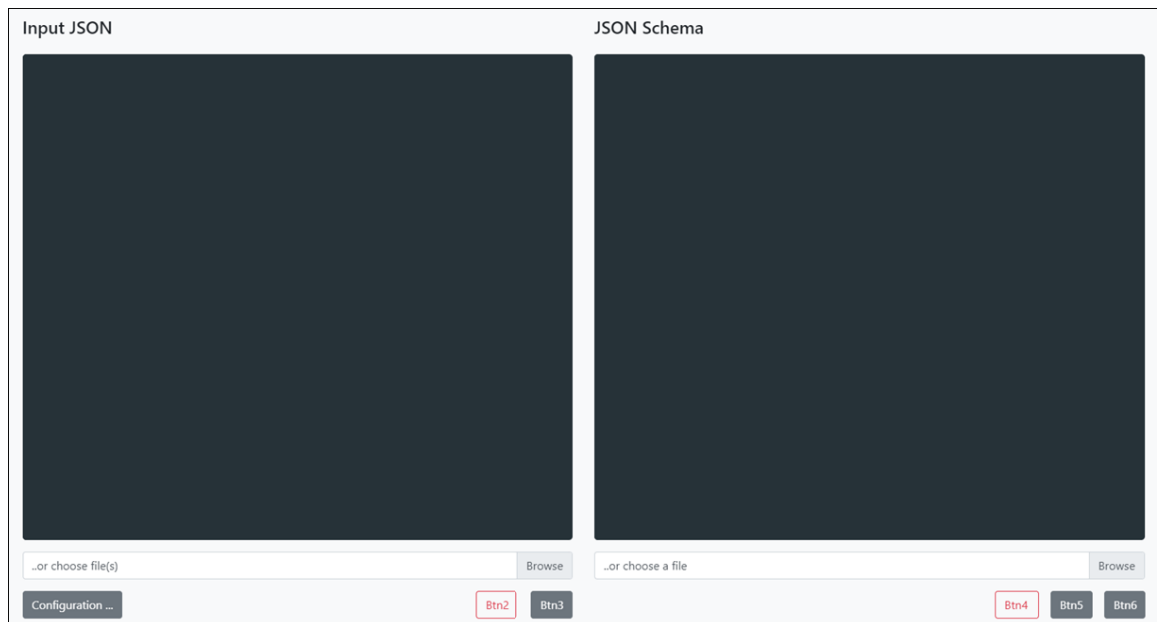


Figure 3.1: The initial layout of the web page. The black areas represent the editors. File pickers are underneath and a palette with placeholder buttons is at the bottom.

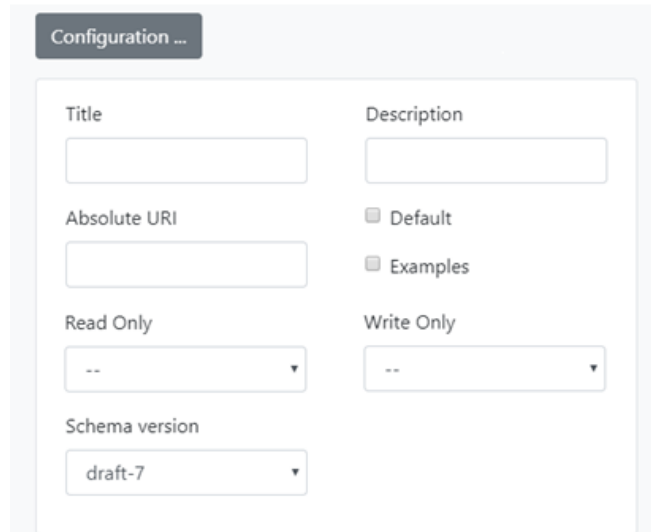


Figure 3.2: The modal dialog containing configuration options.

The final design of the user interface has come through several changes and remakes. Originally, everything was planned to be displayed on the page, with no hidden elements. This was not an optimal solution since there would be less room for the important parts—the editors. Because of this, the configuration has been designed to be an area appearing outside of the view-port at first. This did not look very user-friendly, so a modal dialog was finally picked as a final decision.

3.4 Architectural Overview of the Application

Since the initial planning phase, the application was designed with the client/server model in mind. From the high-level architecture perspective, the system is not too complex—there is no need for a database, the authentication, concurrency, nor routing. Therefore, monolithic architecture was chosen. The client and the server sides will be described separately.

3.4.1 Server-Side Architecture

The server side of the application is rather straightforward in terms of design. For validating an input against a schema, a simple wrapper over the `Json .NET Schema` framework² is sufficient, as it provides all the functionality needed. Two arguments are all that is needed to be provided—a JSON document to be validated and the schema used for validating. Table 3.2 then represents the body of an HTTP request.

| Property | Type | Meaning |
|----------|--------|---------------------------------|
| Json | string | JSON document to be validated |
| Schema | string | JSON Schema used for validating |

Table 3.2: Body structure of an HTTP request for validation.

²<https://github.com/JamesNK/Newtonsoft.Json.Schema>

As for generating a schema, the starting point is the original content (an empty schema for the first time). Then there is a couple of options on how to process the input documents and create the resulting schema. One way would be merging those inputs into a single super-document and then treat this as a single input. This would be a rather non-trivial task since the inputs can be very different from each other and it would require complete pre-processing of the structures. Another possibility is generating the resulting schema incrementally. First, an inter-schema is generated from each input independently. Then, these are merged together to form the final schema (**R_Iterative**). This option was considered more suitable and was implemented.

The way how the schema will be generated is configurable with a few options. These include the presence of selected annotations or setting strict policy on the presence of an object's properties. This inclines to define a request structure which is explained in Table 3.3.

| Name | Type | Meaning |
|----------------------|-----------|--|
| InputInstances | string[] | Collection of JSON instances |
| Schema | string | The current value of the schema (initially empty) |
| Id | bool | Controls the presence of the \$id in the nested schemas |
| SchemaVersion | number | Desired draft version of the schema |
| AbsoluteUri | string | The value for the \$id of the root schema |
| AllRequired | bool | Specifies if all object's properties are mandatory |
| AdditionalProperties | bool | Allows additional properties not present in the properties keyword (objects only) |
| AdditionalItems | bool | Allows additional items not present in the items keyword (arrays only) |
| Title | bool | Controls the presence of the title annotation in the result |
| Description | bool | Controls the presence of the description annotation in the result |
| Default | bool | Controls the presence of the default annotation in the result |
| Examples | bool | Controls the presence of the examples annotation in the result |
| ReadOnly | bool/null | Controls the presence/value of the read-Only annotation |
| WriteOnly | bool/null | Controls the presence/value of the writeOnly annotation |

Table 3.3: The body structure of an HTTP request for generating a schema.

One more operation will be available, and it will generate a code snippet in a given programming or scripting language which will serve as a validator of the JSON instances against the current content of the schema. The request will need to include information about the desired language and obviously the schema. The language is an enumeration with the values described later in Section 4.2.2. This request is portrayed in Figure 3.4.

| Name | Type |
|---------------------|--------|
| ProgrammingLanguage | number |
| Schema | string |

Table 3.4: The body structure of an HTTP request for generating a validation code snippet.

