



**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

MASTER THESIS

Bc. Jan Janda

**CSV file validator according to the CSV
on the Web W3C recommendations**

Department of Software Engineering

Supervisor of the master thesis: RNDr. Jakub Klímek, Ph.D.

Study programme: Computer Science – Software and
Data Engineering (N0613A140015)

Study branch: ISDP (0613TA140015)

Prague 2024

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In date

Author's signature

I would like to thank my kind supervisor RNDr. Jakub Klímek, Ph.D. for his feedback and countless remarks. His supervision elevated this thesis and guided its progress.

Title: CSV file validator according to the CSV on the Web W3C recommendations

Author: Bc. Jan Janda

Department: Department of Software Engineering

Supervisor: RNDr. Jakub Klímek, Ph.D., Department of Software Engineering

Abstract: The comma-separated values (CSV) format is a popular format for tabular data on the web. It stores tables of data in a very simple textual way, but it stores only the values of the table without their meaning and structure. Users of a table must remove the ambiguity and guess the exact meaning of data. The standard called CSV on the Web provides recommendations for metadata about CSV tables on the web. The metadata can describe a CSV table, its structure, and the meaning of its values in the JSON-LD format. The standard allows a creator of a table to remove ambiguity, increase certainty, and create confidence. There are CSV tables together with their JSON-LD metadata descriptions, and each table should match its particular description. In this thesis, we develop a CSV file validator. The validator is a computer program which checks whether tables actually match their metadata descriptions. It reads a table with its description, compares them, and searches for discrepancies. This text describes the development process and its result.

Keywords: CSV JSON-LD W3C web validator OTAVA

Název práce: Validátor CSV souborů dle W3C doporučení CSV on the Web

Autor: Bc. Jan Janda

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Jakub Klímek, Ph.D., Katedra softwarového inženýrství

Abstrakt: Formát comma-separated values (CSV) je populárním formátem pro tabulková data na webu. Tento formát ukládá tabulky dat velmi jednoduchým textovým způsobem, ale ukládá jen hodnoty tabulky bez jejich významu a struktury. Uživatelé tabulky musí odstranit nejednoznačnost a hádat přesný význam dat. Standard CSV on the Web poskytuje doporučení pro metadata o CSV tabulkách na webu. Metadata mohou popsat CSV tabulku, její strukturu a význam jejích hodnot v JSON-LD formátu. Tento standard umožňuje tvůrci tabulky odstranit nejednoznačnost, zvýšit jistotu a vytvořit důvěru. Máme CSV tabulky spolu s jejich metadatovými popisy v JSON-LD a každá tabulka by se měla shodovat se svým konkrétním popisem. V této práci vyvíjíme validátor CSV souborů. Tento validátor je počítačový program, který zkontroluje, zda se tabulky skutečně shodují se svými popisy. Validátor přečte tabulku s jejím popisem, porovná je a vyhledá nesrovnalosti. Tento text popisuje proces vývoje a jeho výsledek.

Klíčová slova: CSV JSON-LD W3C web validátor OTAVA

Contents

1	Introduction	3
2	Standards and specifications	4
2.1	Open data	4
2.2	Resource Description Framework	5
2.2.1	Turtle	6
2.2.2	JSON-LD	8
2.3	CSV on the Web	11
2.3.1	Data model	13
2.3.2	Transformations	14
3	Analysis	17
3.1	The goal of this thesis	17
3.1.1	Anticipated users	17
3.1.2	Functional requirements	18
3.1.3	Quality requirements	20
3.2	Existing solutions	21
3.2.1	csvw-validator	22
3.2.2	RDF::Tabular	23
3.2.3	CSV Lint	23
3.2.4	Evaluation of the solutions	25
4	Design	28
4.1	Library	28
4.1.1	Validation checks	28
4.1.2	Documents	29
4.1.3	Manager	30
4.1.4	Summary	32
4.2	Command line interface	32
4.3	Web application	32
4.3.1	Client	33
4.3.2	Server	35
4.3.3	Database	36
4.3.4	Worker	36
4.3.5	Summary	37
4.4	Summary	37
5	Implementation	38
5.1	Structure of the project	38
5.2	Library	38
5.2.1	Validation checks	39
5.2.2	Implemented checks	40
5.2.3	Documents	44
5.2.4	Manager	46
5.2.5	Tests	49

5.3	Command line interface	49
5.4	Web application	51
5.4.1	Client	51
5.4.2	Server	53
5.4.3	Database	53
5.4.4	Worker	54
5.4.5	Containers	55
5.5	Summary	55
6	Documentation	56
6.1	Library	56
6.1.1	Validation principles	56
6.1.2	Documents	57
6.1.3	Dependency	57
6.1.4	Validation extension	57
6.2	Command line application	58
6.2.1	Examples	59
6.3	Web application	59
7	Evaluation	61
7.1	Assignment	61
7.2	Parts of the project	61
7.2.1	Command line interface	61
7.2.2	Web application	64
7.2.3	Summary	66
7.3	Requirements	66
7.4	Comparison	67
7.5	Performance	68
7.6	Summary	72
8	Conclusion	73
	Bibliography	74
	List of Figures	78
	List of Tables	79
A	Attachments	80
A.1	Project repository	80
B	Example of JSON-LD 1.1	81
C	Web application diagram	82
D	Server in Openapi	89
E	W3C tests	93

1. Introduction

This thesis exists in the world of data and data formats. This domain contains problems and solutions related to data in the purest form. People seek new and better ways to store data and preserve its meaning. Our effort focuses on the popular comma-separated values (CSV) format. This format is well known and widespread on the web, but it has its flaws. It is used to store tables of data in a very simple textual way. The problem is that it stores only the values of the table, and we can easily lose the meaning of data and connections with the real world. The meaning of the stored data is in the context and possible improvised descriptions. This way does not preserve the meaning very well, and users of the data must guess the exact meaning. It can be a serious problem in practical applications.

Fortunately, there is a technical standard with a solution to this problem. The standard consists of CSV on the Web W3C recommendations, and it provides a way to preserve meaning of data and to store metadata about a CSV table. The standard uses the resource description framework (RDF) and the JSON-LD format to store the metadata. It was created by professionals at the World Wide Web Consortium (W3C) in accordance with the best practices of the modern web. It is very extensive and capable high-quality standard. It can describe many details of a CSV table. It can prescribe the structure of the table and disambiguate the meaning of values with RDF. This universally accepted standard significantly enhances quality of data in CSV tables published on the web.

We have a CSV table and its JSON-LD metadata description according to the standard. It is only natural that we want to make sure that the table actually matches the description. We need a computer program which can validate tables with their descriptions and tell us whether they match each other. This thesis satisfies this need and creates a CSV file validator according to the CSV on the Web W3C recommendations. This is the goal of this thesis, and this thesis describes the complete process of software development. There are already some other similar solutions and implementations, but they are insufficiently maintained and hard to use. We gain inspiration and experience from the earlier implementations, and then we develop our own solution to the validation problem in this thesis.

This text gradually guides readers through the entire process. We start with the standards and specifications because they provide a theoretical and technological background. Then, we analyse our requirements and existing similar solutions. This part precisely defines the goal of this thesis in relation to the real world. The software design follows the analysis. The design is a skeleton of our software, and its structure allows us to systematically fulfil the requirements. The implementation brings the design to life. It shows used technologies, structure of the source code, solutions of technical problems, etc. There is a concise description of the project in the documentation. The documentation helps users in different roles to use the developed software. Finally, we evaluate our solution. We show how we fulfilled the requirements and the assignment. We also measure the performance of our solution, and then we conclude our thesis.

2. Standards and specifications

Standardization is an important part of modern world. It allows us to communicate and cooperate with people around the globe, it increases efficiency and removes barriers. This thesis is heavily influenced by several important worldwide standards and specifications. This chapter presents them.

2.1 Open data

Open data is data that is freely accessible, editable, exploitable and shared. It is particularly utilized in relation with governments and public administrations in general. Freedom of information is in some form legislated in many civilized democratic countries. There are regulations in the European Union [8], the United States of America [22] and the Czech Republic [30]. Open data provide a useful tool to technically achieve this freedom. It is obvious that this freedom is essential for every free and prosperous society. There are many groups of enthusiasts and experts working tirelessly in the world of open data. One major group is the Open Data Institute [24]. It is a non-profit company, founded in 2012 by Sir Tim Berners-Lee and Sir Nigel Shadbolt. They provide comprehensive information about the vision, the beliefs and the manifesto of open data. It is a good source of ideas behind open data, but it is not the only one, and everything should be treated as a recommendation.

There are many noble principles, let us see a few of them. The mission is to work with companies and governments to build an open, trustworthy data ecosystem. Sectors and societies must invest in and protect the data infrastructure they rely on. Everyone must have the opportunity to understand how data can be and is being used. We need data literacy for all, data science skills, and experience using data to help solve problems. Data must inspire and fuel innovation. Everyone must benefit fairly from data. People and organisations must use data ethically. Everyone must be able to take part in making data work for us all. This summarizes the ideals of open data.

There are also many practical implementations of open data. They usually take the form of a catalogue presented as a web page. There are catalogues in the United Kingdom [13], the Czech Republic [29] and the United States of America [46]. The general definition of open data is rather vague, but it gains specific details in each implementation. These catalogues provide links to particular sources of data within agencies and organizations. The data can be provided in a format with various levels of quality.

It is important to provide data with sufficient quality. The quality is measured with the 5-star rating [2]. Every star represents a desirable quality of data. The first star means that the data is available on the web under an open licence. It is a basic foundation for future progress. It is simple to publish and everyone can look at it. The second star means that the data is available as structured data. It can be directly processed with proprietary software. The third star means that the data is available in a non-proprietary open format. It can be manipulated without the need to own any proprietary software package. The fourth star means that there are uniform resource identifiers (URIs) to denote things. The data can

be safely combined. URIs are a global scheme so if two things have the same URI then it is intentional, and if so that is well on its way to being 5-star data. Finally, the fifth star means that there are links to other data to provide context. A reader can discover more related data while consuming the data. This is simple yet powerful way to measure quality of (open) data.

2.2 Resource Description Framework

The top quality from the open data 5-star rating is also called linked open data or linked data. The links significantly enhance accessibility and connect data to form a web of data. The Resource Description Framework (RDF) [55] is a practical representation of linked data. We use the version RDF 1.1 unless stated otherwise. It is a framework for representing information in the Web. It consists of RDF graphs and RDF datasets. RDF is a graph-based data model. There are actual graphs, and datasets are used to organize collections of graphs. Each dataset contains one default graph and zero or more named graphs. RDF is defined by its abstract syntax (also known as a data model).

The core structure of the abstract syntax is a set of triples. A set of triples is an RDF graph. Each triple consists of a subject, a predicate and an object. The meaning of a triple is obvious. Some relationship, indicated by the predicate, holds between the subject and the object. Figure 2.1 shows an example of a triple. There are three possible kinds of nodes in a triple: an Internationalized Resource Identifier (IRI) [10], a literal, and a blank node. Any IRI or literal can denote everything. They can represent every real-world thing and every abstract concept. IRI is a special string used as a global identifier and literal is a literal value with a datatype such as number, string or date. Blank node is a node without a global identifier. It may have only a local identifier which is valid only within its document. It is merely an auxiliary structure and it is used to group other things together. Subject can be an IRI or a blank node. Predicate can be an IRI. Object can be an IRI, a literal or a blank node.

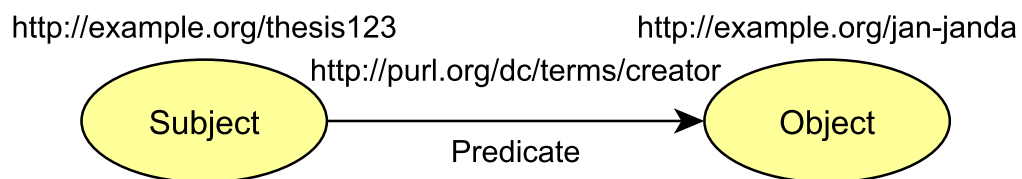


Figure 2.1: Example of an RDF triple

Triples, graphs and datasets comprise fundamental concepts of RDF, but there are some more details. A thing denoted by an IRI or a literal is called resource. Formally, we should differentiate between a resource and its corresponding IRI or literal. A triple represents a so-called RDF statement. We should again differentiate between a statement and its representation. The predicate IRI denotes a property. It is a resource that can be thought of as a binary relation. RDF utilizes vocabularies. An RDF vocabulary is a collection of IRIs intended for use

in RDF graphs. The triple in Figure 2.1 has a predicate from the vocabulary DCMI Metadata Terms [6]. A literal consists of two or three elements. It has a value encoded as a Unicode [44] string, a datatype IRI and possibly a language tag.

Naturally, we do not use pictures of graphs for storing and transferring RDF datasets. There are several serializations that define a textual syntax for RDF. Each one has its advantages and disadvantages. We show two of them because one is perfect for human-readable examples and the other is technically used in this thesis.

2.2.1 Turtle

RDF 1.1 Turtle [1] is a straightforward textual syntax. We simply call it Turtle. It allows an RDF graph to be completely written in a compact and natural text form. It is best to see examples. The triple in Figure 2.1 is written as follows.

```
<http://example.org/thesis123> <http://purl.org/dc/terms/creator>
<http://example.org/jan-janda> .
```

It may be written on one line if there is enough space. The number of lines is insignificant. There is a simple sequence of subject, predicate and object terms separated by whitespace and it is terminated by a full stop "." character. IRIs are enclosed in angle brackets.

We can define prefixes in order to shorten IRIs. Here we have the same statement with prefixes.

```
@prefix ex: <http://example.org/> .
@prefix dcterms: <http://purl.org/dc/terms/> .

ex:thesis123 dcterms:creator ex:jan-janda .
```

There is a recommendation for prefixes of well-known IRIs. It recommends a particular prefix for each well-known IRI. These IRIs usually correspond to popular vocabularies. The IRIs in an RDF vocabulary often begin with a common substring known as a namespace IRI. Some namespace IRIs are associated by convention with a namespace prefix. Namespace IRIs and namespace prefixes are not a formal part of the RDF data model. They are merely a syntactic convenience for abbreviating IRIs. It is a convention, not a rule. We do not have to follow it and we can define prefixes however we want. We mostly follow the conventions in this thesis. There is an online service which maps common prefixes onto their recommended IRIs. It is Prefix.cc [7]. Let us assume that all the following examples have the necessary prefixes defined. This is only for brevity. We use common prefixes and the meaning of each IRI is fairly self-explanatory. We can find the prefixes with Prefix.cc. These examples are not supposed to explain the exact meaning of prefixes.

Let us add some properties with literal values. This example shows a datatype with a datatype IRI and language tags. The literals with a language tags have an implicit datatype `rdf:langString`. The order of triples is insignificant because it is a set. A literal without any datatype or language tag has an implicit

datatype `xsd:string`. Such literal is only enclosed in quotation marks like so "some literal". Turtle has a shorthand syntax for writing integer values, arbitrary precision decimal values, double precision floating point values and boolean values. For example `"-5"^^xsd:integer` can be written as `-5`. These features are not shown in this example for the sake of brevity.

```
ex:thesis123 dcterms:creator ex:jan-janda .
ex:thesis123 dcterms:dateSubmitted "2021-07-20"^^xsd:date .
ex:thesis123 dcterms:title "Data monitoring"@en .
ex:thesis123 dcterms:title "Monitorování dat"@cs .
ex:thesis123 rdf:type schema:Thesis .
```

We can clearly see the unnecessary repetition of some terms. It will be easier to read and understand if we remove the redundant terms. Turtle provides a useful syntax for that. We can rearrange the triples for better clarity because their order is not important.

```
ex:thesis123 a schema:Thesis ;
             dcterms:creator ex:jan-janda ;
             dcterms:dateSubmitted "2021-07-20"^^xsd:date ;
             dcterms:title "Data monitoring"@en ,
                          "Monitorování dat"@cs .
```

The predicate `rdf:type` is now represented by the token `a`. This predicate is very common and it means that the subject is an instance of a class in object. It represents the relation "is a". The thesis123 *is a* thesis. We use a semicolon ";" if we want to implicitly repeat the subject in the next statement. We use a comma "," if we want to implicitly repeat the subject and the predicate.

Finally, let us see a blank node. A blank node lacks an identifier. We have to create an auxiliary identifier. Turtle provides simple syntax with such identifiers. Figure 2.2 shows an RDF graph with a blank node and the following examples represent the same graph in Turtle.

```
ex:jan-janda ex:hasCar _:b0 .
_:b0 ex:color ex:blue .
_:b0 ex:brand ex:vw .
_:b0 ex:used-in ex:czechia .
```

There is another syntax without ad hoc identifiers. It uses square brackets to denote blank nodes.

```
ex:jan-janda ex:hasCar [
  ex:color ex:blue ;
  ex:brand ex:vw ;
  ex:used-in ex:czechia
] .
```

This way, we do not have to figure out unnecessary identifiers. Brackets enclose properties of a blank node.

We introduced every important aspect of Turtle. There are more advanced details, but we do not need them in our thesis. It is important to understand

the basics because Turtle is probably the most human-friendly RDF serialization. Turtle is useful for showing RDF graphs to humans. It might be even better than the pictures of graphs.