The server exposes a simple API for these three operations. Table 3.5 recaps the interface.

| Operation | Endpoint URI | Request body |
|-------------------------|----------------------|---------------|
| Validation | api/validate | See Table 3.2 |
| Generating a schema | api/generate/schema | See Table 3.3 |
| Generating code snippet | api/generate/snippet | See Table 3.4 |

Table 3.5: Web API exposed by the server. All three operations are used with the HTTP POST method.

Similarly, the responses will have a predefined structure whose purpose is easy processing on the client-side. In case when everything goes fine, and the operation finishes successfully, the response to a generation request contains the contents of the generated schema. The validation response is composed of a boolean representing the status of the validation. In addition to this flag, it contains a list of eventual errors, where each record can contain nested error records. The complete structure of the error record is the same as discussed in Section 2.3.

When the request finishes successfully, a 200 OK status code is used. On the other hand, if it ends with a failure—often because of invalid request data—the response results in a 400 Bad Request status code with more information in the response body about the cause of the failure.

3.4.2 Client-Side Architecture

The client-side is all about the interaction with the user. Its purpose is to provide ways to present the results to the user and to further manipulate with it. Since there is no need for persisting data, the application is implemented as a single page (**R_Single-page**). It also takes care of reading local files (**R_Upload**) and handling drag and drop (**R_Drop**) actions. To allow easy interaction with multiple inputs at the same time (**R_Inputs**), the editor for inputs uses a concept of tabs to quickly navigate between the currently open input instances (**R_Multi_input**). The results of a validation aim for user-friendly presentation so the incorrect parts can be easily located in both editors (**R_Error**, **R_Invalid**).

The possible states and activities are shown in Figure 3.3. Each input instance holds its state, meaning that one input can be in state *invalid JSON* and another in the *corresponding* state at the same time. Considering the state that the system is currently in, different actions are possible to execute. All of the actions are triggered by user interactions, except for the validation, which is executed whenever the contents of the editors change. Changing the configuration does not change the state. Editing the inputs and the schema can be performed by typing, undoing, redoing, pasting, uploading, or dropping files.

| Requirement ID | Description | Section |
|------------------------|--|-----------------|
| R_Generator | The application generates a valid JSON Schema. | 4.2 |
| R_Inputs | The application accepts one or more input JSON instances to generate the schema from. | 3.4.2, 4.2 |
| R_Version | The application supports at least draft-07 version of JSON Schema. | 4.4 |
| R_Validate | The application validates the input document against the current content of the schema regularly. | 4.3 |
| R_Error | A user-friendly message is provided upon failed validation of input against a schema. | 3.4.2, 4.1, 4.4 |
| R_Editors | Input and output text areas are editable. | 4.1, 4.4 |
| R_Syntax | Editors of JSON and JSON Schema provide syntax highlighting of their contents. | 4.1 |
| R_Invalid | Editors of JSON and JSON Schema underline an invalid part of their content. | 3.4.2, 4.1 |
| R_Config | The user can modify the configuration to control how the schema will be generated. | 4.2.1 |
| R_Config_1 | The user can provide a URI to be used as an \$id of the root schema. | 4.2.1 |
| R_Config_2 | The user can toggle the presence of the \$id keyword in nested keywords and sub-schemas. | 4.2.1 |
| R_Config_3 | The user can mark the schemas to be read-only and/or write-only. | 4.2.1 |
| R_Config_4 | The user chooses which annotations (title, description, default, and examples) will be included in the generated schema. | 4.2.1 |
| R_Config_5 | The user can control if all properties of objects are required. | 4.2.1 |
| R_Config_6 | The user can control if additional properties of objects are allowed. | 4.2.1 |
| R_Config_7 | The user can control if additional items in arrays are allowed. | 4.2.1 |
| R_Iterative | The application generates the output schema in an iterative manner if used with more than one input JSON instance. | 3.4.1, 4.2 |
| R_Multiplatform | The application is platform-agnostic. | 2.4.3 |
| R_Single-page | The tool is a single-page web application. | 3.4.2 |
| R_Multi_Input | Editor for input documents can have multiple input JSON documents opened at once. | 3.4.2 |
| R_Upload | The user can upload input files as well as a schema. | 3.4.2, 4.4 |
| R_Drop | The user can add files by dragging and dropping to the editor areas. | 3.4.2, 4.4 |
| R_Tests | The application's functionality is covered by automated tests. | 5 |

Table 3.1: Requirements posed on the JSON Schema Maker.

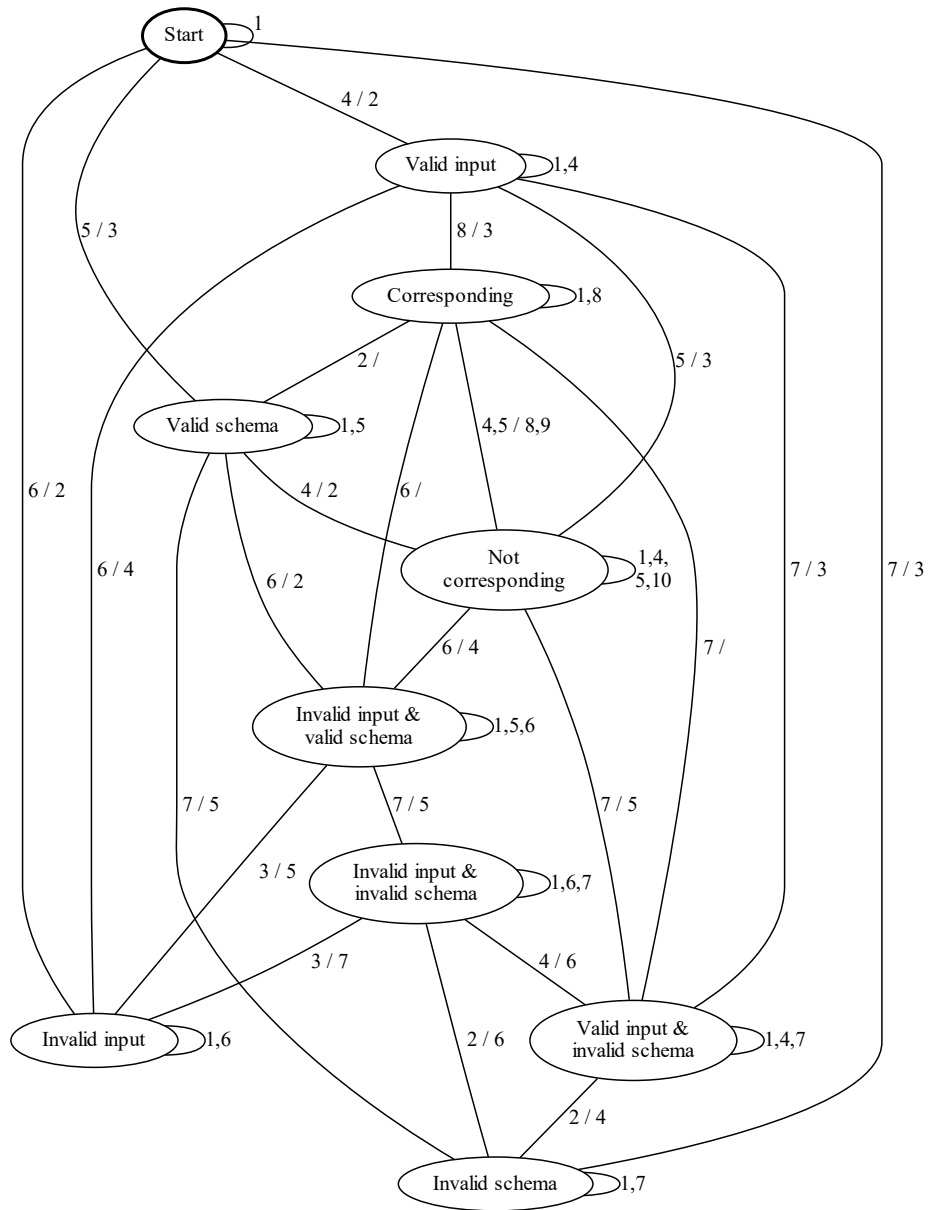


Figure 3.3: Different states of the application and transitions between them. This is not a representation of a „guard/action“! It uses this signature for simplifying purposes. Most of the transitions are bi-directional. Actions for different orientations are separated by a slash (/) to save space and keep the diagram clean and readable. The first number is directed downwards, the optional second heads upwards. The terms (*in*)*valid* refer to the syntactic correctness of a JSON format, while (*not*) *corresponding* mean the validity against the schema. The transitions are explained in Table 3.6.

| Transition # | Explanation |
|---------------------|------------------------------------|
| 1 | Edit configuration |
| 2 | Clear input |
| 3 | Clear schema |
| 4 | Edit input to valid JSON |
| 5 | Edit schema to valid JSON Schema |
| 6 | Edit input to invalid JSON |
| 7 | Edit schema to invalid JSON Schema |
| 8 | Generate JSON Schema |
| 9 | Validation success |
| 10 | Validation failure |

Table 3.6: Explanation of transitions in Figure 3.3

Chapter 4

Implementation Details of JSON Schema Maker

This chapter discusses the implementation details of *plexSON*. The implementation started with an HTML template page and basic styling. Then I continued with the back-end development. The web API was created with the validator, and the generator part was added subsequently. Lastly, the client-side was implemented as a presentation layer.

4.1 Third-Party Frameworks and Libraries

There were several third-party libraries used in this project to simplify the development. Most of them are licensed under MIT if not stated otherwise, and they are documented in this section.

Bootstrap

This project uses the popular front-end framework Bootstrap¹. It is an open-source project created at Twitter². Using Bootstrap, styling a web page is accomplished by applying special classes directly to the HTML elements. The main advantage of using this library is a responsive design out-of-the-box. However, the responsiveness is only partial. The current version of the framework is 4.4. It also provides a lot of icons, which are used instead of some buttons captions.

jQuery

To simplify traversing and manipulating the HTML document and its elements and to unify the JavaScript interpretation across different browsers, this application is using the jQuery³ library. This library can be also used for making asynchronous calls from the client JavaScript to a server, but another way was used for this purpose which will be described later. The latest version of the jQuery framework is currently 3.5, while in this project is used the version 3.4.1.

¹<https://getbootstrap.com/>

²<https://twitter.com/>

³<https://jquery.com/>

Fetch API

The Fetch API provides an interface for fetching resources (including across the network). It also defines related concepts such as CORS⁴ and the HTTP origin header semantics [16]. The Fetch API conforms to the Fetch Standard⁵. The request can be made by invoking the global `fetch()` method.

Monaco Editor

The Monaco Editor is the code editor that powers Visual Studio Code IDE. It supports Classic Edge, Edge, Chrome, Firefox, Safari, and Opera browsers [14]. It is an open-source project of Microsoft, written in TypeScript language. For this thesis, the version 0.19.3 is used, which is the second latest one. The latest version available is 0.20.0.

Monaco already has strong built-in support for JSON language mode, including syntax highlighting (**R_Syntax**) and validation of JSON syntax (**R_Invalid**). Furthermore, it also provides validation against a JSON Schema, but this feature is not used in order to craft own, more accurate error feedback upon failed validation (**R_Error**).

Except for Monaco, other web components were considered for realizing the editors, mainly CodeMirror⁶ and Ace⁷, but Monaco provides the best functionality and user experience for this particular use-case.

Newtonsoft

As for the back-end production code, two libraries were used as NuGet⁸ packages. They are both from the same author and historically they were part of one package, but Json .Net Schema was later extracted to its own package.

Json .NET⁹ is the most popular .NET library overall. It is a complete framework for JSON serialization and deserialization.

Json .NET Schema¹⁰ extends the general Json .NET package with operations related to JSON schemas, like creating, generating, and validating. This package is used under the AGPL 3.0 license as described in Section 2.2.

4.2 Implementing the Generator Part

The generation of resulting schema is a multi-step process, as mentioned in Section 3.4.1. After checking all the input parameters, a type-specific generator's `FillSchema` method is called to generate applicable schema definitions. The decision which generator should be used is determined by the type of the input instance itself. Different type generators are displayed on the class diagram, in Figure 4.1.

When all inputs have their own schema generated (**R_Inputs**), they are all combined together to create the resulting schema. Starting with the original schema, the doubles of the schemas are merged into one and in this way reduced to one, final schema (**R_Iterative**).