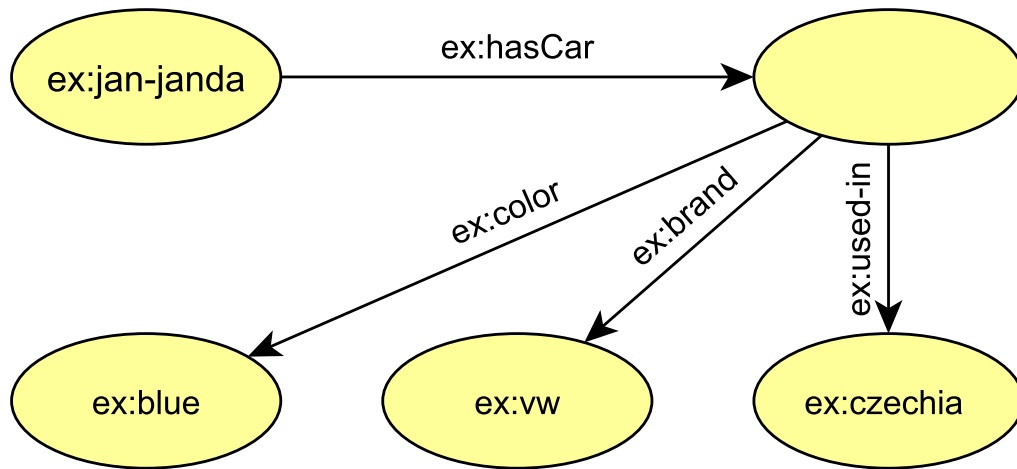


Figure 2.2: Example of an RDF graph with a blank node

2.2.2 JSON-LD

JSON-LD 1.1 [36] is a more technical textual syntax. We simply call it JSON-LD. JavaScript Object Notation (JSON) [5] is a lightweight data-interchange format. It is relatively easy for humans to read and write. It is easy for machines to parse and generate. JSON comes from JavaScript. This is probably the main reason why it is so popular on the Web. It is also widely supported outside of the Web. JSON-LD is a JSON-based serialization for linked data. It harnesses the popularity of JSON for linked data and RDF. It is completely compatible with JSON. This is an advantage because users that do not care about linked data can still use the data as a plain JSON. JSON-LD sensitively introduces new features to JSON.

Most of the necessary information is added to a JSON file in the form of a so-called context. The context is used to map terms onto IRIs. Terms are case sensitive, and most valid strings that are not reserved JSON-LD keywords can be used as a term. A context is introduced using an entry with a special keyword. The following example shows the triple in Figure 2.1.

```
{
  "@context": {
    "creator": {
      "@id": "http://purl.org/dc/terms/creator",
      "@type": "@id"
    }
  },
  "@id": "http://example.org/thesis123",
  "creator": "http://example.org/jan-janda"
}
```

We recommend playing on the JSON-LD Playground [16]. It is a good tool, and it helped us perfect the examples. JSON-LD looks more complex compared to Turtle. We can see its syntax. There is an entry with the key `@context`, and it specifies the meaning of the other keys in the document. It specifies the meaning of the key `creator` like so. First, it says that `creator` is a shorthand for `http://purl.org/dc/terms/creator`. It also says that a string value associated with the key `creator` should be interpreted as an identifier that is an IRI. Once we know this, we understand the main part of the document after the context. There is a subject identified by the entry with the key `@id`. It has a well-defined predicate `creator`, and the object is the value associated with the predicate. The reserved keywords tend to start with `@`.

Let us see a more comprehensive example. It shows all the important features of JSON-LD. The document is divided into smaller parts for clarity, but the parts are actually inseparable. We have the context first.

```
"@context": {
  "ex": "http://example.org/",
  "dcterms": "http://purl.org/dc/terms/",
  "schema": "http://schema.org/",
  "submitted": {
    "@id": "dcterms:dateSubmitted",
    "@type": "http://www.w3.org/2001/XMLSchema#date"
  },
  "changed": {
    "@id": "dcterms:modified",
    "@type": "http://www.w3.org/2001/XMLSchema#date"
  },
  "title": {
    "@id": "dcterms:title",
    "@container": "@language"
  }
}
```

We can now write the actual data. There is an identifier of the described thing. We have prefixes defined in the context, and we use them here instead of full IRIs. The keyword `@type` denotes the property `rdf:type`. It is similar to the token `a` in Turtle. The property `submitted` is defined in the context, and it is defined as an instance of the `xsd:date` datatype.

```
"@id": "ex:thesis123",
"@type": "schema:Thesis",
"submitted": "2021-07-20"
```

This part looks like this in Turtle. Turtle is not a part of the JSON-LD document.

```
ex:thesis123 a schema:Thesis ;
  dcterms:dateSubmitted "2021-07-20"^^xsd:date .
```

There is a property with multiple values. The exact meaning and datatype is defined in the context. Even though the values are in an array, the order is not preserved. It is treated as a set without any order.

```
"changed": [ "2021-07-18", "2021-07-15", "2021-07-10" ]
```

This part looks like this in Turtle. Turtle is not a part of the JSON-LD document.

```
ex:thesis123 dcterms:modified "2021-07-10"^^xsd:date ,  
  "2021-07-18"^^xsd:date , "2021-07-15"^^xsd:date .
```

We can embed other statements into properties and build deeper graphs. The embedded thing is represented by a blank node because it does not have `@id`.

```
"ex:reader": {  
  "ex:rank": "professor",  
  "ex:name": "John Doe"  
}
```

This part looks like this in Turtle. Turtle is not a part of the JSON-LD document.

```
ex:thesis123 ex:reader [  
  ex:name "John Doe" ;  
  ex:rank "professor"  
] .
```

The key `title` is defined as a language container in the context. It is used like so. There are multiple values for different languages.

```
"title": {  
  "en": "Data monitoring",  
  "cs": "Monitorování dat"  
}
```

This part looks like this in Turtle. Turtle is not a part of the JSON-LD document.

```
ex:thesis123 dcterms:title "Data monitoring"@en ,  
  "Monitorování dat"@cs .
```

We can use more verbose syntax instead of a language container. It has the same effect.

```
"dcterms:format": [  
  {  
    "@value": "paper",  
    "@language": "en"  
  },  
  {  
    "@value": "papír",  
    "@language": "cs"  
  }  
]
```

This part looks like this in Turtle. Turtle is not a part of the JSON-LD document.

```
ex:thesis123 dcterms:format "paper"@en , "papír"@cs .
```

Some literals do not need quotation marks and they have implicit datatypes according to their values.

```
"ex:ready": true,  
"ex:copies": 3
```

This part looks like this in Turtle. Turtle is not a part of the JSON-LD document.

```
ex:thesis123 ex:ready true ;  
ex:copies 3 .
```

That is the end of this example. We assume that the necessary prefixes are defined for the Turtle snippets in accordance with the JSON-LD example. This example is in the appendix B. It is not divided into parts there.

Context does not have to be in a document itself. It can be only linked.

```
"@context": "http://example.org/contexts/thesis.jsonld"
```

We can combine external and local contexts. The following example specifies an external context and then layers an embedded context on top of the external context.

```
"@context": [  
  "https://json-ld.org/contexts/person.jsonld",  
  {  
    "pic": {  
      "@id": "http://xmlns.com/foaf/0.1/depiction",  
      "@type": "@id"  
    }  
  }  
]
```

Finally, there is a way to preserve the order of values. We need to use the keyword `@list` in a right place. A property can be defined as a list container in the context. It is similar to the language container. A property can be defined as a list outside of context as well.

The examples and their descriptions gradually introduced the standard with its important aspects. Let us conclude our description of JSON-LD by saying that there are many more advanced features, but we do not need them in this thesis.

2.3 CSV on the Web

The comma-separated values (CSV) [34] file format is a well known, simple and quite popular format for tabular data. It is both human-readable as well as machine-readable and it can be easily processed in software without any need for

special parsing techniques or complex external libraries. It is also supported by many spreadsheet programs. A CSV file is a text file with a special structure to record a table of data. Each row of the table is recorded in the file as a line of text, and individual cells in a row are separated by the comma character. The first line of the file may be a header. It means that it does not contain actual data, but it records names of the columns of the table. This straightforward structure is probably the main reason for the wide use of this format.

The format emerged spontaneously during the early days of general purpose electronic computers. Its long and unguided history resulted in some more or less compatible variants. The most notable differences are the choice of separators, line breaks and text encoding. Alternative separators include semicolon and space. These separators can define a more general delimiter-separated values (DSV) format. Various DSV formats are out of the scope of this thesis. Special characters used to mark end of lines depend on the platform. Windows operating system works with the combination CR LF and Linux distributions use LF from the ASCII table [53]. Various encodings are a more general problem with text files. Fortunately, the UTF-8 encoding [54] almost completely solves this issue. These variations constitute different CSV dialects. In this thesis, we focus on the standard CSV with consideration for some compatible deviations. We want to uphold the best standards and therefore we do not focus on different dialects. Even though there are relatively recent attempts for standardization, different variants are still being used and implemented.

Despite these problems, CSV gained popularity in the world of open data. We can see why the CSV file format is so popular. We get three stars of the 5-star rating simply by publishing data in the CSV format. That is definitely a good start, but how do we get the other two stars? Aside from the already mentioned problems, there are more problems to overcome. These problems are not technical, they relate to semantic of data. A CSV file does not contain any information about its content. So-called metadata should contain important information about data types in columns, identifiers of described things etc. There is a standard for such metadata. It is known as CSV on the Web (CSVW) [52] and it was developed by the World Wide Web Consortium (W3C) [3].

The standard allows us to provide extra information about CSV files using a JSON metadata file. The description of a table within a metadata file is very versatile. It can include documentation about the CSV file as a whole (description, authorship or licensing information), a definition of the structure of the CSV file, and instructions that enable processors to transform the CSV file into other formats, such as JSON or RDF. It utilizes the JSON-LD standard to record the information. The following example shows a simple metadata file.

```
{
  "@context": "http://www.w3.org/ns/csvw",
  "url": "countries.csv",
  "tableSchema": {
    "columns": [{
      "titles": "country"
    }, {
      "titles": "continent"
    }]
  }
}
```

```
}  
}
```

We can easily recognize the structure of a JSON-LD document. There is an extensive external context. It contains all the necessary definitions for a description of a CSV file. There is a Uniform Resource Locator (URL) of the described CSV file, and there is a schema of the table. The schema defines the columns of the table. The property `columns` is defined as a list container in the context. It means that the order of the columns is preserved. Our example may describe this CSV file.

```
"country", "continent"  
"France", "Europe"  
"Canada", "America"
```

Metadata files can also describe several CSV files at once, like so.

```
{  
  "@context": "http://www.w3.org/ns/csvw",  
  "tables": [{  
    "url": "countries.csv"  
  }, {  
    "url": "organizations.csv"  
  }]  
}
```

The examples show the fundamental structure of the metadata. It is just a skeleton. Many properties can be added to each entity. We can precisely describe every table, schema, column, etc. Some properties, such as title or description, are meant for humans. Other properties, such as datatype or keys, are meant for machines. The metadata disambiguates the true meaning of the CSV file. The CSV on the Web standard provides truly versatile tools for description of CSV files, and it allows us to achieve all five stars of the 5-star rating.

2.3.1 Data model

Let us dive into the CSV on the Web data model [37] and see its details. The data model is a model for tables that are annotated with metadata. The model provides annotations at several levels. We introduce these levels one after another. Higher levels are more general, and lower levels are more specific. There are required and optional properties at each level. These properties are also specified by the Metadata Vocabulary for Tabular Data [31].

First of all, there are top-level properties. The top-level object of a metadata document must have a `@context` property. The property must have the specified IRI which references the necessary external context. We can see the mandatory IRI in the example above. Only this particular IRI can be used. The external context may be combined with a local context. The local context may add a base URL and a default language.

A table group is at the highest level of actual data just below the context. A group of tables comprises a set of annotated tables. It contains a set of annotations that relate to that group of tables. It must contain a property with the tables in the group. A table is at the next level and it may be annotated with many properties such as its URL, columns, and foreign keys. Finally, there are last three levels, i.e. column, row, and cell in this order. These entities represent and annotate the respective structures in a table. We can specify every conceivable property of these structures, e.g. name, property URL, titles, and value.

The metadata and annotations are very useful for several reasons. We focus on validation. Validation checks a CSV file against its JSON-LD metadata file in order to determine whether the CSV file actually fulfils the requirements prescribed by the metadata. We can check the number of columns and their names, what kind of values columns contain, whether any of those values are unique, and whether they match values in other CSV files.

We usually want to limit the set of values that may appear in a column. This can be done with a datatype annotation. Every column can have a specified datatype. We simply add the following property to a column object. In this example, we expect every value in the column to be a number.

```
"datatype": "number"
```

There are many predefined datatypes and we can modify them to achieve greater specificity, like so.

```
"datatype": {  
  "base": "integer",  
  "minimum": "1",  
  "maximum": "5"  
}
```

The syntax is obvious. We have predefined datatype as a base and then we add new properties and restrictions to it. The predefined datatypes are based on the set defined in the XML Schema specification [49]. This is a very powerful tool, and it uses intuitive keywords. We can specify shape of string values, size of numbers, mix of types, etc. We can say whether values are required in a column, and we can set default values. We can define keys within a table and between tables. It is possible to add many machine-readable properties and specify a structure of a table. If we define these properties, a computer should be able to check a CSV file against them.

2.3.2 Transformations

We know that CSV is a very useful and popular format, but it may still be necessary to transform the data into the RDF format. The annotations and metadata can help us with the transformation. This process is described in detail in its own specification [51]. We introduce the minimal mode transformation and show some related annotations because it is an important application of the CSV on the Web recommendation. We contributed to the project Dataspecer [45]

before this thesis, and we implemented similar transformations in that project. We show the RDF in Turtle.

The following example illustrates the transformation. We have a simple inventory of furniture in CSV.

```
"furniture","number"  
"chair","12"  
"table","3"
```

The transformation is very crude without any metadata. The result looks like this.

```
[ <#furniture> "chair" ;  
  <#number> "12" ] .  
[ <#furniture> "table" ;  
  <#number> "3" ] .
```

There is a blank node for each line of data. The predicates are relative IRIs constructed from the header line, and the objects are the values from respective lines.

Let us add a JSON-LD metadata file. We show only the part of the metadata file with the list of columns for brevity.

```
"columns": [{  
  "titles": "furniture",  
  "lang": "en"  
},{  
  "titles": "number",  
  "datatype": "integer"  
}]
```

This file provides valuable information about the CSV file. We can use it to enhance the result. The languages and datatypes are represented in the result.

```
[ <#furniture> "chair"@en ;  
  <#number> 12 ] .  
[ <#furniture> "table"@en ;  
  <#number> 3 ] .
```

It is good to define some IRIs according to the principles of linked data. It helps us remove ambiguity. Let us modify the metadata file.

```
"aboutUrl": "http://example.org/my-furniture/{designation}",  
"columns": [{  
  "titles": "furniture",  
  "name": "designation",  
  "propertyUrl": "http://example.org/furniture-type",  
  "valueUrl": "http://example.org/{designation}"  
},{  
  "titles": "number",  
  "propertyUrl": "http://example.org/number",  
  "datatype": "integer"  
}]
```

There are new properties. The property `aboutUrl` is a URI template property that indicates what a cell contains information about. It uses the name of the column as a variable in curly brackets and it belongs to a table schema next to columns. The property `name` gives a single canonical name for the column. The property `propertyUrl` creates a URI for the property in the column. The property `valueUrl` is a URI template property that is used to map the values of cells onto URLs. The result of the transformation looks much better now.

```
@prefix ex: <http://example.org/> .
@prefix my: <http://example.org/my-furniture/> .

my:chair ex:furniture-type ex:chair ;
  ex:number 12 .
my:table ex:furniture-type ex:table ;
  ex:number 3 .
```

We provided proper IRIs and removed the blank nodes. The result of the transformation looks like a good RDF and the semantic meaning is preserved.

It is worth mentioning that a CSV file can be transformed into JSON format as well. It is also possible to enhance the transformation with the metadata according to the standard [50]. The result of the transformation is created in a similar way. There are some differences because JSON does not support all the features of RDF. We do not introduce it because it is not relevant to this thesis.

This section contains an introduction and important details of the CSV on the Web recommendation. It is probably the most important standard in this thesis. We omitted specific and advanced details because they are not important enough. It is possible that such details will be necessary later in this thesis. If it happens, the details will be explained on the spot. This rule applies to every section of this chapter.

3. Analysis

In this chapter, we analyse everything important for the context of this thesis. We use already developed standards which guide this thesis. We set the goal and analyse requirements. We present existing solutions, and we gain experience from them.

3.1 The goal of this thesis

This chapter contains analysis of anticipated users, possible use-cases and requirements. We analyse in gradually increasing detail. Everything originates from the goal of this thesis. We endeavour to create a useful application to enable CSV file validation according to the CSV on the Web W3C recommendations. The validation is a process which checks whether a table matches its description according to the standards. This determines the purpose of the application. At the beginning of this chapter, we have anticipated users. It is the most coarse grained part. Individual users represent large groups of requirements. We continue with a finer grained analysis until we reach and derive the most specific requirements. These requirements are then directly implemented and verified. They constitute the skeleton of the application and its development.

3.1.1 Anticipated users

We recognize three kinds of users on the basis of their skill level. Every kind has a set of requirements. We assume that the requirements of a less skilled user are a subset of the requirement of a more skilled user. Users are organized in a way to emphasise this property. Figure 3.1 shows the anticipated users and their relations. More demanding users inherit some requirements and needs from less demanding users. This subsection presents each requirement as if they were stated by users themselves. They have broader and less specific meaning.

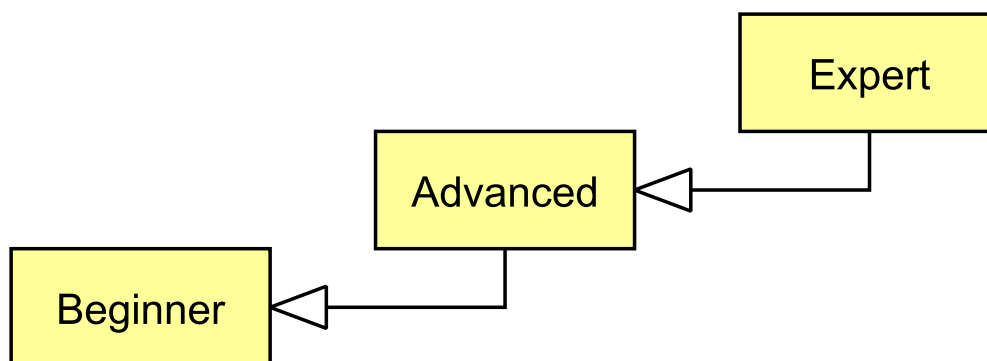


Figure 3.1: Diagram of the anticipated users

The first user is a random person who needs to occasionally validate a CSV file, i.e. a beginner. This kind of user just want to get his job done with minimal

effort and see a clear answer. It means that a prolonged installation of the application is very undesirable. The best way is to use the application directly from a web browser without any preparation. The graphical user interface should take the form of an uncluttered web page containing few control elements with clear and intuitive meaning. The result of the validation should be equally simple and readable. The user expects an output with apt answers.

Another user may be a clerk of a public administration who uses the application regularly, i.e. an advanced user. This user is more knowledgeable, but still enjoys the convenience of web application. This kind of user wants to configure the application in a limited way and can read and understand more complex results. The configuration and results should not clutter the interface, but they should remain easily accessible. It might be useful to see both the used configuration and the generated result. The application should respond without unnecessary delay and keep its user informed about the process of the validation.

Finally, we have an expert. This kind of user utilizes full potential of the application, uses advanced configuration and understands detailed output. The interface must provide full control over validation, and detailed output must contain every available piece of information. In case of intensive and advanced use, it might be better to avoid web interface and to use the core functionality directly and locally on user's computer via a command line interface.

Even though all users desire to perform virtually the same task, their conception of necessary tools can be very different. It is important for us to satisfy all groups of potential users and to develop an application which fulfils its role at multiple levels of complexity.

3.1.2 Functional requirements

In this subsection, we formally state our functional requirements. They form a skeleton of the application. We show how we fulfilled them later in this thesis.

It must be possible to

- (F1) provide a local path of a CSV file,
- (F2) provide a local path of a JSON-LD metadata file,
- (F3) provide URL of a CSV file on the internet,
- (F4) provide URL of a JSON-LD metadata file on the internet,
- (F5) provide all files together,
- (F6) provide only some files,
- (F7) tell the application to find the required files,
- (F8) explicitly provide and override the required files,
- (F9) explicitly assign tables to their descriptions,
- (F10) validate a stand-alone CSV file,

- (F11) validate a stand-alone JSON-LD metadata file,
- (F12) validate all files with each other,
- (F13) view results of the validation in plain text,
- (F14) view results of the validation in Turtle, and
- (F15) view results of the validation in JSON.

We briefly describe different kinds of validation in order to better understand the functional requirements. The simplest kind validates only one file. It checks only syntax and elementary structure of a file. It works with CSV as well as JSON-LD, but the file is always checked separately and individually without any other file. Then we have a more complicated kind. There is a CSV file and a JSON-LD file. The JSON-LD file describes the CSV file and the validation checks whether the CSV file complies with the description. The most complex kind validates multiple CSV files. There are JSON-LD files that describe the tables. They describe multiple tables in separate CSV files. The validation checks whether each CSV file complies with its description. It can also check links, references and foreign keys between tables. Some files may be missing, e.g. JSON-LD metadata file for a CSV file or one of the described CSV files. We can tell the application to find them or we can provide them manually. The application can assign the corresponding files to each other, but we can override it and assign files manually to each other. This flexibility can be useful in some very advanced cases.

Figure 3.2 shows a use case diagram of the system. The diagram provides a visual summary of the functional requirements. We present only two kinds of users for simplicity. There is a beginner who uses only the simple features of the system. The elementary functionality can be used without deep comprehension of the subject. The simple form of results is intuitive and easy to understand. There is also an advanced user who is capable of utilizing every part of the system. Naturally, an advanced user can do everything that a beginner can do. We simplified the diagram and omitted these redundant obvious lines in the picture. The advanced features are easily available to an advanced user and they usually include the elementary features. The individual use cases are described in the following list.

(UC1) Provide CSV file

A user provides a CSV file by entering its path or URL. This use case contains the functional requirements F1 and F3.

(UC2) Provide metadata file

A user provides a JSON-LD metadata file by entering its path or URL. This use case contains the functional requirements F2 and F4.

(UC3) Validate CSV file

A provided stand-alone CSV file is validated. This elementary validation is usually the first step in more complex validations. This use case contains the functional requirement F10.

(UC4) Validate metadata file

A provided stand-alone JSON-LD metadata file is validated. This elementary validation is usually the first step in more complex validations. This use case contains the functional requirement F11.

(UC5) Validate all files with each other

All provided files are validated together and with each other. This is a more complex type of validation. It starts with the validation of each separate file. This use case contains the functional requirement F12.

(UC6) Override assignment

This is very advanced use case. The default behavior of the application can be flexibly adjusted, and files can be manually assigned to each other. This use case also represents other possible advanced settings. This use case contains the functional requirement F8 and F9.

(UC7) View plain text results

A simple plain text results of the validation can be easily accessed. It provides human-readable answers. This use case contains the functional requirement F13.

(UC8) View results in technical format

Results can be viewed in selected technical format. The application supports Turtle and JSON. This use case contains the functional requirements F14 and F15.

3.1.3 Quality requirements

The application fulfils its tasks with appropriate level of quality. The following attributes represent desired qualities. Quality requirements are also known as non-functional requirements.

(Q1) The core functionality is accessible via a command line interface.

(Q2) The application is accessible from a web browser with graphical user interface.

(Q3) The graphical user interface can be viewed in English.

(Q4) The graphical user interface can be viewed in Czech.

(Q5) There is a simple way of adding another language to the graphical user interface.

(Q6) The graphical user interface provides a simplified interface.

(Q7) The core of the application is made in Java.

(Q8) The application is built upon open source technologies.

(Q9) The application passes a reasonable subset of the tests prescribed by the World Wide Web Consortium for this kind of software. The tests are the validation tests from the CSVW Implementation Report [17].

- (Q10) The application is tested automatically as well as manually.
- (Q11) The constraints and features of the application are well organized in the source code and can be easily extended and modified.
- (Q12) Core part of the program can be used as a library in another program.
- (Q13) The software has an open source license.

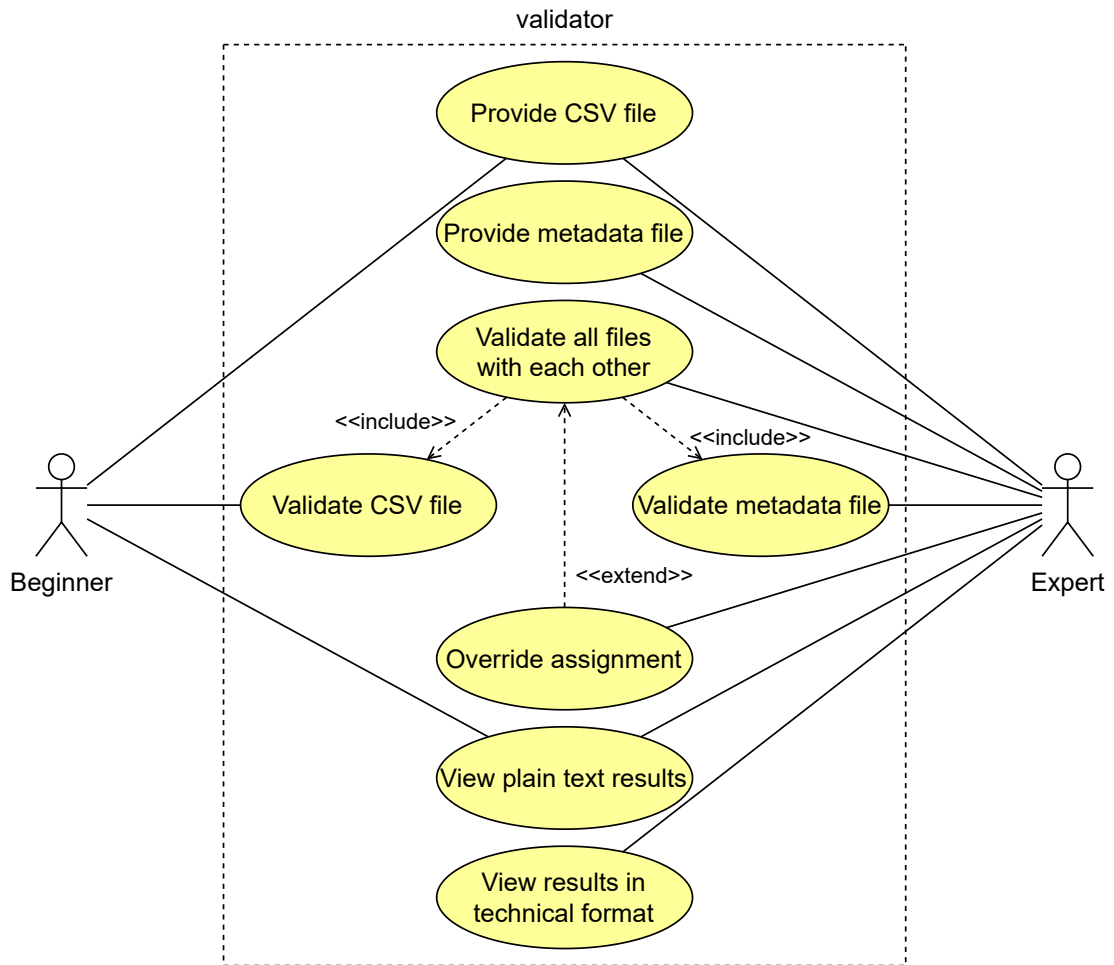


Figure 3.2: Use case diagram of the system

3.2 Existing solutions

There are already several existing solutions which solve similar problems with varying degrees of success. They validate CSV files with JSON-LD metadata files. This section shows their advantages and disadvantages. We study the solutions in order to acquire important knowledge and experience. Some solutions are better and some are worse. We learn from their mistakes and we admire their achievements. We compare the solutions and show the results of our criteria and rigorous tests.

3.2.1 csvw-validator

The csvw-validator [19] is probably the best existing solution with the most extensive documentation. It is able to parse and validate tabular data and metadata. It can work with published online files and local files. It validates CSV files and JSON-LD metadata files separately or together. The validator is accessible via three different kinds of user interface. We have a command line interface, a web service and a web application. Figure 3.3 shows the web application. Even though the interface is in both Czech and English, the documentation is in Czech only.

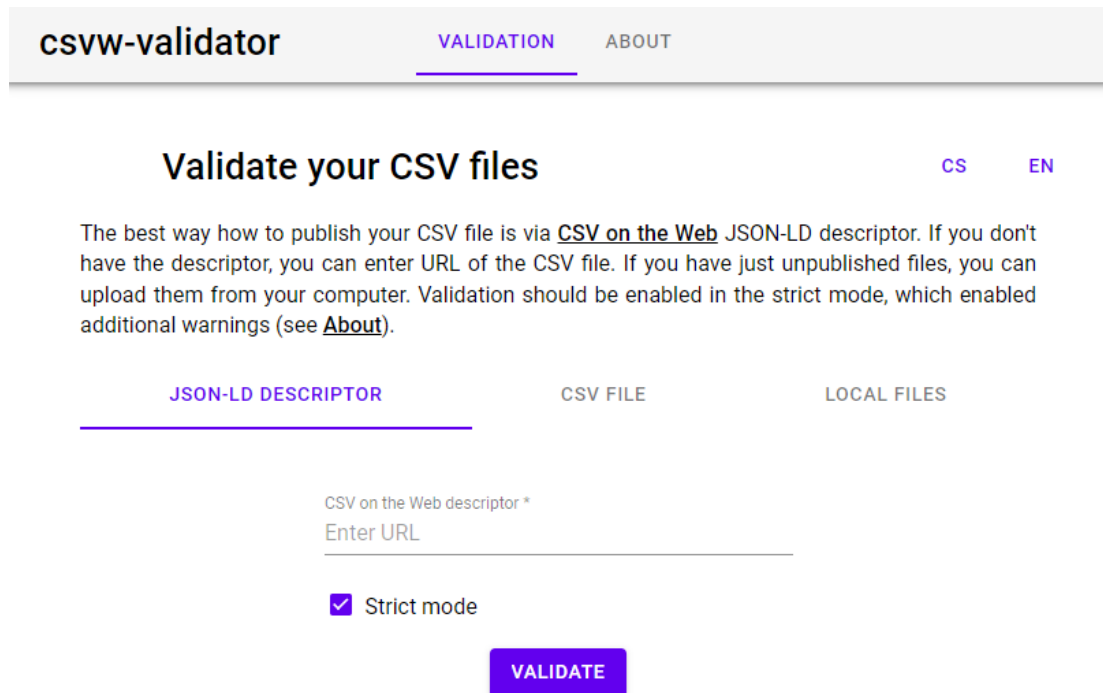


Figure 3.3: Web application interface of the csvw-validator

The web application provides the easiest way to use the validator. We may enter either URL of a JSON-LD descriptor or URL of a CSV file, but not both. The file is validated and a related file might be found by an algorithm. We cannot find any way to override this algorithm and provide both URLs at the same time. It might be useful if we want to force the application to validate a particular pair of published online files. We may also upload a local CSV file and a local JSON-LD descriptor together or separately. Naturally, the validation is more powerful if we provide both files. There is a simple configuration. The validation can be strict. Results of a validation are presented as a web page and we can download them in CSV and RDF format.

There are other ways of using this application. They are less friendly and more professional. The software can be installed locally and accessed via a command line interface. It can be used as a library within some other program. It can be used remotely as a web service via a REST interface. This web service provides endpoints for communication and a user can send request and receive results.

The validator was developed in Java 8 with several libraries. Some libraries were used purely for convenience and some were used for principal functionalities.

We mention some of the libraries and we consider using them later in this thesis. Spring Boot [48] was used to simplify the deployment. It allowed the author to create a self-contained server application with embedded server in Java. The implementation stores data in H2 Database Engine [25] because it is a simple and fast Java SQL database with easy integration. Vaadin [47] was used to create the web user interface. Finally, Apache Maven [38] is a standard tool in this software for building and managing projects. It seems that there are only integrated Java technologies.

The system avoids monolithic architecture and separates functionalities into separate packages. The packages are then grouped into modules. The system mostly follows the single-responsibility principle, but the author admits that the principle is deliberately violated in some packages. The author made good use of design patterns and hid the intrinsic complexity behind a facade. There are several components, e.g. metadata parser, schema locator and model validator.

Overall structure and functionality is well thought-out. The web application was deployed on a server and is practically used. This validator is described very well and it provides an inspiration for this thesis.

3.2.2 RDF::Tabular

RDF::Tabular [18] is another related solution. It parses CSVs, metadata files in JSON-LD and potentially other tabular data formats according to the rules defined for CSV on the Web. It can find a metadata file automatically and a user can provide a metadata file explicitly using a specific link header.

RDF::Tabular was created in the Ruby programming language and therefore it is distributed as a RubyGem. The installation is easy on Linux. We install the `gem` program and then we use it to install RDF::Tabular and its dependencies. We locate the `rdf` executable which acts as a wrapper to perform a number of different operations on RDF files using available readers and writers, including RDF::Tabular. This executable is available if the related RubyGem `linkeddata` is installed. We can not only validate a CSV file and its metadata, but also transform it into Turtle and JSON.

This is a useful application for specialists. It is not user-friendly because of the lack of a graphical user interface. The quality of the software is decent and it is adequately documented. The author of the application is also a co-author of the related W3C recommendations.

3.2.3 CSV Lint

CSV Lint [14] is another application for CSV validation. It provides a support for validation of CSV files with their syntax and contents. It can be used as a library within some other program or as a standalone command line application. There are several capabilities. There is a check of the structural formatting of a CSV file, a validation of a delimiter-separated values file accessible via URL, file, or an IO-style object. It can validate against CSV dialects and against multiple schema standards, including CSV on the Web and so-called JSON Table Schema. The JSON Table Schema is another inferior schema for tabular metadata, and it is outside the scope of this thesis. The project lacks a comprehensive

documentation. It only provides a brief summary in the repository.

CSV Lint was developed in Ruby and it is distributed as a RubyGem. The installation of the package (gem) should be easy, but we encountered difficulties with the process. It required some indispensable development tools as a dependency. It seems that the developers strive to simplify the installation with Docker in a new release. The command line interface accepts a CSV file, a schema file and parameters. The library supports validating data against a schema. The structure currently follows JSON Table Schema with some extensions and rudimentary CSV on the Web Metadata. The result of the validation consists of errors, warnings and information messages. A validation is successful if there are no errors. The output is quite readable and it provides a description and a context.

This solution is a decent application for professionals who can overcome the problems and deal with a command line interface, but it is rather hostile towards inexperienced users. Luckily for such users, there is a web-based graphical user interface called CSV Lint (or CSV Lint IO) [23] and there is an alpha version available online. Figure 3.4 shows the web application. The interface provides simplified and more intuitive way to achieve the same goal. A user may enter the location of a CSV file, or upload it if it is not on the web. A schema may be added in the same way. The interface visually presents errors, warnings and information messages. The service was not functional when we were writing this thesis. The web pages contain instructions with useful information. It includes a manual and a description of the service. It also shows available dialects and it lists common issues with CSVs. There is also an application programming interface (API) that allows a user to post a file and get the result of validation. This is a hybrid approach. It requires a command line interface, but the application does not have to be installed.