⁴CORS—Cross-Origin Resource Sharing

⁵<https://fetch.spec.whatwg.org/>

⁶<https://codemirror.net/>

⁷<https://ace.c9.io/>

⁸NuGet is a package manager for .NET

⁹<https://www.newtonsoft.com/json>

¹⁰<https://www.newtonsoft.com/jsonschema>

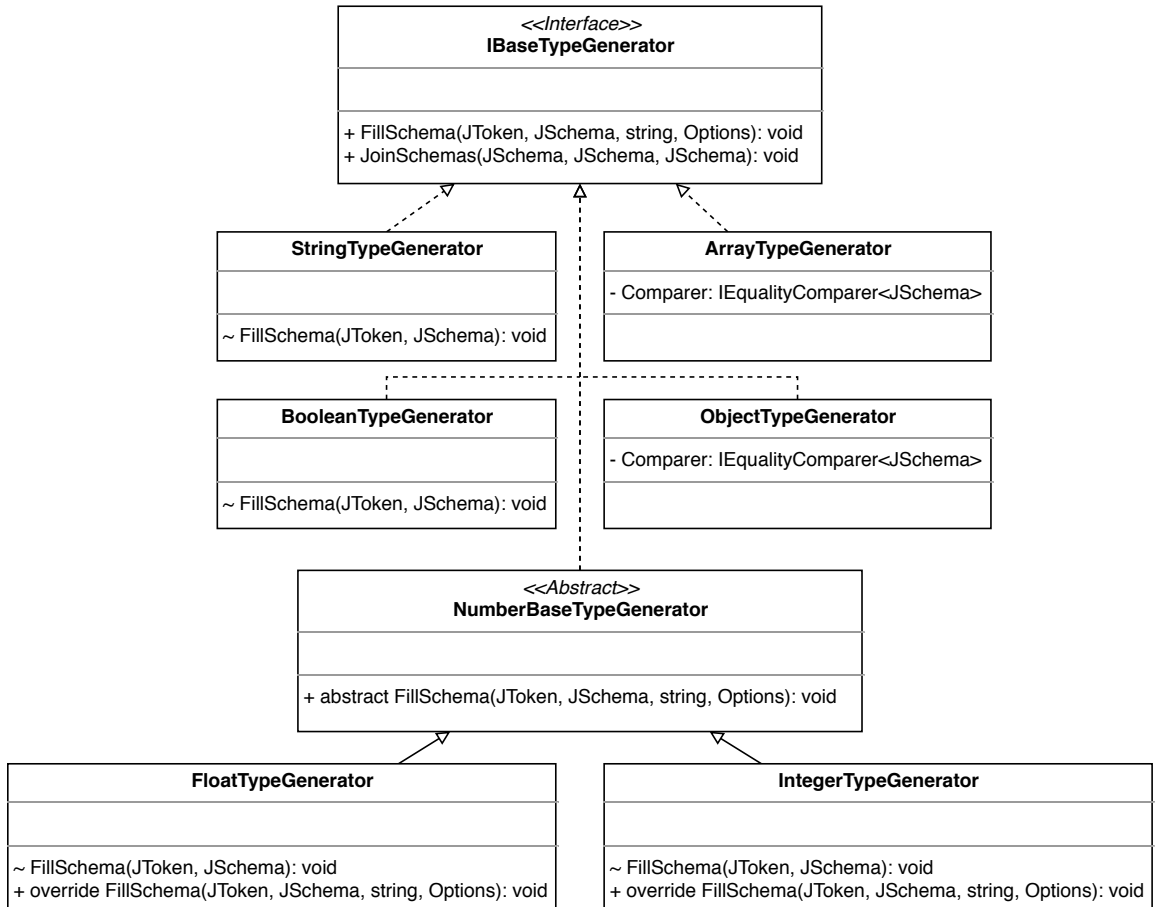


Figure 4.1: Class diagram describing different type generators.

The main logic for this resides in the `MergeSchemas` method of the `SchemaGenerator` class, which handles the whole generation process. The merging starts with the general keywords and annotations. Since the schemas can all describe different types and the resulting schema can contain multiple type-specific keywords, all type generators are used in the next step, more specifically their `JoinSchemas` method.

It is worth noting that the generators are as restrictive as possible in the generation phase. That means they use `const` keyword and strict ranges, where applicable (e.g. number types boundaries, string length, array capacity, etc.). Similarly, during joining schemas, it tries to apply the `const` keyword, if that can not be accomplished, it creates an `enum` of the two `const` values and then further expands the values. This happens only until the number of items in the `enum` exceeds a threshold, then the keyword is dropped to prevent expanding it infinitely. The threshold was set to 6 items after an agreement with the supervisor.

The user's definitions are preserved whenever possible. This applies, e.g. when joining the `items` keyword for array instances. The generators use the tuple validation, but if the user's schema uses the list validation, the result will also contain list validation.

4.2.1 Generator’s Configuration Options

The process of generating the JSON Schema can be configured by the user in certain ways (**R_Config**). Generally, this applies mainly to non-type-specific keywords and annotations, but there are a few options that are relevant only to objects and arrays.

The user can provide a URI that will be used to identify the generated schema. The value will be then included as an `$id` on the top level of the schema (**R_Config_1**). The value must be a valid URI. Otherwise, the generation will not succeed and show an error message to the user about an invalid URI.

Since the JSON Schema allows to specify the `$id` keyword in any nested level, the user can decide whether the generated schema will contain these keywords (**R_Config_2**). A simple switch is used to control this behaviour.

The presence of the `readOnly` and `writeOnly` flags is also controlled by the user’s choice (**R_Config_3**). To distinguish the value `false` of these keywords and the desire not to include them at all, check-boxes could not be used. Instead, selections with three options are available.

The application can pre-fill some of the available annotations for the user. The presence of every annotation can be toggled independently (**R_Config_4**). `Titles` and `descriptions` use a placeholder text that is meant to be replaced by the user. `Defaults` are populated by the usual default values in programming languages—an empty string, 0, false, an empty object, or an empty array, depending on the type of the value. `Examples` are populated by the actual value in the input JSON document.

The next two options are used with object types. The first specifies if all of the properties of the object from the input JSON should be included in the `required` keyword (**R_Config_5**). This is equivalent of injection if we assume the properties of the input JSON as a domain and the properties of a set of all acceptable samples as a codomain. The other option is used to allow any additional properties on top of those that are present in the object (**R_Config_6**). When disabled, this represents a surjection, using the same assumption from before.

The last option is similar to the previous one, but this time it applies to arrays (**R_Config_7**). Figure B.6 represents the modal dialog with all the options mentioned in this section.

4.2.2 Code Snippets for JSON Validators

For cases when the user would like to integrate a validator into his application, *plexSON* has a feature for generating a validation code snippet for a few programming languages. Currently, C#, Go, Java, JavaScript, PHP, and Python are supported. The code snippet serves as a template for those languages which will create a simple class with one method, accepting one string parameter (in the snippet for Python it is not a string, but a JSON value already) representing the JSON instance. The code snippets use the validators described in Section 2.3—*Json .NET Schema* for C#, *Qri* for Go, *Networknt* for Java, *AJV* for JavaScript, *Opis* for PHP, and *Jsonschema* for Python. An example of the output can be seen in Figure B.3 in appendices.

The actual implementation is very straightforward. The only important part is to escape the schema string correctly according to the syntax of the selected language.

4.3 Component for Validating Against a Schema

Outside the preparation of the parameters and error handling, the validation is completely delegated to the Json .NET Schema package. At first, the input instance and the schema are parsed to the internal representation (JToken and JSchema objects), and the validation is accomplished by a single call on the JToken object, as shown by Listing 4.1. The result and the errors are then used to form the response object.

```
bool valid = json.IsValid(schema, out IList<ValidationError> errors);
```

Listing 4.1: Example usage of the Json .NET Schema validator. The *valid* variable contains the validation result. In case of a failure, all information will be saved in the *errors* variable.

4.4 Client-Side Implementation

The client-side implementation is divided into a few parts based on the functionality they provide. Each of those parts is explained in more detail in this section. Figure 4.2 shows a class diagram of the client-side.