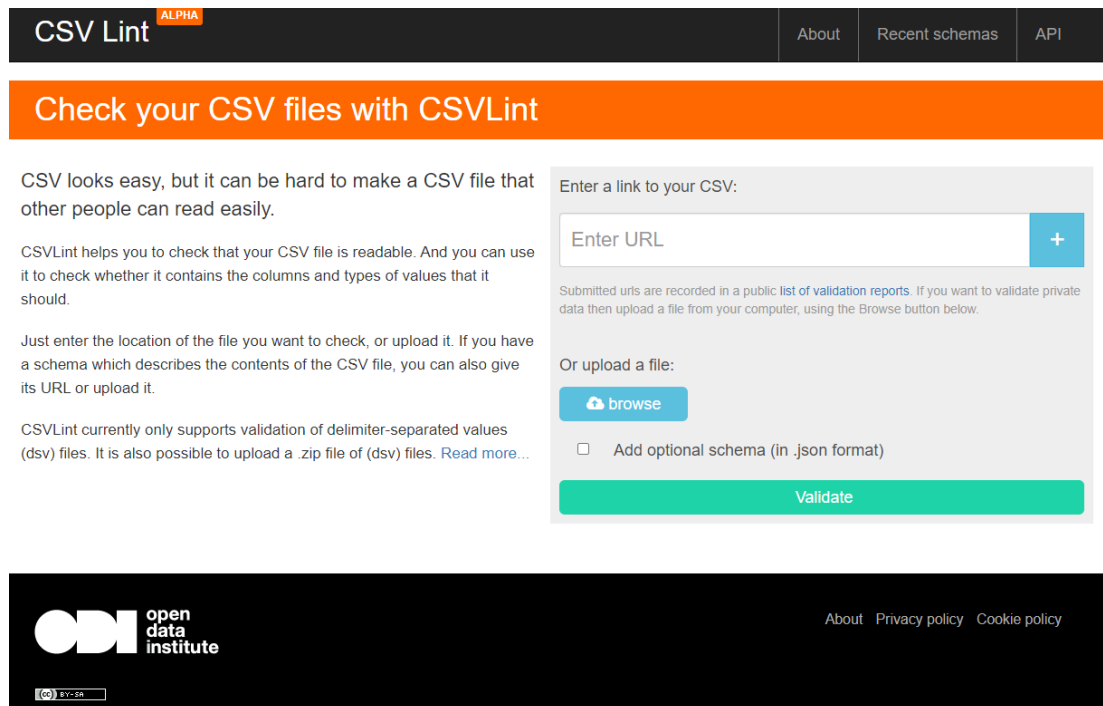


Figure 3.4: Web application interface of the CSV Lint IO

All things considered, the solution is very ambitious, and it is comprised of many parts. Each part provides a specific way of using the application. It may appeal to a large group of users. Unfortunately, the software has marks of chaotic development and poor maintenance. There is not proper documentation. It focuses on a mediocre tabular schema standard and it misses the full potential of the CSV on the Web standard.

3.2.4 Evaluation of the solutions

We presented three unique solutions to the problem of the validation. Each solution contains interesting ideas and inspiration as well as some weaknesses. There are different approaches to user interaction. We observed various stacks of technologies and ways to develop the final product. We have certain feelings about each solution, but we also need a more objective way of evaluating and testing the software. There is a test suite provided by W3C in the CSVW Implementation Report [17]. The suite is a fairly good criterion for evaluation, but the tests are not well balanced with common use cases. The tests do not cover all aspects of the solutions equally. They are not perfect, and their value should not be overstated. There are 281 validation tests in section 2.3 of the report. Table 3.1 shows the aspects of the solutions. It also contains the result of the test suite. In conclusion, the solutions have good qualities and they provide us with a valuable knowledge, but there is still an opportunity for significant improvements. We take the opportunity to write this thesis.

The main contribution of this thesis lies in a collection of details from the other solutions. We aspire to combine successful features of the other solutions and avoid their failures. We also add our own original features. We offer a lighter Java application without heavy Ruby packages and troublesome dependencies. Smart design invites collaborators to use parts of the program in various ways. Our solution supports the well-known and widely accepted standard CSV on the Web. It provides a simplified graphical user interface with clear separation of user skill levels. The interface does not impose superfluous constraints and allows a user to override default behaviour. This enhances the user experience and increases productivity. We keep in mind fast-paced and stressful environment of businesses, organizations and public administrations. Our application makes lives of our users easier and helps the world to improve quality of tabular data.

We noticed certain common problems in the existing solutions. They provide only the main functionality with little or no configuration. Users cannot override default behaviors. Tables can find their metadata at default locations, but users cannot provide different metadata. Users cannot explicitly reassign tables to their metadata and change their URLs during the validation. These features can be very useful when the files are not published on the web and do not have their expected URLs. It seems that the existing solutions do not support complex validation across multiple files. They expect one table file and one metadata file, and it is a very significant restriction. We may want to validate more interlinked tables with more metadata files. Our solution solves these flaws and gives our users more freedom. We provide complex configuration, and our users can override default behaviors and reassign files together. We solve the assignment with multiple tables and multiple metadata files in full generality. The other solutions

solve only special subproblems.

The first existing solution called `csvw-validator` is the most similar to our idea of a solution. It is made in Java with good architecture and it has a decent interface. However, it has some problems. It lacks support for dialects and foreign keys. It does not pass the W3C tests. Its interface imposes unnecessary constraints. This is inconvenient, and it complicates validation of multiple separate tables. The solution is too focused on the technical problem to see the users and their wishes. These are briefly the main issues on which we want to concentrate. Table 3.2 shows aspects of the proposed solution. It summarizes the features of the proposed solution and provides a comparison with the existing solutions in Table 3.1. The assignment of this thesis requires "a reasonable subset of validation rules". Our goal is at least 200 validation tests from the CSVW Implementation Report [17] with the possibility of future extension. We chose this goal heuristically because it is our idea of "reasonable subset of validation rules" from the assignment.

Aspect	csvw-validator	RDF::Tabular	CSV Lint
Language	Java	Ruby	Ruby
Command line	yes	yes	yes
Web API	yes	no	yes
Web application	yes	no	yes
Online files	yes	yes	yes
Local files	yes	yes	yes
CSV on the Web Schema	yes	yes	rudimentary
JSON Table Schema	no	no	yes
Docker container	no	no	yes
Transform table to RDF	no	yes	no
Transform table to JSON	no	yes	no
Extensive documentation	Czech	no	no
W3C tests success rate	262/281	281/281	281/281

Table 3.1: Evaluated aspects of the solutions

Aspect	Proposed solution
Language	Java
Command line	yes
Web API	yes
Web application	yes
Online files	yes
Local files	yes
CSV on the Web Schema	yes
JSON Table Schema	no
Docker container	yes
Transform table to RDF	no
Transform table to JSON	no
Extensive documentation	English
W3C tests success rate	$\geq 200/281$

Table 3.2: Aspects of the proposed solution

4. Design

In this chapter, we describe the structure, design and principles of our solution. We outline the structure of the solution and its individual parts. We describe responsibilities and dependencies of the parts together with their interfaces. We explain our design decisions with their advantages and disadvantages. The design is guided by the requirements and the assignment. The design prepares the project for the actual implementation. The described skeleton is later filled with the implementation.

The structure of this chapter is similar to the structure of our solution. The solution has three main parts. This division follows good practice of software engineering, and it is also specified in the assignment of this thesis. The core part is the software library. The other two parts provide user interfaces for the library. There is the command line interface application and the web application. We discuss these parts in the following sections.

4.1 Library

The library is the most complex part of the entire solution. It contains the core features and does not depend on other parts of the solution. It performs the validation itself together with other supporting tasks. It is subdivided into parts with distinct responsibilities and higher cohesion. The most important parts are validation checks, documents, and manager. There are also other important parts, but their importance is at the implementation level, and we describe them in the chapter about the implementation. They are not parts of the design. We describe the library and explain its principles in this section. We show individual parts separately, and then we show how they work together.

4.1.1 Validation checks

Validation checks are the heart of the validator. The validation system itself consists of validation checks, and they divide and organize the validation process. Each validation check represents a rule of validation, and it is a building block of the validation. Each check is technically an algorithm of validation. It inspects particular aspects of tables and descriptors. It focuses on particular characteristics of the files. The algorithm accepts files as the input, then it reads the files and returns a result. The validation is not one big monolith, but it is divided into many smaller validation checks with mutual dependencies. Each check performs a small part of the entire validation, and it has a clear meaning, purpose, and focus.

Some checks may be meaningless if some other checks fail. For example, if a table is empty, it is meaningless to check its columns. For this reason, there is a way to specify dependencies between checks. Each check may depend on other checks and their results. Each check is executed after its dependencies. Subsequent checks do not have to start their algorithms from nothing, but they can rely on the preceding checks which check certain qualities of the files. The author of a particular check chooses the dependencies according to the meaning

and the purpose of the check. Naturally, these dependencies cannot be cyclical, and they form a tree. Entire validation is a tree of checks represented by its root.

This system has many benefits. It is possible to define a partial validation with a subtree of checks. A subtree of checks contains and executes only some specified checks. There are significant applications for partial validations. A user may want to perform a partial validation because there are some unavailable files. The partial validation executes the checks in the subtree which do not need the unavailable files. A partial result is returned. The same checks can be reused in other partial validations and the complete validation. This creates an extremely powerful, highly flexible, very extensible and magnificently convenient design.

The tree of checks represent a set of validation requirements, and it can easily change to reflect new requirements. It is possible to extend the capabilities of the validator in this small part of the program. The design limits necessary changes to the absolute minimum. The design also provides a small opportunity for parallel execution of some independent checks in the tree, but the actual benefit is insignificant because the number of independent checks in parallel runs is not large enough, and the time of execution depends more on file reading than processor threads.

The system of validation checks facilitates the development of the validation itself. It provides means of implementation of the real world requirements and constraints. Developers can focus on the main task of their validation checks. They do not have to care about preparation, execution and completion of the entire process. This design allows us to easily implement our validation checks. Other parts of the software perform supporting tasks for this core.

4.1.2 Documents

The validation checks do not use raw access to files. Instead, they use an abstraction. The abstraction provides unified interface for different kinds of files. We call the files and their abstraction "documents". Documents encapsulate certain supporting tasks for the validation itself. Documents help with the validation, and they also provide a predictable environment for developers.

There are two types of documents, i.e. table and descriptor. Table represents a CSV table file, and descriptor represents a JSON-LD metadata descriptor. A table provides access to elements of the corresponding table, i.e. rows, columns, cells. It can also provide raw access to the underlying file itself. A descriptor provides access to the corresponding JSON-LD metadata descriptor. A descriptor is parsed as the JSON format, and the interface provides access to the elements of the JSON model. This is adequately low level of abstraction for complex unobstructed inspection of the files in the validation, but the level of abstraction is still high enough for convenient access. This separation of responsibilities creates possibilities for extensions in the future because developers can extend the interface of the documents according to their needs. Additionally, some documents can be transparently implemented in different ways to adjust the performance of the software. We utilize these benefits of this design in the implementation.

A document can also provide data about itself. Name is one example. A name of a document is the location of the document. It may be a path in a file system or a URL. We discovered the following problem with names and devised a solution

to it. The problem is that a user may want to validate some documents which has not been published yet, or which has been published on a wrong location. In this situation, the links in descriptors do not work correctly because they point to invalid locations. Moreover, relative links in descriptors are resolved with the incorrect base URL, and the library cannot derive locations of descriptors from the names of the corresponding tables. The solution is an alias. Every document may have an alias. Alias is an alternative name of a document. A user can provide an alias for a document. If an alias is present, it is used in the validation process instead of the real name. The name of a document is the path to the actual file, and the alias is the expected URL of the document. A document is located and loaded according to its name, but it pretends that it is on a different location according to the alias. This solves the mentioned problems. This system gives users a powerful tool for specification of documents, and it significantly increases the flexibility and capability of the library in various situations. These are very advanced settings for experts.

The documents themselves provide fundamental access to the files. Authors of validation checks use this access to implement the checks. However, multiple checks may need to find the same thing in documents or perform the same operation with the documents. This is a very probable possibility. In such case, a developer creates a method for the desired task and adds it to a place for utilities. Other developers then reuse it. This approach prevents duplication of source code. Additionally, it provides an extensible higher level of access to the documents without any loss of flexibility. Developers can collaborate and create convenient methods for documents. We call the set of such methods "functional model" because it replaces traditional data model. Why did we decide to avoid a traditional data model? We decided to avoid a traditional data model because it is unsuitable in our case. We must always remember that this software is a validator. It validates files and therefore we cannot expect the files to be in the correct format because otherwise the validation would be pointless. A traditional data model would restrict the shape of the input files and reduce the potential of validation checks. Our functional data model is more capable, and it does not impose undesired constraints.

Documents and utilities provide access to files with perfect balance between flexibility and convenience. They leave open possibilities for future extensions. They also support extensibility of validation checks and protect them from low-level complications of raw files. This design allows the implementation to achieve high cohesion with loose coupling. This is another capable and important part of the library.

4.1.3 Manager

The manager is the central part of the library. It orchestrates other parts and performs supportive tasks. The library converges in the manager, and users use the library via the manager. The manager hides the complexity of the library and provides a unified interface. However, the manager is much more than a simple interface. There are important design decisions and auxiliary features. In this subsection, we describe the design, concepts and principles of the manager.

The manager holds the root of the complete tree of the validation checks. It

also holds the roots of certain subtrees. Let us remind ourselves that the roots are just selected validation checks. The complete tree performs the full validation, and each subtree performs a partial validation. There are two significant subtrees selected. One subtree contains validation checks which read only descriptors. This subtree validates only descriptors, no tables. The other subtree contains validation checks which read only tables. This subtree validates only tables, no descriptors. These two partial validations are performed in the full validation. Therefore, the two subtrees are parts of the complete tree. We can clearly see the advantages of this ingenious system. Individual checks perform smallest reasonable parts of validation. Checks are grouped in subtrees, and the subtrees perform partial validations. Subtrees form a complete tree, and the complete tree performs full validation. We can use the checks in various ways without any duplication of code.

The manager accepts input from users. The input is a validation request. The request carries the necessary data for the validation, e.g. names of documents, aliases, configuration for the implementation. The manager uses the request to create corresponding documents. There is an important concept. A user can specify active and passive documents. Specifically, a user can specify active tables, passive tables, active descriptors, and passive descriptors. These two types of documents are fully transparent to the validation. Validation checks do not know whether a document is active or passive. The documents themselves do not know whether they are active or passive. This concept exists only in the manager and the validation request, and only the manager distinguishes these types. The difference is simple. An active table attempts to locate and load its descriptor, a passive table does not. An active descriptor attempts to locate and load its tables, a passive descriptor does not. It means that a user must explicitly provide possible required documents for corresponding passive documents.

If a partial validation for tables only is performed, only passive tables are validated, and active tables are ignored. If a partial validation for descriptors only is performed, only passive descriptors are validated, and active descriptors are ignored. Active documents are used and validated only in the full validation because they do not make sense in partial validations. Active documents attempt to locate and load the corresponding documents, but this behavior is meaningless in partial validations. The manager is responsible for that. The manager technically locates the files, loads them and implements the other features.

The manager creates an application programming interface (API) for direct users of the library. The interface of the validation request is complex enough to accurately specify individual documents and the configuration of the library. It means that the interface is quite complex. The interface contains only the intrinsic complexity without unnecessary complications. The result of the validation is a validation report. The report contains well organized data about the performed validation. The report supports serialization in the required formats, and it can be extended to support more formats. The localization and natural languages of the library are not a part of the mentioned interfaces. Locales are set separately as a context of validation because they usually do not change with every validation request. We kept the locales outside of the main API and thus we simplified the API. This way, we can create a capable library interface which is also as convenient as possible. Some aspects and options of the interface depend on the

implementation and therefore the details of the interface are described together with the implementation in other chapter.

The manager stands in the center of the library. It is a natural place for the implementation of the interface of the library and for shared values of the configuration. It also performs certain supporting tasks which are immediately connected with the interface. The library can be easily extended and the interface of the library can be also easily extended to accommodate future changes. The manager is a central part, but it is not a bottleneck.

4.1.4 Summary

The library is logically divided into several important parts with distinct responsibilities. We show individual parts and their relations. Each part was designed with best practices of programming in mind. Flexibility and extensibility are woven into the fabric of the library. These qualities create good conditions for future development beyond the scope of this thesis. Clever design and simple principles limit the necessary code comprehension to the absolute minimum. Doors are open for possible development of additional features. The design facilitates our implementation of the library, and it keeps its own potential.

4.2 Command line interface

The command line interface application provides direct access to the API of the library via the command line interface (CLI). It translates the API of the library into CLI options. The CLI options correspond to certain elements of the library API. There are several groups of similar options. There are options for the specification of individual documents. They provide a way to fully specify a document, e.g. table, descriptor, active, passive etc. There are options for the output. The CLI applications shows the output of the validation as text on the command line and therefore the output must be serialized into desired format in desired language. These option configure the format and the language of the output. There are options for the type of validation, i.e. tables only, descriptors only, full validation. Finally, there are supporting options. These options are necessary for the implementation of the application, and they do not correspond to the features of the library. Some options depend on the implementation and the API of the library and therefore they are described in the chapter about implementation.

4.3 Web application

The web application provides a friendly and simplified graphical user interface. It is the web interface of the validation library. The web application makes this project more accessible and easier to use. It can also popularizes the project on the web and describes some of its features. The nature of web environment introduces unique challenges and possibilities. It shapes the design of the web application. The design of this part of the project is distinct from the rest of it. This section describes the design.

The web application and its design is divided into four parts, i.e. client, server, database, and worker. It is possible to implement all parts in one unified ecosystem of technologies. However, such approach imposes undesirable constraints. One technology is not the best choice for every task. We want to have the freedom to choose the best technology in every part of the design. Individual parts are designed to provide this freedom, and the design as a whole supports this freedom. Different technologies are connected with universal interfaces. We describe individual parts and their connections. There is a diagram of the web application in the appendix C. It can help with comprehension.

We should mention that the design of the web application does not support the validation of local files on user's computer. It is not an expected use case. Only online files with URLs are supported. We must remember that we use a standard called CSV on the Web and therefore we focus on the validation on the web. We do not want to allow users to upload their files to our web application because the benefit is not significant enough, and it can introduce undesired vulnerabilities, exposures, and complications. It is better to use the command line interface application for such validations. Alternatively, users can upload their files on the web with some other web application and then use our web application to validate them. There are many ways which allow a user to publish files. We do not feel the need to implement another one because we do not want to complicate interfaces and deployments of our web application.

4.3.1 Client

The client part of the web application provides a web-based graphical user interface. It is a set of web pages and therefore it is expected to run in a web browser. This environment affects the design and later the implementation. There are important pages with the main functionality and auxiliary pages with additional content. The main pages provide different ways to submit a validation request and display results. The auxiliary pages provide a convenient way to publish extra information, e.g. information about the project, instruction manual, useful links, etc. It is easy to add more pages and extend the presented information. We implemented these pages, but each page alone is not an important part of the design. We focus on the important pages with dynamic content and user interaction. We created mock pages in order to illustrate our design. The mock pages are created in plain HTML without styles and functionality.

There are three important pages, and they all contain a web form. Every form is used to submit a validation request. Two forms are as simple as possible. One of them has a single input field for a URL of a table, and the other has a single input field for a URL of a descriptor. Figure 4.1 shows mock table validation form. Descriptor validation form is very similar. They both have a switch which sets whether the corresponding document is active or passive. This switch changes the behavior of the document and the validation style. Passive documents do not attempt to locate and load the missing files, and only partial validation is performed. Active documents attempt to locate and load the missing files, and full validation is performed. This system directly corresponds to the concepts of the validation library. They also have a selector which sets the language of the validation report. These forms provide only a limited access to the features of

the validation. They are simple, and they support common use cases. Users can easily and quickly submit a validation request with one table or one descriptor.

OTAVA - WebApp [Search validation](#) [About this application](#) English ▾

Table validation

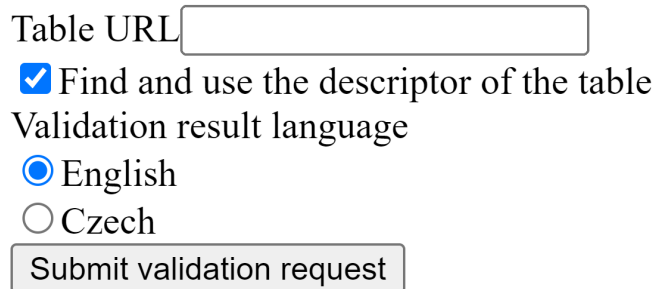


Table URL

Find and use the descriptor of the table

Validation result language

English

Czech

Figure 4.1: Mock table validation form

The third page with a form is significantly more complex. Figure 4.2 shows mock page with this expert validation form. There is a selector which selects a validation style. Possible options are: full validation, partial validation with tables only, and partial validation with descriptors only. There are four large input fields for documents, i.e. passive tables, active tables, passive descriptors, and active descriptors. Each line of the fields represents one document with a URL and optional alias separated by a space. There is a selector which sets the language of the validation report. These options and concepts directly correspond to the concepts in the validation library. The values are simply given to the API of the library. Finally, there is an input field for a description. It just provides an opportunity to add a human-readable description to the validation request. It is not used for anything. It is only saved.

There is a page for the results of the validation, i.e. the validation report. It shows a disabled form with the data from the validation request and a status of the request. The status shows whether the validation is finished. If the validation is finished, there is a validation report. The report is displayed in all available technical formats. The report is in the selected language. The report is generated by the library, and it shows the results of the validation. The page for the results has a unique URL and it can be repeatedly loaded later.

The client is connected to the server via an interface. The client sends requests and receives responses. The server provides URLs for connections. The client can be fully replaced with another implementation because the interface is universal. Multiple different clients can use the server together. Everyone can create their own client or extend the existing client. Some clients can be tailored to special needs and situations. There is not any constraint.

In summary, the client of the web application is a flexible and extensible interface of the validation library. It provides different pages with different complexity for different kinds of users. Beginners can enjoy the simplicity of short and clear forms. Experts may like larger and more powerful forms. If someone does not

like our interface at all, the interface can be changed, extended, or completely replaced with some other implementation. The client does not limit the library and the project, it enriches them.

OTAVA - [WebApp Search validation](#) [About this application](#) English ▾

Expert validation

Validation style

- Full validation
- Tables only validation
- Descriptors only validation

Passive tables

Active tables

Passive descriptors

Active descriptors

Validation result language

- English
- Czech

Description of the request

Figure 4.2: Mock expert validation form

4.3.2 Server

The server is the central part of the web application. It is quite small because its only responsibility is to connect the client to the database. The server is connected to the clients on one side and to the database on the other side. It provides URLs and listens for connections. It accepts data from forms and creates responses. We can simply extend or modify the behavior of connection handlers. The server provides the web interface. There are URLs for pages with static content and for static parts of other pages. One URL provides access to the validation data of submitted validation requests with validation reports. Other URLs accept data from the web forms.

Data model is a part of the server. Connection handlers use the data model to access the database. The server is also responsible for the initialization of the database. It creates the necessary schema and table in the database. We need only one table in the database. The table contains columns for the data. Some columns depend on the implementation details. We describe the table in the chapter about implementation. The model provides a facade over the database, and it executes the queries.

4.3.3 Database

The database simply stores the necessary data, i.e. mostly data from the web forms and results of validations. We need to create, update, and delete records. There are many different database technologies, but relational databases stand out among them. Relational database stores data in rows of tables. It allows us to easily perform the required tasks. It is a time-proven and popular technology in web environment, and we decided to use it. We do not need to perform demanding operations in the database. More advanced databases have undesired complexity for our purpose. We do not want to store the data directly in files because raw files in a file system do not ensure data integrity and parallel access. We do not expect large volumes of data in the database, but a relational database is still ready for it.

Database can be integrated in other technologies. We decided to avoid possible integrated databases because an integrated database can significantly reduce flexibility of deployments. We chose a traditional external database. It means that our design does not depend on any particular database. We can easily choose our favorite relational database and other people can easily replace our database with their own favorite database. The database communicates in the Structured Query Language (SQL), and the SQL queries in other parts of the web application do not depend on any particular database. This is another flexible part of our design.

4.3.4 Worker

The worker is the powerhouse of the web application. It directly uses the validation library to perform the actual validation. The worker is very similar to the command line interface application. The command line interface application uses the command line interface to obtain the user input and print results. The worker uses the database to obtain user input and store results. The worker queries the database and obtains a validation request. It uses the data to create a request for the validation library. The library performs the validation, and the worker serializes the report and stores it in the database. The worker is not included in other parts, it stands alone. It means that we can easily replace the worker with different implementations. It also means that we can spawn more instances of the worker in a deployment and increase the validation performance of the web application. This is a very scalable design. Naturally, the implementation must be ready for such parallel deployment, but it is not a problem. The worker is a powerful and flexible part of the web application.

4.3.5 Summary

The web application is the most diverse part of the project. It successfully brings together various technologies for various tasks. It has a modular design, and individual parts can be easily extended, modified, and replaced. This web application provides the validation services with an unparalleled level of flexibility. There is an unlimited potential for future development of additional features. Users can enjoy nice and uncluttered user interface. The application is tailored to different skill levels of individual users. It greatly enriches the entire project.

4.4 Summary

The design in this chapter forms a perfect skeleton of the project. It describes the structure and high-level decisions. It gives us a good idea of the project and its individual parts. The design upholds the best practice of software development, and it facilitates the extensible and flexible implementation according to the best principles. This chapter intentionally omits many details because they are not important for the design itself, and they would only clutter the text. We describe the low-level details in other chapters. The implementation brings the design to life and adds more details to the project.

5. Implementation

In this chapter, we describe the implementation of the project and its parts together with related technologies. It should be mentioned that we named the project OTAVA – Open Table Validator. Most of the project uses the Java programming language, specifically Java version 17.0.1 was used and tested because it was the latest version with long term support when our work started. The implementation follows the design in the previous chapter. We explain the implementation of the parts from the design. There are also other parts integrated in the text because they are important for the implementation. The repository with the source code of this project is in the attachment A.1.

5.1 Structure of the project

This project is logically structured into directories, modules and packages according to mutual dependencies and technologies. In this section, we introduce the project and describe its structure. The separation of individual parts of the project helps to ensure good practice and principles of high-quality maintainable source code. The project lives on GitHub [15], and it contains a configuration for continuous integration workflow. GitHub automatically builds the project and executes tests. High overall quality is our priority, and automatic tests help us assure the quality. The Java part of the project is separate from the web application part which uses different languages.

The Java part is managed by Maven [38], and it contains three modules, i.e. the library, the command line interface application, and the web worker. There are direct dependencies between the modules and external libraries. They are organized in the project object model (POM) files. The web application part is implemented with different technologies, and it does not have any direct explicit dependency on the Java part. However, there is an implicit dependency on the web worker, and we explain it later in this thesis. The web application is managed by the Node Package Manager (NPM) [21].

5.2 Library

The library is the largest part of the project. It contains the core features. It performs the validation itself together with other supporting tasks. The library is a reusable module in the Java Maven project object model. It is a clear and simple dependency, and it makes the library easy to integrate and use in other modules and projects. It is usually a good idea to separate the core part of the project from other parts because it increases the quality of the source code and simplifies possible additional development in the future. Moreover, the assignment of this thesis clearly requires a Java library. The library is subdivided into several packages with distinct responsibilities. These packages form the skeleton of this section. We describe the library and explain its implementation principles in this section. We show individual parts, their technologies, algorithms and implementations. The library is in the main package `io.github.janjanda.otava.library`.

5.2.1 Validation checks

The design of the validation divides the entire validation into many small validation checks with dependencies. Each check contains an algorithm which inspects certain qualities of input files. This part of the library is implemented in the sub-package `checks`. There are supporting structures and actual implementations of the individual checks. The abstract class `Check` is a common parent for all checks, and it supports their implementations. This base class prepares, executes and completes concrete checks. Specifically, the `Check` base class

- performs the necessary common tasks,
- accepts and stores the documents,
- stores other checks as dependencies, we call these dependencies pre-checks,
- stores the result of the validation check,
- provides methods to execute the validation and to get results, and
- declares the abstract method `performValidation`.

The abstract method `performValidation` is the designated place for the validation algorithm. This abstract method is used in the method `validate` to perform the validation. Their relation is similar to the strategy design pattern. The abstract method performs the actual validation and returns a result builder with the results. The method `validate` adds the duration of the validation to the builder and builds the final result of the validation check. The validation is executed only once. Repeated calls to `validate` do not trigger the validation because it is not necessary.

New validation checks can be easily added to this system. We simply create a class which extends the abstract class `Check`. The constructor accepts a check factory as a parameter. The constructor gets the documents from the factory and gives them to the base class. It also uses the factory to create pre-checks and gives them to the base class. Then, we override the abstract method `performValidation` and implement our validation algorithm in it. The documents are available from the fields of the base class. The validation algorithm reads the documents, validates them, and returns a result. Each validation algorithm should focus on one particular aspect of the overall validation. Our implemented checks usually find a table and its description in the documents, and then check that the table matches one particular aspect of the description. The new check can be integrated as a dependency in a tree.

The class `Result` is a simple immutable class with a nested builder. It gathers and stores data from the execution of a validation check. It stores the name of the corresponding check, duration of the check, state, and messages. The state can be `ok`, `warning`, `fatal`, or `skipped`. The messages contain text messages from the check. They describe discovered problems with the validated files. A result can write itself in plain text, JSON, and Turtle. Each validation check creates a result instance. It is the output of the checks.

The meaning of the state in a result is pretty self-explanatory, and checks can easily use it. If the state is `ok`, no problems were found. If the state is `warning`,

some problems were found, but they are not very serious. If the state is fatal, serious problems were found. If the state is skipped, the check was not performed. Our implemented checks use the following behavior. If a subtree of a particular check contains a fatal result, then the check is skipped because it does not make sense in this situations. This behavior is not mandatory. New possible checks in the future may choose a different behavior.

The check factory is an interface which helps with the creation of validation checks. We can get tables, descriptors, and instances of checks from it. There is one implementation of the check factory. It is the class `SingletonCheckFactory`. This implementation ensures that there is only one instance for each validation check class. This is an important optimization. If two or more checks depend on one common check, this common dependency has only one instance, and the instance is shared. This arrangement prevents unnecessary repeated work in the check and increases performance. The factory checks whether the dependencies between the validation checks are not cyclical.

The abstract class `Check` contains other supporting methods. It can return the result of this check and a set of all results from the tree of its pre-checks including its own result. Developers can easily get lost in pre-check dependencies. Fortunately, there is method which creates a graphical representation of the tree from the current check. The graphical representation and the implemented checks are in the following subsection. Some checks appear on more places in the tree, but remember that there is still only one instance for each check, and the one instance appears as a dependency on more places. The name of the check is duplicated because we want to show the tree without cycles.

Some annoying mathematical purist may say that this structure is not really a tree because there are common shared dependencies and therefore one child can have more than one parent. We say that it does not matter. One common shared dependency is represented by more vertices in the graph, and the graph is a tree. These vertices have one common name in the graphical representation in Figure 5.1. One common shared dependency is implemented with only one shared instance because it improves performance. This is an implementation detail, and it does not spoil the structure.

It is important to mention the implicit meaning of checks. Every implemented validation checks has its purpose and explicit meaning. Naturally, there are also implicit meanings in implementations. For example, if a check explicitly checks the content of the context property in descriptors, it also implicitly checks that the property exists. These side effects are an unavoidable nice bonus, and they silently enhance the validation capabilities.

5.2.2 Implemented checks

We tested the viability of our validation system and implemented practical validation checks. It proves that the system is viable, and it brings our validator to life. Each validation check focuses on a particular aspect of the validation. All checks together create a complete validation capability. We know that every validation check is implemented in its own class. The name of the class is the name of the corresponding check. Here, we describe every check and its purpose. This description provides an idea of the validation capabilities. We know that

the checks are organized in a tree. Figure 5.1 shows a command line interface with the tree of checks.

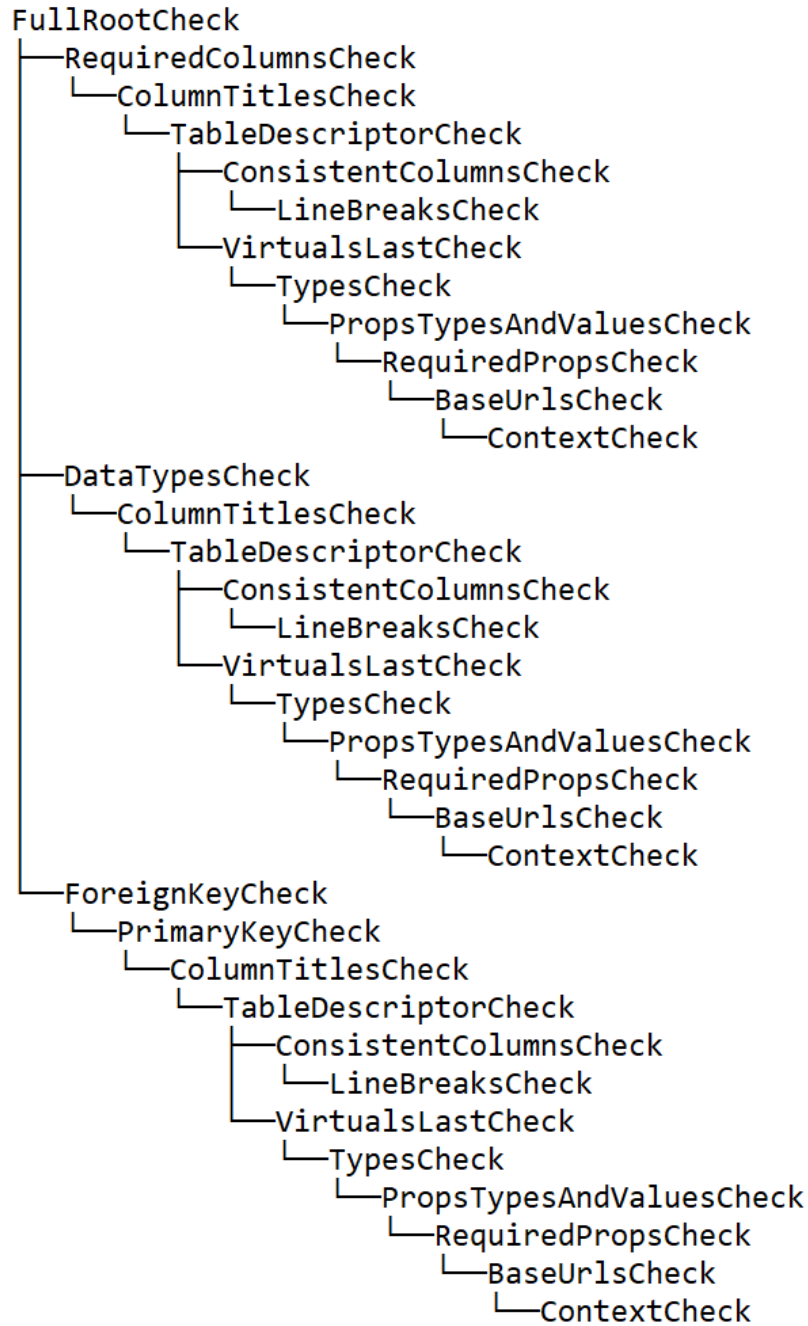


Figure 5.1: Generated graphical representation of the checks tree

BaseUrlsCheck

This class checks base URLs of descriptors. Base URL in a descriptor is very important because it is used to resolve other URLs in the descriptor. This class checks that the base URL is available and correct. If the base URL is wrong, the validation practically cannot continue. Usually, the base URL is the URL of the descriptor, but it can be changed in the context.

ColumnTitlesCheck

This class checks columns in tables and their respective descriptors. There should be columns with same titles in a table and its descriptor. Every column in a table must have a description in the corresponding descriptor, and every column description must have a column in the corresponding table. Naturally, this does not apply to virtual columns.

ConsistentColumnsCheck

This class checks size of rows in tables. Each row in a table must have the same number of cells. It also checks whether tables are not empty. Every table must have this basic rectangular structure, otherwise it is not a table. This class is the root of the subtree for the partial validation with tables only.

ContextCheck

This class checks the @context fields of the metadata descriptors according to the corresponding W3C Recommendation. The context field in a descriptor must have the URL of the CSVW external context. Additionally, it may contain a base URL and a language. If the context contains more than the URL of the CSVW external context, it is an array. Possible structures of the context are clearly defined.