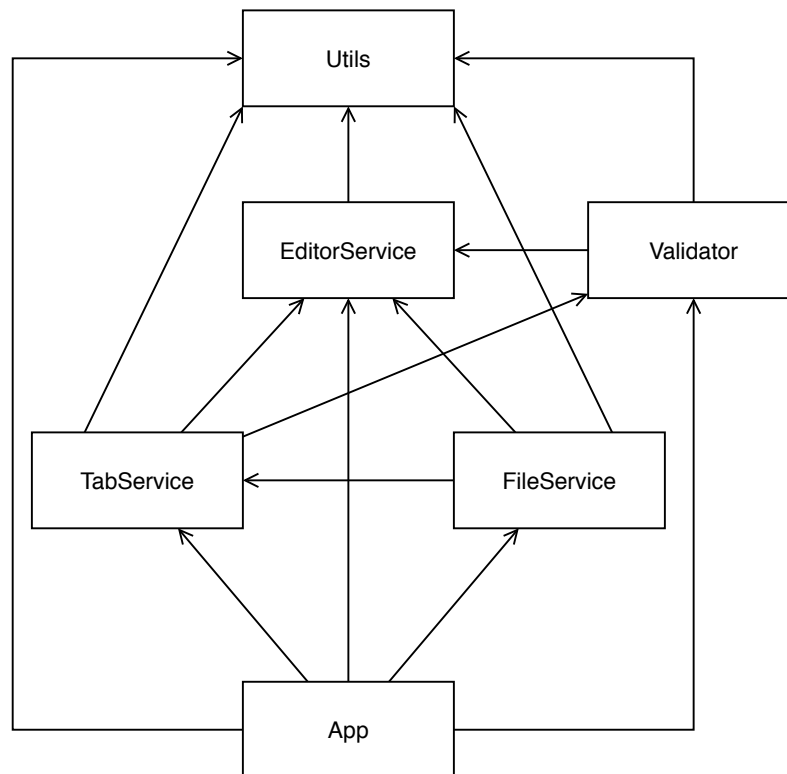


Figure 4.2: Class diagram of the client-side implementation.

Editors

Everything related to the interaction with the editor is present in the `editorservice.js` file. This includes creating the editors, manipulating with their content, but also providing an interface to other components.

The Monaco Editor is used with a concept of *models*. An editor instance is usually created with an already existing model, but it is optional. Upon creating a model, one assigns it a URI to identify it, specifies a language mode, and the initial textual content. Each language can be further configured. In case of a JSON language mode, which is used by this application, there is a possibility to allow comments, enable validation, and even specifying a JSON Schema. This feature is used to validate the content of the editor for schemas. It was originally meant to also be used in the editor for the inputs for a while, which meant there would be no need to use a third-party validator. Unfortunately, while the invalid parts are highlighted by underlining, the messages are not as user-friendly as required. It states the problem but does not mention the path in the schema where the definition is violated.

These models are then attached to the editor instance. Each model has its state, which is beneficial in e.g. preserving an undo/redo stack or the position of the cursor when the model of an editor changes. This implies that an editor's model can be changed anytime. This feature comes handy when having multiple files open in one editor, which is the case in this project. The editor for validation code snippets is read-only. Its content can not be changed by the user. Figure 4.3 depicts the EditorService class.

| EditorService |
|---|
| <ul style="list-style-type: none"> - inputEditor: IStandaloneEditor - schemaEditor: IStandaloneEditor - snippetEditor: IStandaloneEditor |
| <ul style="list-style-type: none"> - createEditor(string, ITextModel): IStandaloneEditor - onContentChanged(bool): void - setJsonDefaults(): void + createModel(string, Uri): ITextModel + clearInput(): void + clearSchema(): void + setSchemaContent(string): void + getInputModel(): ITextModel + getCurrentInput(): string + getSchemaModel(): ITextModel + getSchemaContent(): string + getSchemaMarkers(): number + setMarkers(IMarkerData[]): void + deleteAllInputMarkers(): void + getSnippetContent(): string + setSnippetContent(string, string): void |

Figure 4.3: Preview of the EditorService class.

Tabs

When using multiple input instances, each of them is represented by its own tab. In the beginning, an empty tab is already prepared, so the application is ready to be used. New tabs are created when uploading files or manually, like in web browsers. Tabs can be closed as well. After closing the last opened tab, an empty one is created. The editor can not be closed as a whole. Creating, destroying, and switching between the tabs is a responsibility of `tabservice.js`. The editor for the schema does not use tabs, as working with multiple schemas at a time is not in the scope of this work. Figure 4.4 depicts the TabService class.

| TabService |
|--|
| <ul style="list-style-type: none"> - tabId: number - tabs: object[] - editorService: EditorService - validator: Validator |
| <ul style="list-style-type: none"> + createTab(ITextModel, string): void + changeTab(HTMMLLElement, number): void + getTabModelContent(number): string + destroyTab(HTMMLLElement, number): void |

Figure 4.4: Preview of the TabService class.

Files

Various tasks related to files are implemented in `fileservice.js`. Only the files containing *json* in their `Content-Type` property are processed, others are ignored. It is important to note, that only browsers supporting the `FileReader` from the File API specification¹¹ can use this feature. All major browser support it, so it should not cause any impediments.

The files are read *asynchronously* after selecting files by the file picker (**R_Upload**) or by dragging and dropping over an editor (**R_Drop**). This will work only if the files are dropped exactly over the editors. Regarding the editor for the schema, only one file is allowed. If multiple files are attempted to open, only the first one is processed. Figure 4.5 depicts the FileService class.

| FileService |
|--|
| <ul style="list-style-type: none"> - editorService: EditorService - tabService: TabService |
| <ul style="list-style-type: none"> - saveAs(string, string, string): void - handleDragOver(Event): void - handleInputFileSelect(Event): void - handleInputSelectDrop(Event): void - setInputContent(FileList): void - handleSchemaFileSelect(Event): void - handleSchemaSelectDrop(Event): void - setSchemaContent(File): void |

Figure 4.5: Preview of the FileService class.

Validating

`Validator.js` is responsible for managing the validations. The current contents of both editors are used to form a validation request to the server. The response is processed, and the user interface changes accordingly: if the input instance is valid, a green message about successful validation will appear under the editor for inputs, but in case of any validation error, a red message is displayed, the problematic parts are underlined with yellow, wavy lines, and further details will appear after hovering the mouse over the underlined text. The details include a path to the violated schema definition. Figures B.2 and B.1 show the result upon failed validation.

¹¹<https://w3c.github.io/FileAPI/#dfn-filereader>

A similar red message is also used to inform the user when either of the editors contains an invalid JSON. In this case, the underline decorations are red.

In order to reuse relevant validation results as much as possible, the application caches them so that a new request is not initiated when not needed. This might be just switching between tabs without actually changing the contents. Each tab's validation result is cached at the end of processing the response. Figure 4.6 depicts the Validator class.

| Validator |
|--|
| - endpointUri: string - editorService: EditorService |
| + validateCurrentInput(): void + printStatusMessage(number): void + toggleSchemaMessage(bool): void - processValidationResult(object): void |

Figure 4.6: Preview of the Validator class.