DataTypesCheck

This class checks whether values in tables match their defined data types. The data types are defined in column descriptions. Most common data types are supported. It supports the following built-in data types:

- string
- anyURI
- boolean
- date
- datetime
- number
- integer
- time
- byte
- unsignedLong
- unsignedShort
- unsignedByte

- `positiveInteger`
- `negativeInteger`
- `nonPositiveInteger`
- `nonNegativeInteger`
- `double`
- `float`

It also supports derived data types. A derived data type uses a built-in data type as a base and other properties to restrict the base data type. The listed numeric bases support minimum and maximum constraints (both inclusive and exclusive). The base `string` supports length, minimum length, maximum length and format constraints. The format expects a regular expression. The bases `date`, `time` and `datetime` support format constraint. The format expects a datetime formatter pattern. Unsupported features and invalid constraints are ignored because they may still be reasonable outside the scope of this validator.

ForeignKeyCheck

This class checks specified foreign keys between tables. A table description can define a foreign key which imposes a constraint on values in columns.

FullRootCheck

This class does not perform any actual check. It is a root for the tree of other checks. This is the main root of the tree for the full validation.

LineBreaksCheck

This class checks the line breaks of the tables according to the CSV specification. Line breaks must be CRLF. This check does not create fatal results because this requirement is not universally accepted.

PrimaryKeyCheck

This class checks the defined primary keys of the tables. Values of a primary key must be unique in the table.

PropsTypesAndValuesCheck

This class checks actual types and values of properties in descriptors. Some objects in descriptors must have certain properties with certain values according to the specification. For example, tables array cannot be empty, table schema must be an object, columns must be an array, etc.

RequiredColumnsCheck

This class checks required columns in tables. Cells of required columns cannot be empty.

RequiredPropsCheck

This class checks the required properties in descriptors. Descriptors must have certain properties, e.g. tables in table group.

TableDescriptorCheck

This class checks whether every table has a description in descriptors, and every description has a corresponding table. It couples tables with descriptors. It also checks whether table URLs in the descriptors can be resolved. Naturally, this check can handle a descriptor with one table and a descriptor with more tables.

TypesCheck

This class checks @types of entities in descriptors. These properties contain RDF types, e.g. a column description should have @type value Column.

VirtualsLastCheck

This class checks the order of virtual and non-virtual columns in table descriptions. Virtual columns must appear after non-virtual columns in a table description. This class is the root of the subtree for the partial validation with descriptors only.

5.2.3 Documents

Documents in the library are an abstraction over actual CSV files with tables and JSON-LD files with metadata. We utilized inheritance in Java and created a hierarchy of interfaces and classes. Figure 5.2 shows the inheritance between the documents. This part of the library is implemented in the subpackage `documents`. The classes implement access to different types of files in different locations. The interfaces hide the complexity and provide a unified set of methods. This approach separates the implementation details from users of the documents, and it allows users to easily add new implementations according to their needs.

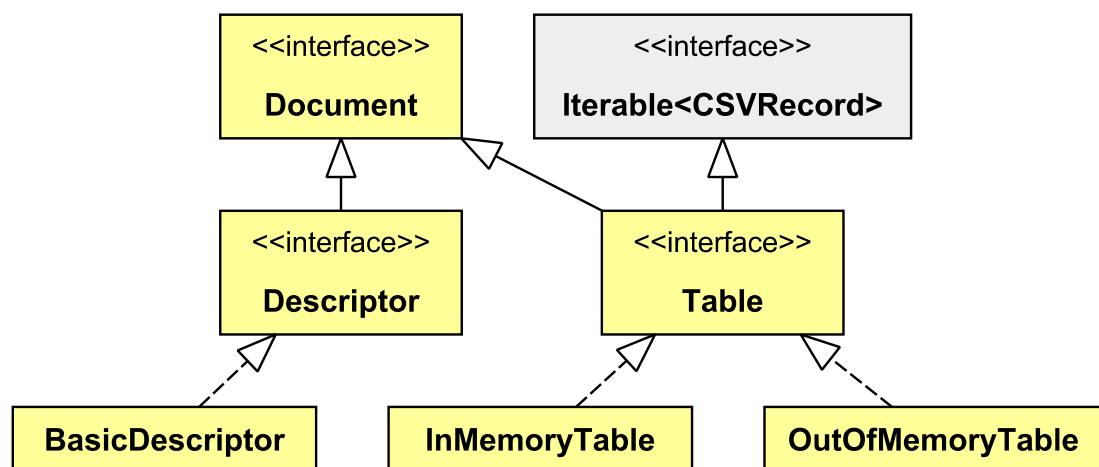


Figure 5.2: Class diagram of the documents, gray interface is from `java.lang`

The main interface `Document` declares three common methods which simply get the respective information about the corresponding document. These are name, alias, and preferred name. Every document has a name. A name is an actual location of the document, i.e. path in a file system or URL. Every document may have an alias. An alias is an alternative name. The preferred name provides an option to return the best available name. Currently, implementations return alias if an alias is available, otherwise they return the name. New implementations may implement it differently without any constraint. We cared about extensibility in every little place of the source code.

Documents can be either local in a file system or online on the internet. This duality is not represented by individual types of documents. We chose a simpler solution instead. There is a functional interface `ReaderMaker` which represents a function for creation of an input reader from file name. This interface can be implemented with two prepared functions. One reads from file system, and the other reads from URL. Implementations of the `Document` interface can expect the functional interface `ReaderMaker` as a parameter and use it to create their readers. A particular implementation of the functional interface can be then provided, and the behavior of the corresponding document can be modified this way. It is similar to the dependency injection pattern, and it provides the desired extensibility. A user can easily implement new documents with new input readers without unnecessary constraints.

The class `DocsGroup` is simply an immutable collection of documents. We implemented our own simple collection because we were not satisfied with the collections in Java. Groups of documents are shared across the library. We wanted to increase the security in the source code by providing a tight interface. Some standard collection would also work, but it would not provide semantic meaning of the type and related understanding of security.

There is a document factory class. It simplifies the creation process of document instances. It accepts certain data as parameters, it can get other data from other sources. The factory currently works with one main CSV format object for CSV parser. It is possible to easily add support for more dialects because the factory can be simply extended. The factory can be used to better adapt the process of creation. This is the main purpose of the factory.

Tables

The interface `Table` extends the interfaces `Document` and `Iterable`. It declares methods for standard iteration over the lines of the table and for reading the first line. It can also provide raw reader for maximum flexibility. This interface creates the final abstraction for tables, and other parts of the library use this type to represent a table.

The interface table is implemented by two classes. Both classes use the library Apache Commons CSV [40]. Apache Commons CSV is a popular Java library for CSV processing. It provides a simple and convenient interface methods, and it is a default choice for developers. It is easily accessible via the Maven Central Repository. We chose this library for these reasons. The two classes use the same library, but they have an important distinction. One of them represents an in-memory table, and the other represents an out-of-memory table.

The in-memory table class loads the table completely into memory. The CSV file is read and parsed only once. This makes any subsequent access to the data faster, but it may require a lot of memory. It is better for small tables. The out-of-memory table class does not load the table completely into memory. It saves the system memory, but the file is read multiple times. Therefore, every access to the data takes more time. Users can choose the best implementation for their purpose and adjust the performance. They can prefer memory or speed.

Descriptors

The interface `Descriptor` extends the interface `Document`, and it represents a CSVW metadata JSON-LD file. The JSON-LD format has a structure of a tree, and the interface provides access to the root node and a shortcut to access children according to the path. Only one class implements the interface. It uses Jackson [12]. Jackson is a popular Java library for JSON processing. It is significantly more popular than other alternatives, and it is a default choice for developers. Its is easily accessible via the Maven Central Repository. We chose this library because its popularity gave us enough confidence. Descriptors are usually small even if they describe large tables. We do not provide an out-of-memory implementation, but it can be always added in the future. Our implementation simply provides a facade over file readers and the Jackson library.

Descriptor files have the JSON-LD format which is derived from the JSON format. It means that we can read a JSON-LD file as a JSON file, and we chose this approach. Why did we choose the JSON approach over the JSON-LD approach? In the early stage of the development, we tried to use Apache Jena [41]. Apache Jena is a specialized open source Java framework for linked data applications. It can be used to parse specifically JSON-LD. We must remember that this project is a validator. The program cannot expect a valid JSON-LD format because one of its tasks is to validate the format. The program needs direct access beyond JSON-LD to see the JSON structure and validate it. In addition to that, Apache Jena has complex interface and some problems with linked context. We decided to avoid Apache Jena and use better suited Jackson.

5.2.4 Manager

The manager is a broad term in our case. It includes the manager of the library, API, and certain supporting structures. The most important class is `Manager`. It gathers other parts of the library in one place. The manager contains shared values and the main configuration for the library. The public methods in the manager comprise large part of the API for the library. There are not superfluous constraints, and the manager can be easily extended with new methods and functionalities. The manager is responsible for several things, and we describe them here. Figure 5.3 shows a sequence diagram with the manager.

Locales

The manager stores the locale settings for the entire library. The locale is in a static field in the manager. It means that it is easily accessible and usable everywhere in the library. There are static methods which get and set the locale.

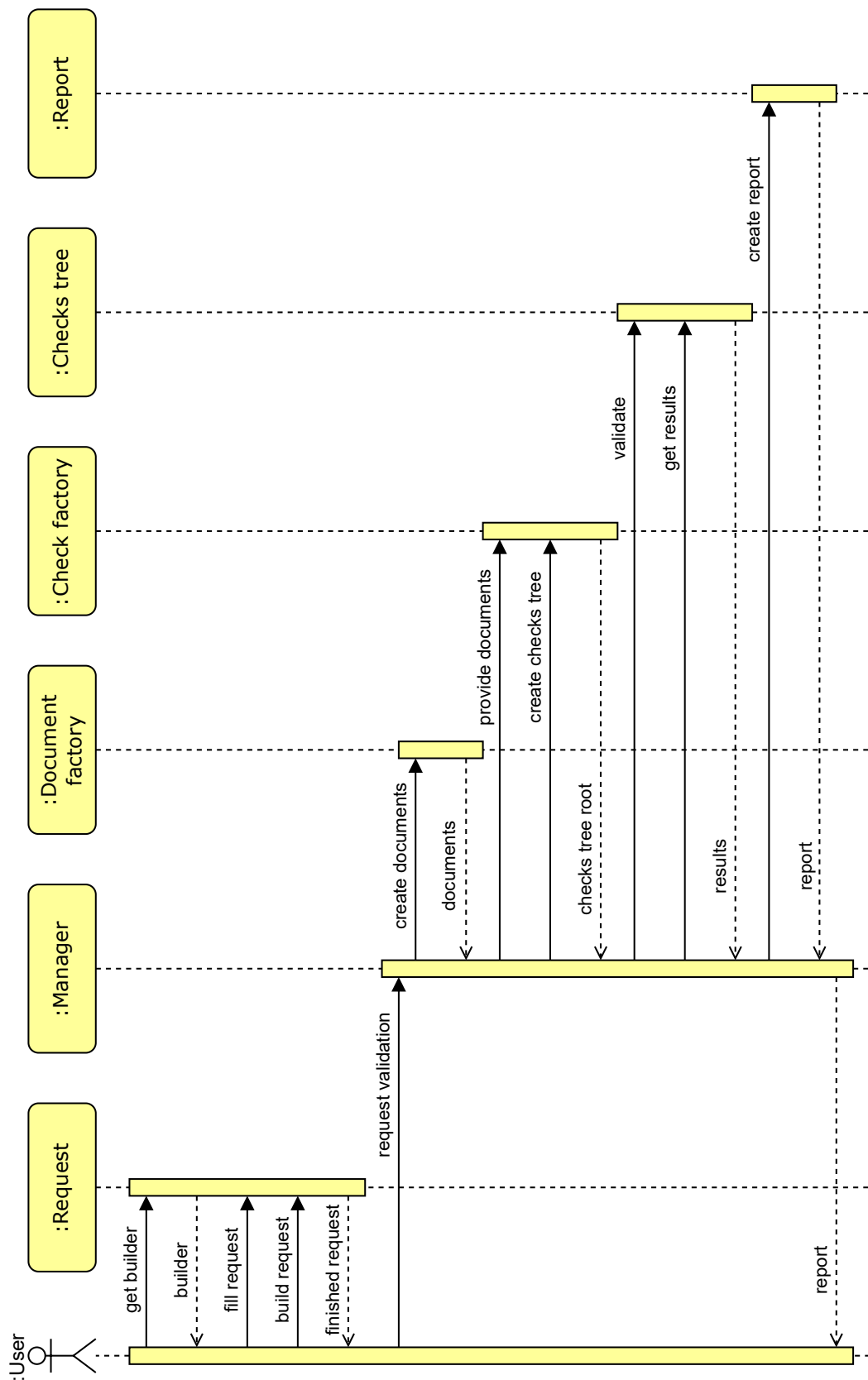


Figure 5.3: Simplified sequence diagram of the user's interaction with the library

The Java language contains some support for internationalization with resource bundles, but these features contain excessive complexity. Resource bundles get translation from external files, and the required strings are identified by string identifiers. The bundles have unsafe interface because a typo in an identifier may cause a hidden error. We are not satisfied with this feature in Java and therefore we implemented our own locales. They are simpler and safer.

There is an interface `Locale` in the subpackage `locales`. It declares the methods for locales. The methods return strings in desired languages. The methods may accept strings as parameters and integrate them into final strings. The classes `CzechLocale` and `EnglishLocale` implement the interface. Each class implements the methods in the corresponding language. The interface is simple and safe because identifiers of methods and parameters are checked in compile time. It is very easy to add another language. A developer simply creates a new class which implements the interface. The desired class is then stored in the manager in the field with the type of the interface `Locale`, and it is then used in the library. Our approach effectively uses virtual methods and polymorphism.

Validation

The manager has three methods for three styles of the validation, i.e. tables only, descriptors only, and full validation. The methods accept an instance of validation request as the only parameter and return an instance of the validation report. A validation request contains necessary data for the validation from a user, and a validation report contains results of the validation. This interface is as simple as possible. The manager uses the data from the submitted request to create the documents. Then, the manager uses the check factory to create the corresponding tree of validation checks. Each tree is specified by its root. The manager stores the roots. The class `ConsistentColumnsCheck` is the root for the tables only validation. The class `VirtualsLastCheck` is the root for the descriptors only validation. The class `FullRootCheck` is the root for the full validation.

There is an additional step in the full validation. The manager attempts to create the required documents for the active documents in the validation request. The manager reads the active descriptors, finds table URLs in them, and uses the URLs to create tables. The manager gets the preferred names of active tables, uses the preferred names to create names of required descriptors, and creates the descriptors. This process happens only once. New located and created documents are not used to locate and create more new documents because such behavior is unreasonably complicated and may cause infinite loops.

The official CSVW standard describes several ways of locating metadata. We implemented the most common way because it is used in the vast majority of use cases, and the other ways require advanced network communication. Our way simply adds the suffix `"-metadata.json"` to the end of the name of the table. The system is very extensible and uses factories. The other ways of locating metadata can be implemented in the future, but we do not think they are very useful. Our validator allows users to explicitly provide the locations of the documents. Users can easily override incorrectly located documents. It is not worth the effort to add complex ways of locating metadata to the validator.

The validation request in the class `Request` contains large part of the library API. The class contains a builder for convenience. It gathers user input to the

validation. Users can add passive tables, active tables, passive descriptors, and active descriptors. A descriptor has a name and a flag which sets whether the descriptor is local or online. It can also have an alias. A table has the same things and a flag which sets whether in-memory table or out-of-memory table is used. There is a save memory flag in the interface, and users can set it. If they do it, out-of-memory tables are used for the tables automatically located from active descriptors. Otherwise, in-memory tables are used for them.

The validation either returns a validation report or throws an exception. The validation report is implemented in the class `Report`, and the validator exception is implemented in the class `ValidatorException`. Both classes implement the interface `Outcome`. The interface declares methods for serialization in three formats, i.e. plain text, JSON, and Trutle. This abstraction means that we have a valid outcome of the validation in every case. The report stores data about the validation, i.e. duration, table names, descriptor names, and results of the individual validation checks.

Finally, the manger has methods which return graphical representation of the validation check trees as strings. The manager can print three trees, i.e. tables only validation tree, descriptors only validation tree, and full validation tree. These trees correspond to the three validation styles. The manager simply uses the method implemented in the validation check class. Developers and users can use them to view the structure of the validation.

5.2.5 Tests

The library contains a substantial suite of unit tests. We used the JUnit testing framework [42] because it is a usual choice for testing in Java. The tests thoroughly test the important features of the library. There are tests for individual validation checks. They test the behavior of the checks with different tables and descriptors. They also test the check factory. Other tests test the documents and data model. Documents in tests need input files. The files with tables and descriptors are in the test resources. We implemented the validation tests from the CSVW Implementation Report [17]. They are used to evaluate our solution with external criteria, and we discuss them in the chapter about the evaluation. The automatic tests help us with quality assurance, but we used manual tests too.

5.3 Command line interface

The command line interface application is implemented in Java, and it is directly connected with the validation library. The dependency is described in the POM file. The application is a bridge between the command line interface and the API of the library. It uses the library Apache Commons CLI [39]. The library allowed us to easily define our command line options, and it takes care about the user input. It is a popular library, it comes from the Maven Central Repository, and it is also described in the POM file. The library is flexible, and we can always add more options in the future. The interface is the main point of the library, and the options are the most important part. The application only takes the values from the options and gives them to the library. Then, it prints the results.

```

usage: otava [-help] [-tablestree -descstree -fulltree] [-lang en|cs] [-savemem] (-tables | -descs | -full)
[-text | -json | -turtle] [-sep <char>] [-plit <path[;alias]...>] [-plot <path[;alias]...>]
[-poit <url[;alias]...>] [-poot <url[;alias]...>] [-alit <path[;alias]...>] [-alot
<path[;alias]...>] [-aoit <url[;alias]...>] [-aoot <url[;alias]...>] [-pld <path[;alias]...>]
[-pod <url[;alias]...>] [-ald <path[;alias]...>] [-aod <url[;alias]...>]
-ald <path[;alias]...> add active local descriptor to the validation request
-alit <path[;alias]...> add active local in-memory table to the validation request
-alot <path[;alias]...> add active local out-of-memory table to the validation request
-aod <url[;alias]...> add active online descriptor to the validation request
-aoit <url[;alias]...> add active online in-memory table to the validation request
-aoot <url[;alias]...> add active online out-of-memory table to the validation request
-descs validate only passive descriptors alone
-descstree print the tree of validation checks for descriptors
-full perform full validation with tables and descriptors
-fulltree print the full tree of validation checks
-help print this description of the options
-json print result as JSON
-lang <language tag [en|cs]> set the language of the results, en is default
-pld <path[;alias]...> add passive local descriptor to the validation request
-plit <path[;alias]...> add passive local in-memory table to the validation request
-plot <path[;alias]...> add passive local out-of-memory table to the validation request
-pod <url[;alias]...> add passive online descriptor to the validation request
-poit <url[;alias]...> add passive online in-memory table to the validation request
-poot <url[;alias]...> add passive online out-of-memory table to the validation request
-savemem do not load tables from active descriptors to memory
-sep <regex> set separator between name and alias of a table or a descriptor, ; is default,
regex parameter to String.split(String regex)
-tables validate only passive tables alone
-tablestree print the tree of validation checks for tables
-text print result as plain text
-turtle print result as RDF 1.1 Turtle

```

Figure 5.4: Command line interface with the options

The application is implemented in a single class `CliApp`. There are the definitions of the options. There are many options for maximum user comfort. The option `help` prints the description of the options, and Figure 5.4 shows that. The options `tablestree`, `descstree`, and `fulltree` print the graphical representation of the respective validation tree. The option `lang` sets the language of the result according to its argument. The option `savemem` sets the save memory flag in the validation request. The option `tables` executes the partial validation for tables only. The option `descs` executes the partial validation for descriptors only. The option `full` executes the full validation. Only one style of validation can be selected. The option `text` prints the report in plain text. The option `json` prints the report in JSON. The option `turtle` prints the report in Turtle. Users can select multiple output formats.

There are many options for documents. The options are short acronyms for the sake of brevity. These options may appear many times for more documents. Users select and use the correct option for each validated document. Each option expects an argument. The argument is the name of the document and optional alias separated by the separator. The separator is semicolon, but it can be changed with the option `sep`. The acronyms in the options clearly refer to the concepts in the API of the library. Figure 5.4 shows the command line interface after the option `help` was used. There are the options and their descriptions.

5.4 Web application

The implementation of the web application combines various languages and technologies. We decided to use popular and widely used technology stack instead of integrated Java technologies. This good choice enables easier development in the future because there are more developers with the necessary experience with our environment. The implementation has distinct parts according to the design. Each part uses different approach and different technologies. Appendix C contains a useful diagram of the web application.

5.4.1 Client

The client of the web application is just a set of web pages. This context determines the languages of the implementation. Web uses Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript. We decided to use React [20] for the implementation of the pages. React is a library for web user interfaces. It provides a convenient abstraction over the elements of web development. Individual parts of the web interface are encapsulated in their components. We used less verbose functional components. The functions create, update, and return web elements. The React library then renders the elements.

We tried to use Create React App (CRA) [11] as our development environment for the client. It worked, but there were flaws. It seems that CRA contains deprecated dependencies, and nobody is maintaining it. We migrated the client to Vite [56]. Vite is next-generation frontend tooling environment. It is better maintained and faster than CRA. It is also very popular. Vite provided friendly development tools, and we were satisfied with it. The client package is managed by NPM. It also manages the dependencies.

The source code of the client has simple hierarchy. The file `main.jsx` is the main entry point. It contains references to other files with the actual pages and other supporting components. We use client side routing with React Router [33]. This approach allowed us to create pages with concise URLs. The router changes the displayed page according to the URL. Users can easily change pages with links. The changes usually do not need to wait for a server response because they are handled in the frontend.

The components in React worked perfectly for our purposes. There are three important pages with three web forms. The forms are used to submit validation requests. They are also used to show details of submitted requests. We created one component for each form. The parameters of the components change the behavior of the form and adjust it for different roles. One component can be used in more pages. Flexible components helped us prevent code duplication. The forms have an easy job. Each form only sends its data to the corresponding URL.

The client can display the content in different languages. We implemented a support for translations. We added translations in Czech and English language. There are tools and libraries for software translation and localization available in NPM, but we decide to avoid them because they introduce unnecessary complexity. We used features of the React library itself. React contains two useful features, i.e. state and context. State contains a value. If the value changes, the components are updated and rendered with the new value. Context is used to create globally accessible values. We used the state to store and change the translations. We used the context to make the translations accessible in the client. The translations are stored in a simple JavaScript object with key-value pairs. The values contain the translations. This implementation is easily extensible. We can always add a new object with new translations and use it in the client.

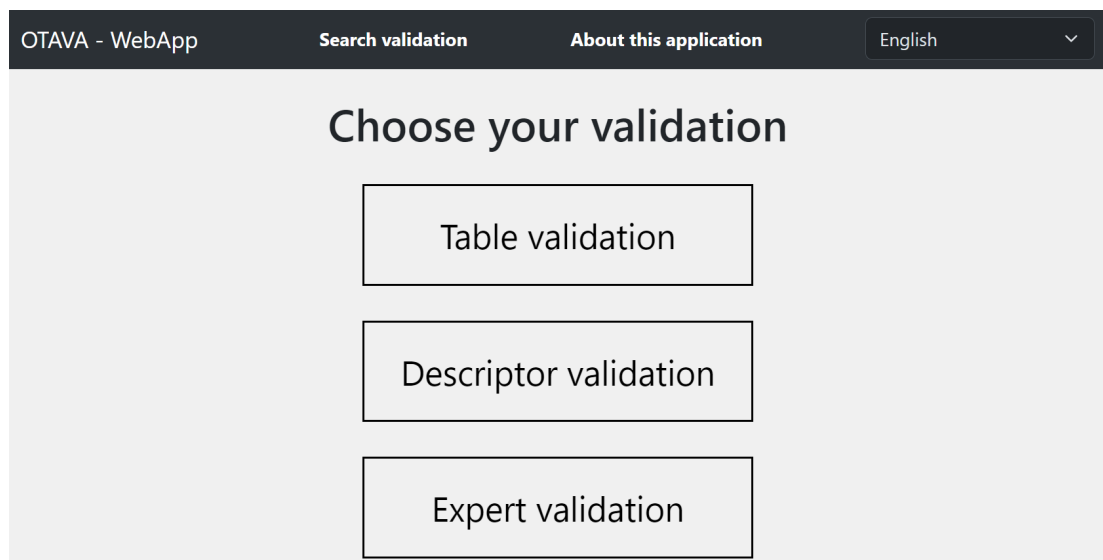


Figure 5.5: The main page of our web application

We did not want to waste precious time creating graphical design. For that reason, we decided to use React Bootstrap [32]. React Bootstrap is the most popular front-end framework for React. It provides many different components

with pretty design and style. We simply picked and used the components in our client. We adjusted some components and placed them in our web pages. React Bootstrap simplified the development and enhanced the appearance of the graphical user interface. Figure 5.5 shows the graphical design of the web application.

The client is loaded from the server to the user's web browser. The client then uses the interface on the same server. Single server provides these two conceptually different services. We did not want to separate them because they are small, simple, and related. The client submits validation requests from individual validation forms. The client also fetches the data from the server. It fetches details about validation requests and validation reports. There may be a delay between the static and dynamic parts of the content. The client displays loading elements in the meantime, and the user knows that the response is on its way. The web interface and the URLs are described in the implementation of the server.

5.4.2 Server

The server is implemented in JavaScript, and it runs in the Node.js [27] environment. Node.js is an asynchronous event-driven JavaScript runtime. It executes JavaScript source code on the server. We decided to use the well known Express [26] library together with Node.js. Express is a fast, unopinionated, minimalist web framework for Node.js. This is probably the most popular combination of technologies on the web nowadays. It is basically a default choice for web development. There is an abundance of developers with the experience.

Express allowed us to easily implement the interface of the server. We added the interface URLs with a few lines of code. Each URL has a handler, i.e. a function which handles the requests. The static content of web pages is served directly from the directory in the client. There are four URLs in the interface. The server interface is tailored to the needs of our client, but it still provides a universal web service. It is very easy to add new URLs to the interface and adapt the server to new clients with different needs. The server is very flexible. The documentation of the web interface and its URLs is in the appendix D. We used Openapi [43] standard because it is an excellent documentation tool. It is significantly better than any possible documentation in the text of this thesis.

The model part of the server uses MySQL 2 [35]. MySQL 2 is a simple and popular JavaScript library in NPM. We used it to connect the model to the database and send queries. The model uses environment variables to access database host and password. These values can be easily set when the application starts. The model communicates directly with the database. It uses prepared statements to prevent SQL injection attacks. The functions in the model are directly used by the URL connection handlers in the server.

5.4.3 Database

We did not implement the database. Such implementation is unnecessary and outside the scope of this thesis. We decided to use the MySQL [28] database. MySQL is a widely used, open-source relational database management system, and it fits our needs. We use the database in a simple way. There is one table

for validation data. The server is responsible for the creation of this table. The following SQL code defines the table.

```
CREATE TABLE IF NOT EXISTS 'validations' (  
  'id' INT UNSIGNED NOT NULL AUTO_INCREMENT ,  
  'language' VARCHAR(20) NOT NULL ,  
  'style' VARCHAR(20) NOT NULL ,  
  'passive-tables' TEXT NOT NULL ,  
  'active-tables' TEXT NOT NULL ,  
  'passive-descriptors' TEXT NOT NULL ,  
  'active-descriptors' TEXT NOT NULL ,  
  'description' TEXT NOT NULL ,  
  'form' VARCHAR(20) NOT NULL ,  
  'request-time' DATETIME NOT NULL ,  
  'finish-time' DATETIME NULL ,  
  'state' VARCHAR(20) NOT NULL ,  
  'outcome-text' TEXT NULL ,  
  'outcome-json' TEXT NULL ,  
  'outcome-turtle' TEXT NULL ,  
  PRIMARY KEY ('id'));
```

We can see that the columns of the table mostly correspond to the user's input in the web forms. The database automatically creates an identifier. There is an additional space with data for the client in the `form` column. It stores the type of the used web form. We store request time and finish time in the database to show the speed of the validation and the workload of the application. The state and the outcomes are used by the worker.

5.4.4 Worker

The worker performs the actual validation for the web application. Its implementation is similar to the command line interface application. It is implemented in Java and therefore it is in the Java part of the repository. It uses the library to perform the validation and a database connector to connect to the database. We implemented the worker in a single Java class. The worker gets the host and the password for the database via environment variables. It reads validation requests from the database and writes validation reports to the database.

The worker uses and changes the state in the database. There are three states, i.e. queueing, working, and finished. New validation requests are queueing and waiting for validation. If the state is "working", the validation of the request is in progress. Finally, the validation is finished. The worker ceaselessly looks for queueing validation requests in the database. It selects the oldest request and changes its state to working. It creates and prepares the validation request for the library and starts the validation. All tables are out of memory to accommodate large tables. When the validation is finished, the worker changes the state in the database and writes outcomes. The implementation of the worker expects that there can be more workers. It prevents race conditions during the access to the database. More workers can work simultaneously and increase the performance of the web application.

5.4.5 Containers

The Java part of this project can be easily compiled, packaged, and executed using Maven. The web application is different. It needs many different technologies installed in one environment. We can install the dependencies, prepare the technologies, and create the environment for the web application on our machine, but this approach is complicated and very time consuming. We decided to simplify the deployment process of the web application with lightweight virtualization.

We used Docker [9]. Docker is lightweight virtualization technology. Our web application runs virtualized in Docker containers. The containers are created from images. An image is basically a file system of a particular virtual machine with additional settings. Container is a running instance of an image. Images are created according to dockerfiles. A dockerfile contains text which describes the creation process of an image. There are two images in our project and one external image. One dockerfile is in the `java-code` directory. It creates an image with the Java part of the project, and it is used to run the worker. The other dockerfile is in the `web-app` directory. It creates an image with the web application part of the project, and it is used to run the sever with the client. The third image contains the database. The database image is downloaded from Docker Hub, an online repository of images. We use Docker Compose to create the three containers from the images and run them. This configuration is in the file `docker-compose.yml`. We can create more containers from the image with the worker and increase the validation performance of the web application.

We utilized containerization to create predictable and repeatable deployment environment for our web application. It simplifies the deployment process. It creates an opportunity for performance adjustment. The web application is scalable. An operator of the system can spawn more containers with the worker and adjust the performance according to the hardware capabilities. Docker is a useful technology in our project.

5.5 Summary

Our implementation is a natural extension of our design. The design is a solid skeleton which holds the implementation. Abstract concepts and principles from the design were implemented with various technologies. We utilized the design benefits, and we did not have to restrict the implementation. Individual parts of the project are free to use the best available technology and language. The parts are properly separated, and they contain clear responsibilities. We used advanced technologies for the implementation itself and also for the deployment. The structure of the web application resembles microservices. Parts can be flexibly deployed on multiple machines. The implementation fulfils the potential of the design, creates a working useful software with unlimited opportunities for possible extensions, and upholds the best practice of software engineering.

6. Documentation

This chapter contains a concise description of the most important parts of the project. This brief description serves as programmer's, user's, and administrator's documentation. Some parts of the documentation are useful for all three roles, and some parts are useful for individual roles. Generally, the part about the library is for programmers, the part about the command line application is for users, and the part about the web application is for administrators. However, the parts are connected, and this division is not very strict. This project has three parts which can be directly used. The library can be used in other software projects. The command line interface application can be used locally on a computer. The web application can be deployed on a server. This documentation does not contain the most specific and the most detailed description of the source code. Additional description is in the source code comments.

6.1 Library

The library contains the core functionality of this project. It performs the validation and other supporting tasks. The library is used via the manager in the class `Manager` in the main package `io.github.janjanda.otava.library`. The manager provides simple public methods for users, and it hides the internal complexity of the library. The names of the methods are self-explanatory, and there are comments with additional instructions. The library is ready for international environment. Users can set a locale in the manager. Users can use implemented locales in the subpackage `locales` or implement their own locale. The default locale is English, and there is also a Czech locale.

6.1.1 Validation principles

There are three styles of the validation: tables only, descriptors only, and full validation. Each style has a corresponding method with comments and descriptions in the manager. The validation itself is implemented in many individual classes. This ensures great extensibility. Each class represent a so-called validation check, and it checks a particular feature of tables and descriptors. Some checks do not make sense if other checks fail. For this reason, a check can depend on other checks. These dependencies create a tree, and the checks in the tree are executed from leaves to the root. If more checks depend on one check, the one check appears on more places in the tree, but it is executed only once for effectiveness. The manager can print the validations trees for each style of the validation. The methods for the validation itself require one parameter with a validation request. A validation request contains documents and other input for the validation, and it has a convenient builder. The validation returns a validation report with results and other data. Each validation check creates one result. The report can be shown in several formats. If a validation cannot be performed, an exception is thrown. The exception can also be shown in several formats just like the report.

6.1.2 Documents

It is important to understand the documents in the library. There are following types of documents: active table, passive table, active descriptor, and passive descriptor. An active table attempts to locate and load its descriptor, a passive table does not. An active descriptor attempts to locate and load its tables, a passive descriptor does not. Every document is either local or online. Every document has a name. A name is an actual location of the document (path in a filesystem or URL). Every document may have an alias. An alias is an alternative name. If an alias is present, it is used in the validation instead of the real name. This is useful when a user wants to validate a table which has not been published yet. The name of the table is the path to the actual file, and the alias is the future URL of the table. The table is located and loaded according to its name, but it pretends that it is on a different location according to the alias. The same thing can be done with a descriptor. Tables are either in memory or out of memory. In-memory tables are loaded completely into memory, out-of-memory tables are not. This can be used to adjust the performance of the library.

6.1.3 Dependency

The library is implemented in Java and managed by Maven [38]. It is very easy to use as a dependency in other Java projects. The other parts of our own project use the library, and they provide examples of its use. It is necessary to include the library in the structure of the other project. Notably, programmers should use the following dependency in the `pom.xml` file. Then, programmers can use the public methods of the library manager according to the comments and descriptions.

```
<dependency>
  <groupId>io.github.janjanda.otava</groupId>
  <artifactId>library</artifactId>
  <version>1.0</version>
</dependency>
```

6.1.4 Validation extension

The validation core of the library can be easily extended with new validation checks. Programmers can improve the validation according to their needs. Here, we show an example of the process. There are many implemented checks in the library, and each one of them is a good example. The process of extension has two steps. We create a new check, and then we use it. It is a very bad idea to include large parts of source code in these narrow pages of paper. We show and describe only the most important parts of the source code. The implemented checks in the library provide complete examples.

First, we create a new class. This new class must extend the `Check` class. We implement the constructor of our new class, like so. Our constructor accepts a check factory as a parameter and calls the constructor of the base class. The constructor of the base class accepts a variable number of parameters. The first parameter contains the tables from the factory. The second parameter contains

the descriptors from the factory. Then, there are zero or more parameters, and they specify pre-check dependencies of our new check. The pre-check instances are created by the factory. The following snippet shows our constructor.

```
public OurNewCheck(CheckFactory f)
throws CheckCreationException {
    super(f.getTables(),
        f.getDescriptors(),
        f.getInstance(RequiredColumnsCheck.class),
        f.getInstance(DataTypesCheck.class),
        f.getInstance(ForeignKeyCheck.class));
}
```

We implement our validation in the method `performValidation` in our new class. We can access and read the tables and descriptors from the base class. They are in the protected fields `tables` and `descriptors` respectively. We implement the behavior of our check and write the results in a result builder. We can use helping utility methods in the library. The following snippet shows simple implementation of the method.

```
@Override
protected Result.Builder performValidation() {
    Result.Builder rb = new Result.Builder(this.getClass().getName());
    rb.addMessage("Hello, world!");
    return rb;
}
```

When our new check is finished, we must add it in the tree of checks. This is the second step of the extension process. We find a good place for our new check according to its meaning and purpose. We find a good parent and add our check to its pre-check dependencies. We add our check to the parent's constructor just like we added the pre-check dependencies in our constructor in the first snippet. The library can print a graphical representation of the tree of checks, and it can help us understand the structure of the tree and the position of our new check. The command line interface application can be used to print the tree. There is more inspiration in the source code of the implemented checks.