Utils

The functionalities, that are relevant for multiple components, are stored in `utils.js`. It provides methods for making HTTP requests to the back-end, copying content to the clipboard, showing various notifications to the user in the form of *toast messages* which can be seen in Figure B.4, and also manipulating with the mentioned cache. The meta-schema¹² for the JSON schemas (JSON Schema draft-07 version—**R_Version**), which is used to validate the content of the editor for the schema, is also stored here. Figure 4.7 depicts the Utils class.

| Utils |
|---|
| + needsValidation: bool + validationCache: object + metaSchema: object |
| + postData(RequestInfo, object): Promise + showToastMessage(string, string): void + createToast(string, string): void + copyToClipboard(string): void + getFromCache(string): object undefined + saveToCache(string, number): void + deleteCache(string): void + invalidateCache(): void |

Figure 4.7: Preview of the Utils class.

App

The starting point of the actual client-side logic resides in `app.js`. The initialization of, e.g. tool-tips over the buttons is here, along with registering the event handlers. The validation requests are fired periodically every two seconds if the contents have changed

¹²<https://json-schema.org/specification.html#meta-schemas>

since the last validation. Similarly, it prepares the generation request and handles the response (updating the content in the editor for the schema or showing an error message if anything goes wrong). Figure 4.8 depicts the App class.

| App |
|--|
| - untitleCount: number - editorService: EditorService - fileService: FileService - validator: Validator - tabService: TabService |
| - addToSchema(): void - newTab(): void - generateSnippet(): void + Start(): void |

Figure 4.8: Preview of the App class.

Chapter 5

Evaluation of Implemented Solution

Verification of an application's functionality is an integral part of any software development cycle. Before the development of the production code, automated unit-tests were prepared to quickly verify the final product's functionality during the development. After the first prototype was finished, end-to-end tests were added to verify the overall functionality (**R_Tests**). The application passed both of the W3.org's markup¹ and style² validators successfully.

5.1 Unit Tests

Because of the low complexity of the server-side component's architecture, the unit tests were chosen to assure the verification. Several tests were created to test both endpoint URLs. There are a total of 26 unit tests, and they belong to a separate project. As a testing framework was chosen NUnit³. Listing 5.1 shows a simple test usage with this framework. The tests can be run in different ways. The easiest might be running it from the command-line with the `dotnet test` command followed by the project or solution containing the tests. During the development, it is useful to run them directly from an IDE like Visual Studio with an extension like ReSharper⁴ providing a test runner. The NUnit also provides a console runner.

```
using NUnit.Framework;

namespace Test.Project
{
    [TestFixture]
    public class Tests
    {
        [Test]
        public void Test()
        {
```

¹<https://validator.w3.org/nu/>

²<https://jigsaw.w3.org/css-validator/>

³<https://nunit.org/>

⁴<https://www.jetbrains.com/resharper/>

```
        Assert.AreEqual(4, 2 + 2);
    }
}
}
```

Listing 5.1: Writing unit test with NUnit 3.

These tests cover **R_Inputs** and **R_Config**.

5.2 End-To-End Tests

To verify the overall functionality of the application, a suite of end-to-end (E2E) tests was created. Similarly to the unit tests, they were developed using the NUnit framework. Additionally, Selenium⁵ was chosen as browser automation project with WebDriver⁶ implementations for Google Chrome (ChromeDriver⁷), Mozilla Firefox (GeckoDriver⁸, Mozilla Public License), Opera (OperaDriver⁹), and the new Microsoft Edge (Microsoft Edge Driver¹⁰), all under BSD-3-Clause License, except for GeckoDriver. Figure 3.3 served also another purpose than its original one: it was used as a source for defining a test suite for E2E testing. All browsers passed the test suites. For specific versions tested, refer to Table 5.1 in the next section.

During E2E testing, which consisted of a total of 36 test cases, several minor defects in the application were discovered and addressed. The problems resided in wrongly leaving action buttons allowed when they should be disabled. These tests also discovered a slightly different interpretation of some JavaScript features by different browsers, like static class fields.

The E2E test category covers **R_Validate** and **R_Editors**.

5.3 Exploratory Testing

On top of the automated test suites, manual exploratory testing was required for verification of the part of the application’s workflow, mainly the use-cases tied with the interaction with files. The main reason was the lack of possibility to test the uploading of files. This also included functionality of the tabs. The configuration was also tested manually due to the nature of the editor’s implementation since it does not provide easy access to its content for automated testing.

Manual testing was required to cover **R_Error**, **R_Syntax**, **R_Invalid**, **R_Inputs**, **R_Upload**, and **R_Drop**. The test scenarios included:

- uploading and dropping JSON file(s)—successful loading in editors,
- uploading and dropping non-JSON file(s)—ignored, skipped their loading,
- downloading the editors’ contents—successful saving with correct file name and content,

⁵<https://www.selenium.dev/documentation/en/>

⁶<https://w3c.github.io/webdriver/>

⁷<https://chromedriver.chromium.org/>

⁸<https://github.com/mozilla/geckodriver>

⁹<https://github.com/operasoftware/operachromiumdriver>

¹⁰<https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/>

- copying the editors' contents to the clipboard,
- creating, switching, and closing tabs of the editor for inputs—predictable switching to another tab when closing the currently active one,
- appearance and automatic hiding of toast messages after timeout,
- displaying and stacking of multiple toast messages in a natural manner,
- marking an invalid or not corresponding part of the JSON document/schema,
- appearance and content of the error messages displayed upon hovering mouse over an invalid or not corresponding part of the JSON document/schema, and
- generating code snippets for JSON validators for the current schema

5.3.1 Demonstration of the Application's Functionality

An artificial use cases demonstrating the functionality of the application were performed and the results are saved in plain text files. Each of these files represent a single use case. These files contain a short description of the given example, the generation request, i.e. all input instances and the whole configuration, and the resulting schema as the output. The results are saved in the `examples` folder (see the project folder structure in Appendix A).

5.4 Compatibility Testing

The application was tested on the following browsers with their versions as depicted by Table 5.1.

| Browser | Version |
|---------------------------|----------------|
| Google Chrome | 81 |
| Mozilla Firefox | 75 |
| Opera | 67 |
| Microsoft Edge (Chromium) | 81 |

Table 5.1: Verified browsers and tested version.

Microsoft Internet Explorer can not be used as it does not implement JavaScript classes, which are used in the application. Similarly, the legacy Microsoft Edge browser, using the EdgeHTML engine, does not support class fields and thus does not provide enough features for the application to function properly.

The application was developed on the Windows platform, while **R_Multiplatform** was verified by running the application inside a Linux container.

Chapter 6

Conclusion

The goal of this thesis was to design and implement a single-page application for manipulation with JSON schemas, including generating them and validating JSON documents. After researching existing solutions with a similar purpose, the application fills a gap in the market. The innovative feature is the ability to produce a JSON Schema from multiple JSON instances at once while supporting the draft version 07 of the JSON Schema.

The project was implemented with portability in mind, which influenced the selection of the technologies to use. The most important operations—that are validating and generating—are implemented as web API endpoints. This allows for easy replacement of the presentation layer, should it be required in the future. Also, the endpoints can be used by external tools and programs, e.g. to validate incoming JSON data. The web interface takes advantage of the powerful Monaco editor, known from the most popular and free VS Code environment.

The functionality of the application was verified throughout the development by automated tests of multiple levels, like the unit and end-to-end tests. Additionally, manual interaction was performed with different browsers to ensure compatibility and the same behaviour across them. Sample outputs of the applications are included to demonstrate the functionality, see included storage media.

To further develop and enrich the application, several possible enhancements come to mind. These could include reverse inferring of a sample JSON document from the schema, to quickly illustrate its structure by an example. Another useful feature could be extending the supported versions of the JSON Schema draft, and the eventual standard, should it be released. The desired version would be selected as part of the configuration.

6.1 Unfinished Functionality

In the final stages of the semester, I started to develop a feature beyond the assignment which would allow the user to select and load publicly available JSON Schemas from the schema store¹. Unfortunately, it could not be finished due to certificate issue on the schema store's website and the lack of time to solve or workaround it. Because of this, plexSON's JavaScript code was not able to fetch the schemas. Nevertheless, the code is present, but commented-out. The preview of this feature can be found in Figure B.5.

¹<http://schemastore.org/json/>

Bibliography

- [1] ALBAHARI, J. and ALBAHARI, B. *C# 7.0 in a Nutshell*. 7th ed. O'Reilly Media, Inc., 2017. ISBN 978-1-491-98765-0.
- [2] APIGEE CORP. *Web API Design: The Missing Link* [online]. Google, LLC, 2018 [cit. 2020-04-04]. Available at: <https://pages.apigee.com/rs/351-WXY-166/images/Web-design-the-missing-link-ebook-2016-11.pdf>.
- [3] BRAY, T. *The JavaScript Object Notation (JSON) Data Interchange Format* [Internet Requests for Comments]. RFC 7159. RFC Editor, March 2014 [cit. 2020-06-02]. <http://www.rfc-editor.org/rfc/rfc7159.txt>. Available at: <http://www.rfc-editor.org/rfc/rfc7159.txt>.
- [4] CHAND, B. *Difference between Client-Server and Peer-to-Peer Network* [online]. 2019 [cit. 2020-04-04]. Available at: <https://bimalchand.com.np/difference-between-client-server-and-peer-to-peer-network/>.
- [5] DROETTBOOM, M. *Understanding JSON schema* [online]. 2020 [cit. 2020-03-17]. Available at: <https://json-schema.org/understanding-json-schema/index.html>.
- [6] FIELDING, R. and RESCHKE, J. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing* [Internet Requests for Comments]. RFC 7230. RFC Editor, June 2014 [cit. 2020-04-04]. Available at: <http://www.rfc-editor.org/rfc/rfc7230.txt>.
- [7] FIELDING, R. and RESCHKE, J. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* [Internet Requests for Comments]. RFC 7231. RFC Editor, June 2014 [cit. 2020-04-04]. Available at: <http://www.rfc-editor.org/rfc/rfc7231.txt>.
- [8] FLANAGAN, D. *JavaScript: The Definitive Guide*. 6th ed. O'Reilly Media, Inc., 2011. ISBN 978-0-596-80552-4.
- [9] JACOBSON, D., BRAIL, G. and WOODS, D. *APIs: A Strategy Guide*. 1st ed. O'Reilly Media, Inc., 2011. ISBN 978-1-449-30892-6.
- [10] LANDER, R. *Announcing .NET Core 1.0* [online]. Microsoft, june 2016 [cit. 2020-03-13]. Available at: <https://devblogs.microsoft.com/dotnet/announcing-net-core-1-0/>.
- [11] LANDER, R. *Introducing .NET 5* [online]. Microsoft, may 2019 [cit. 2020-03-13]. Available at: <https://devblogs.microsoft.com/dotnet/introducing-net-5/>.
- [12] LIE, H. W. *Cascading HTML style sheets – a proposal* [online]. 1994 [cit. 2020-04-12]. Available at: <https://www.w3.org/People/howcome/p/cascade.html>.

- [13] LUBBERS, P., ALBERS, B. and SALIM, F. *Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development*. 1st ed. Apress L. P., 2010. ISBN 978-1-430-22790-8.
- [14] MICROSOFT CORPORATION. *Monaco Editor* [online]. 2020 [cit. 2020-03-13]. Available at: <https://microsoft.github.io/monaco-editor/index.html>.
- [15] MOZILLA CONTRIBUTORS. *About JavaScript* [online]. 2020 [cit. 2020-03-13]. Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript.
- [16] MOZILLA CONTRIBUTORS. *Fetch API* [online]. 2020 [cit. 2020-04-12]. Available at: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.
- [17] RAUSCHMAYER, A. *Speaking JavaScript*. 1st ed. O'Reilly Media, Inc., 2014. ISBN 978-1-449-36503-5.
- [18] WOOTTON, J. *JSONschema.Net* [online]. 2020 [cit. 2020-04-01]. Available at: <https://jsonschema.net/home>.
- [19] WRIGHT, A. and ANDREWS, H. *JSON Schema: A Media Type for Describing JSON Documents* [Working Draft]. Internet-Draft draft-handrews-json-schema-01. IETF Secretariat, March 2018 [cit. 2020-06-02]. Available at: <http://www.ietf.org/internet-drafts/draft-handrews-json-schema-01.txt>.
- [20] WRIGHT, A., ANDREWS, H. and LUFF, G. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON* [Working Draft]. Internet-Draft draft-handrews-json-schema-validation-01. IETF Secretariat, March 2018 [cit. 2020-06-02]. Available at: <http://www.ietf.org/internet-drafts/draft-handrews-json-schema-validation-01.txt>.

Appendix A

Contents of the Included Storage Media

The attached CD contains the following folder structure:

- `examples/` - reports of the artificial use cases demonstrating the application's functionality
- `src/`
 - `PlexSON.API/` - project containing the server-side implementation
 - `css/` - contains the style sheet definitions
 - `js/` - client-side implementation and jQuery minified source
 - `monaco-editor/` - sources of the Monaco editor component
 - `index.html` - mark-up of the web page
- `out/` - compiled, executable computer program
- `thesis/` - source files of the thesis text with figures and assignment
- `tests/` - project containing automated tests
- `xfujac00-plexson.pdf` - Text of the technical report

Refer to the `README.md` for instructions on how to build, run, test, deploy, and use PlexSON.

Appendix B

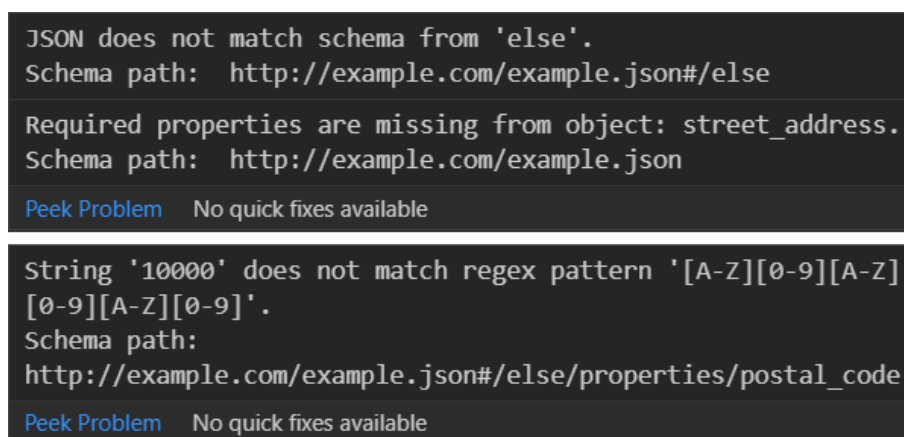
Screenshots of the Web Application

Figure B.2 is a preview of the final visual appearance of the application. This specific screenshot was chosen because it presents the most features on a single screen. All actions are possible to execute at this point.

The details of validation failures are presented in Figure B.1. Hovering the mouse on the yellow-underlined text in the editor displays the messages.

An example of a validation code snippet for JavaScript language can be seen in Figure B.3.

Even though the feature for loading and using schemas from the schema store is not part of the final version, here is a preview of how it would look like. Figure B.5 captures the top-right corner of the page, where above the editor for the schema would be a select-box with public schemas. The default *Custom schema* option would be automatically selected upon any manipulation with the schema editor not to mislead the users that they still have the original schema loaded.



```
JSON does not match schema from 'else'.  
Schema path: http://example.com/example.json#/else  
  
Required properties are missing from object: street_address.  
Schema path: http://example.com/example.json  
  
Peek Problem No quick fixes available
```

```
String '10000' does not match regex pattern '[A-Z][0-9][A-Z][0-9][A-Z][0-9]'.  
Schema path:  
http://example.com/example.json#/else/properties/postal_code  
  
Peek Problem No quick fixes available
```

Figure B.1: Sample details of each validation error from Figure B.2.

plexSON ?

Input JSON

```
example.json x Untitled-1 x +
1 {
2   "country": "Canada",
3   "postal_code": "10000"
4 }
```

..or choose file(s) Browse

x Your JSON is invalid against the schema.

✂ 🗑️ ⬇️ ▶️

JSON Schema

```
1 {
2   "$schema": "http://json-schema.org/draft-07/schema",
3   "type": "object",
4   "properties": {
5     "street_address": {
6       "type": "string"
7     },
8     "country": {
9       "enum": ["United States of America", "Canada"]
10    }
11  },
12  "if": {
13    "properties": { "country": { "const": "United States of
14  }},
15  "then": {
16    "properties": { "postal_code": { "pattern": "[0-9]{5}(-[
17  }},
18  "else": {
19    "properties": { "postal_code": { "pattern": "[A-Z][0-9][
20  }
21 }
```

..or choose a file Browse

🗑️ 📄 ⬇️ 📄

Figure B.2: The final layout of the application with an example of a failed validation.

Validation snippet ×

Python ↻

```

1 # $ pip install jsonschema
2 from jsonschema import Draft7Validator
3
4 class Validator:
5     def __init__(self):
6         self.schema = {"$schema": "http://json-schema.org/draft-07/schema", "type": "array"}
7         try:
8             Draft7Validator.check_schema(self.schema)
9         except jsonschema.exceptions.SchemaError:
10            print('Invalid schema')
11            raise
12
13     def validate(self, json):
14         v = Draft7Validator(self.schema)
15         errors = []
16         for error in v.iter_errors(json):
17             errors.append(error)
18         valid = len(errors) == 0
19         return self.ValidationResult(valid, errors)
20
21 class ValidationResult:
22     def __init__(self, valid, errors):
23         self.valid = valid
24         self.errors = errors

```

📄 ⬇

Figure B.3: Preview of the validation snippet modal dialog for Python.

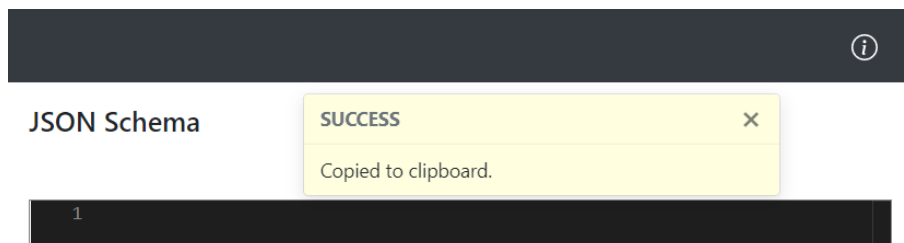


Figure B.4: Example of a notification toast message.

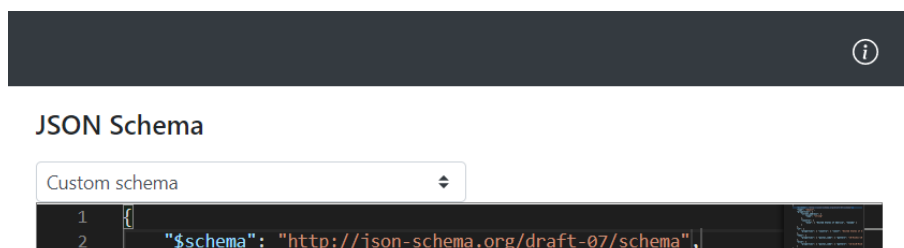


Figure B.5: Preview of the unfinished schema store support.

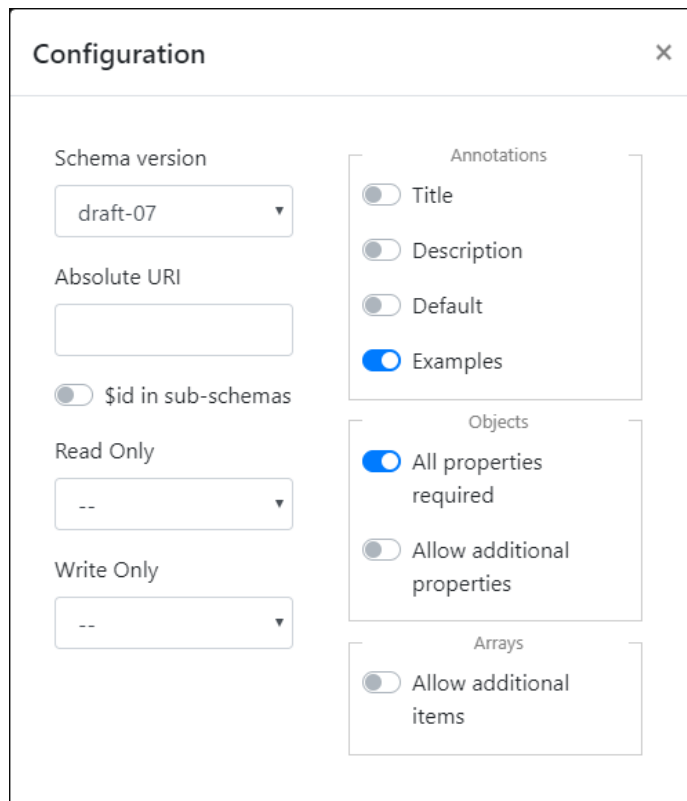


Figure B.6: Modal dialog with configuration options.