6.2 Command line application

The command line application provides user access to the features of the library. It can be compiled and executed respectively with the following commands in the directory `java-code`.

```
mvn package
java -jar cli/target/cli-1.0-jar-with-dependencies.jar -help
```

The option `-help` shows possible options and their syntax with arguments. The names and descriptions of the command line options clearly refer to the corresponding concepts in the library from the previous section.

6.2.1 Examples

The examples have more lines because they are too long for one line. Each example is only one command, and it should be on one line.

Perform full validation with the provided active online descriptor, and write the report as plain text.

```
java -jar cli/target/cli-1.0-jar-with-dependencies.jar
-full -text -aod
https://w3c.github.io/csvw/tests/test011/tree-ops.csv-metadata.json
```

The following examples are executed in the directory with the necessary resources `java-code/library/src/test/resources`.

Perform full validation with the provided passive local in-memory table and the provided passive local descriptor, and write the report as plain text. The table and the descriptor have both name and alias because the links in the descriptor would not be valid otherwise.

```
java -jar ../../../../../../cli/target/cli-1.0-jar-with-dependencies.jar
-full -text -plit tables/table005.csv;https://example.org/tree-ops.csv
-plit metadata/metadata001.json;https://example.org/metadata001.json
```

Perform tables only validation with the provided tables, and write the report as plain text. It is possible to validate multiple documents together.

```
java -jar ../../../../../../cli/target/cli-1.0-jar-with-dependencies.jar
-tables -text -plit tables/table002.csv -plit tables/table013.csv
-plit tables/table014.csv -plit tables/table015.csv
```

6.3 Web application

This section describes the deployment process of the web application in ten easy steps.

1. Make sure that Docker Engine and Docker Compose are running and ready on your machine.
2. Choose a directory for the application.
3. Open your command line in that directory.
4. Run `git clone https://github.com/JanJanda/otava.git` to download the source code of the project. Alternatively, use the repository in the attachment A.1.
5. Use `cd otava` to go to the directory of the project.
6. Edit the downloaded file `docker-compose.yml` like so: Come up with an original password, and replace every occurrence of `my-password` in the file with your password.

7. *Optional:* Increase the `replicas` value under `web_worker` in the file `docker-compose.yml` to spawn more workers and increase validation performance.
8. Run `docker compose up` in the directory `otava` to start everything and wait. It can take a long time. Command line prints "Database is ready" and stops printing other text when the application is ready.
9. Open your web browser and go to the address `localhost` to see the web application.
10. Open another command line in the `otava` directory and run `docker compose down` to stop everything.

The application uses the created directory `database-volume` to store its data. An instruction manual is in the web application. Repeat the steps 7-10 to start the application again. If you want to deploy a new version of the application, delete the two built `otava` images, run `git pull` in the directory `otava` to download the new version from the repository, and then use the steps 7-10.

7. Evaluation

We evaluate the software, this thesis, and the project as a whole in this chapter. We show that individual parts do their tasks, and we evaluate their qualities. We recall our requirements and connect them with our solution. There is a review of the assignment and the solution in this chapter. There are automatic tests and external evaluation criteria. We compare our solution with the other known solutions. This chapter creates a clear picture of the quality in the project.

7.1 Assignment

The assignment states that the goal of this thesis is to implement a validator of CSV files based on the CSV on the Web W3C recommendations. The assignment has several parts and enjoins us to do various things. We must get familiar with CSV on the Web and JSON-LD. We accomplished this part of the assignment in the chapter 2. We must study current implementations. We studied the current implementations in the section 3.2. We must design the validator architecture, so that it is easily extensible with additional validation rules. The subsection 4.1.1 describes the extensible design, and new validation rules can be implemented in validation checks. The solution must have three parts, i.e. Java library, command line runner, and web service runner. The library must have a reasonable subset of validation rules. The solution must represent the validation results in RDF and CSV. We must evaluate the implemented validator on a given set of tests against other implementations. We show these final qualities of the solution later in this chapter.

7.2 Parts of the project

Here, we show some examples of the developed features in the specific parts of the project. The examples show the desired functionality. The solution is clearly divided into required parts, and we show similar functionalities in different parts. The validation library is included in the other parts and therefore it is unnecessary to show it separately. We show the library via developed user interfaces because it cannot be executed alone. Generally, software libraries cannot be executed alone. They are used inside larger projects, and that is their purpose.

7.2.1 Command line interface

There are examples of command line interface application with outputs and descriptions. The examples start in the directory `java-code` in the project repository in the attachment A.1. First, we use `mvn package` to compile the source code and create an executable file. We use `cd cli/target` to change the directory. Then, we perform the validation with the following command.

```
java -jar cli-1.0-jar-with-dependencies.jar -full -text -aod
https://data.mvcr.gov.cz/soubory/číselníky/
stupně-přístupnosti.csv-metadata.json
```

The command should be on one line, but this page is not wide enough. We get the following output.

```
Validation Report
Total duration: 0.734 s

Tables:
https://data.mvcr.gov.cz/soubory/číselníky/stupně-přístupnosti.csv

Descriptors:
https://data.mvcr.gov.cz/soubory/číselníky/
stupně-přístupnosti.csv-metadata.json

Results:
Result
io.github.janjanda.otava.library.checks.BaseUrlsCheck
Duration: 0.0 s
ok

Result
io.github.janjanda.otava.library.checks.ColumnTitlesCheck
Duration: 0.015 s
ok

Result
io.github.janjanda.otava.library.checks.ConsistentColumnsCheck
Duration: 0.0 s
ok

Result
io.github.janjanda.otava.library.checks.ContextCheck
Duration: 0.0 s
ok

Result
io.github.janjanda.otava.library.checks.DataTypesCheck
Duration: 0.0 s
ok

Result
io.github.janjanda.otava.library.checks.ForeignKeyCheck
Duration: 0.0 s
ok

Result
io.github.janjanda.otava.library.checks.FullRootCheck
Duration: 0.0 s
ok

Result
io.github.janjanda.otava.library.checks.LineBreaksCheck
Duration: 0.015 s
```

```

ok

Result
io.github.janjanda.otava.library.checks.PrimaryKeyCheck
Duration: 0.0 s
ok

Result
io.github.janjanda.otava.library.checks.PropsTypesAndValuesCheck
Duration: 0.0 s
ok

Result
io.github.janjanda.otava.library.checks.RequiredColumnsCheck
Duration: 0.0 s
ok

Result
io.github.janjanda.otava.library.checks.RequiredPropsCheck
Duration: 0.0 s
ok

Result
io.github.janjanda.otava.library.checks.TableDescriptorCheck
Duration: 0.0 s
ok

Result
io.github.janjanda.otava.library.checks.TypesCheck
Duration: 0.0 s
ok

Result
io.github.janjanda.otava.library.checks.VirtualsLastCheck
Duration: 0.0 s
ok

```

We can see the validation report. It shows data about the performed validation and results of individual validation checks. Everything is *ok*, and no problems were found.

The following example uses local documents, and it is executed in the directory with the documents `java-code/library/src/test/resources`.

```

java -jar ../../../../cli/target/cli-1.0-jar-with-dependencies.jar
-full -text -plit tables/table005.csv;https://example.org/tree-ops.csv
-pld metadata/metadata001.json;https://example.org/metadata001.json

```

We show only the interesting part of the output for brevity. It shows that a particular column is missing in the table. The column is in the descriptor, but it is not in the table.

```
Result
io.github.janjanda.otava.library.checks.ColumnTitlesCheck
Duration: 0.015 s
fatal
Column inventory_date is not in described table tree-ops.csv
```

The following example shows more documents. There are two tables and two descriptors. One descriptor specifies a foreign key between tables, but the tables violate the foreign key.

```
java -jar ../../../../cli/target/cli-1.0-jar-with-dependencies.jar
-full -text
-plit tables/table020.csv;http://example.org/tree-ops.csv
-plit tables/table019.csv;http://example.org/reference.csv
-pld metadata/metadata022.json;http://example.org/metadata022.json
-pld metadata/metadata023.json;http://example.org/metadata023.json
```

There is the interesting part of the output. The duration of the check is 0.0 s because the time is too short. The real time is below rounding error.

```
Result
io.github.janjanda.otava.library.checks.ForeignKeyCheck
Duration: 0.0 s
fatal
Foreign key violated in the table tree-ops.csv because values are
missing in the referenced table http://example.org/reference.csv
```

We already know from the text of this thesis that there are other options for documents, languages, validation styles, and output formats. We do not show these options here because they are analogous, and they do not bring any extra interesting value to this evaluation. All the options would make this text long and boring.

7.2.2 Web application

We show similar examples in the context of the web application. We use the web application to validate the files. First, we deploy the web application according to the documentation in the section 6.3. Then, we use the graphical user interface to submit our validation request. Naturally, the results of validations are the same as in the case with the command line interface. The purpose of the examples is to show the web application. Figure 7.1 shows the web application with a descriptor validation. We can see that the report is available in three formats below. Figure 7.2 shows the web application with an expert validation. The web application works well and creates same results as the command line interface application.

OTAVA - WebApp Search validation About this application English

Validation 29

The validation is finished.

Request details - Descriptor validation

Request time: 11/4/2023, 2:54:58 PM

Descriptor URL

`https://data.mvcr.gov.cz/soubory/číselníky/stupně-přístupnosti.csv-metadata.json`

Find and use tables from the descriptor

Validation result language

English

Czech

Validation report

Finish time: 11/4/2023, 2:55:02 PM

- Text
- JSON
- Turtle

Figure 7.1: Web application with a descriptor validation

OTAVA - WebApp Search validation About this application English

Validation 30

The validation is finished.

Request details - Expert validation

Request time: 11/4/2023, 3:04:13 PM

Validation style

Full validation

Tables only validation

Descriptors only validation

Passive tables

`https://raw.githubusercontent.com/JanJanda/otava/master/java-code/library/src/test/resources/tables/table020.csv http://example.org/tree-ops.csv`
`https://raw.githubusercontent.com/JanJanda/otava/master/java-code/library/src/test/resources/tables/table019.csv http://example.org/reference.csv`

Active tables

Passive descriptors

`https://raw.githubusercontent.com/JanJanda/otava/master/java-code/library/src/test/resources/metadata/metadata022.json http://example.org/metadata022.json`
`https://raw.githubusercontent.com/JanJanda/otava/master/java-code/library/src/test/resources/metadata/metadata023.json http://example.org/metadata023.json`

Figure 7.2: Web application with an expert validation

7.2.3 Summary

The project has the three required parts, i.e. Java library, command line runner, and web service runner. It is clearly visible from the design in this thesis and this evaluation examples. Users interact with the command line interface application and the web application. These two parts use the library to perform the validation. The library is the third part. The solution must represent the validation results in RDF and CSV. Our solution can represent the validation results in three formats, i.e. plain text, Turtle, and JSON. Plain text is good for humans. JSON is a popular machine-readable format. Turtle is an RDF format. Our solution cannot represent the validation results in CSV because it does not make sense. The results do not have the shape of a table. It is practically impossible to force the results into a tabular format because there are individual checks with different number of messages and other complex data. We would need more connected tables or one denormalized table with some duplicate values, and it would look horrible in any case. This particular part of the assignment is unfeasible, but there are better formats and a better solution. The other parts of the assignment were accomplished.

7.3 Requirements

In this section, we show how we fulfilled the requirements from the section 3.1. The requirements comprise a more detailed assignment of this thesis. The requirements are fulfilled in different parts of the project. Some concepts influence the entire project. Some requirements are explicitly fulfilled in particular parts of the project. Other requirements are implicitly fulfilled. We show the qualities and their connections to the concepts and requirements.

We decided to create a beginner friendly software with advanced features for experts. We anticipated different kinds of users with different levels of knowledge and experience. Our solution successfully provides an interface for different kinds of users with different skills. The solution can adjust its complexity almost continuously. We divided individual concepts into practically independent parts of the interface. Users can choose and use their favorite features, and they do not need to understand the advanced features. For example, an advanced user may enter a document with a name and an alias, but a beginner simply uses only a name because a beginner does not understand aliases. If an advanced feature of the interface is not used, it is invisible and does not complicate the interface. An advanced user may use the options with active documents, but a beginner simply ignores these options and uses only the options with passive documents. Users can customize their experience according to their skills and needs. This important concept is in every part of the project. It is in the API of the library. It is in the options of the command line interface application. The web application even uses different pages to enhance its graphical interface.

The solution fulfils the stated functional requirements. The concept of active and passive documents fulfils the requirement F7. The application attempts to find the required files for active documents. The concept of aliases fulfils the requirements F8 and F9. The aliases are used to establish and override links between documents. There are partial validations for the requirements F10

and F11. The partial validations validate only the selected types of files. The other functional requirements are fulfilled obviously. Specific groups of functional requirements are represented by the use cases. The functional requirements are fulfilled and therefore the use cases are prepared and achieved. The quality requirements are also fulfilled. It follows from the design and implementation. The tests in the requirement Q9 are demonstrated later in this chapter. We used the MIT license to fulfil the requirement Q13 because it is a common choice. The solution completely achieves the analysed formal goal of this thesis.

7.4 Comparison

In this section, we compare our project with the best known competing project. The competing project is the csvw-validator from the subsection 3.2.1. We inspect the project and evaluate its advantages and disadvantages. We use our personal judgment and opinions. We also use validation tests as an external criterion. It is a difficult task to compare the projects, because some aspects are a matter of personal preference. Objective criteria may have a misleading meaning, and they may not correspond with the real use of the project. We create a picture of the two projects next to each other.

We used the validation tests from the CSVW Implementation Report [17]. The test suite contains tests for validators. The tests have specified documents for validation and expect a particular result of the validation. The tests provide an objective criterion for evaluation, but we must be careful with the meaning of the tests. The tests uniformly cover some expected functionality of a validator. Very common use cases are represented by roughly the same number of tests as rare and unusual use cases. This distribution creates an interesting relationship. If a validator passes 20 % of the tests, it can be useful in 80 % of real-life situations. Naturally, this is just our simple approximate heuristic. Some of our features are not tested by any prescribed test. Some tested features are bizarre, and users probably never use them, e.g. complex formatting patterns for numbers. The capabilities of our validator are not a subset of the tests, they intersect the tests.

The results of the tests are in the appendix E. We achieved our target and implemented our validator with the success rate 201/281. This success rate corresponds with the requirement "reasonable subset of validation rules" from the assignment. The target success rate was chosen heuristically, and it represents a good functionality with enough space for flexibility and future extensions. The significance of the success rate is not very big. It simply shows that the validator can be practically used. Our validator can be extended to increase the success rate, but we did not do it because the benefit was not worth the time anymore, and we focused on other parts. We think that the failed test represent very rare real-world use cases, because they contain uncommon data types and complex features. The competing csvw-validator claims to have the success rate 262/281. We have a similar success rate because the difference between the success rates is in the uncommon real-world use cases, and it has a little practical significance.

The competing csvw-validator was created according to almost identical assignment, but the solution is very different. It did not solve the problem in full generality, but it solved only a special situation. It can validate one table and one descriptor, and it lacks the advanced features, i.e. reassign files together

and override default behaviors. Consequently, it is difficult to use, and it cannot validate relations between multiple tables. Figure 7.3 shows the command line interface options of the csvw-validator. We can see Figure 5.4 with our options and understand the glaring difference in complexity and capabilities. The competing project does not have the support for multiple validated documents, aliases, active and passive documents, in-memory and out-of-memory tables, etc. We devised these innovative concepts and implemented them in our trailblazing solution. We solved the assignment in full generality and created our groundbreaking project.

We accomplished every aspect from Table 3.2. The other known competing projects have significant flaws, and they lag behind our next-generation project. Our project is a serious contribution to the world of open source software. It simplifies the validation and provides needed flexibility with a friendly interface. It is well documented, the deployment is trouble-free, and the source code is as simple as possible. It stands out above other comparable projects.

```
java -jar csvw-validator-cli-app-1.0.0-SNAPSHOT.jar [-f <FILE>] [-s <SCHEMA>]
[-o <OUTPUT>] [--strict] [-h] [--rdf] [--csv]

-f,--file <FILE>           The CSV file URL to be processed
-s,--schema <SCHEMA>      The CSVW schema URL to be processed
-o,--output <OUTPUT>      The name of output file without suffix
  --not-strict             Disables strict mode
  --csv                    Validator will generate output also in CSV format.
  --rdf                    Validator will generate output also in RDF format
-h,--help                  Show help
```

Figure 7.3: Command line interface options of the csvw-validator project

7.5 Performance

We measured the performance of our solution. Specifically, we measured the time of validation for different input data and different configurations. It is interesting to know the time of validation duration. This section describes the process and the results. This performance test shows the true meaning of some design decisions in this thesis. Notably, we see the difference between in-memory and out-of-memory tables. We test the impact of different descriptors. This section shows the times and the general complexity of the problem.

We used the command line interface application for our tests. The application performed full validation. The application generated validation reports in JSON. The JSON format helped us automatically collect total duration times from the reports. This format is a useful feature in our project. There were two documents in every validation, i.e. a passive local table and a passive local descriptor. Naturally, the descriptor matched the table every time, and validations did not find any problems. We focused on the performance, not the validation report. We performed the tests with in-memory and out-of-memory configuration for the table. We performed the tests with different number of rows in the table. We performed the tests with two descriptors. One descriptor is simple, and the other descriptor has a primary key. The two descriptors are almost identical. The only difference is the primary key. The primary key descriptor has the primary key,

and the simple descriptor does not have it. The following snippet shows the primary key descriptor. We repeated every test ten times on an ordinary desktop computer.

```
{
  "@context": "http://www.w3.org/ns/csvw",
  "url": "table.csv",
  "tableSchema": {
    "columns": [{
      "name": "A",
      "titles": "A",
      "datatype": "integer",
      "required": true
    }, {
      "name": "B",
      "titles": "B",
      "required": true
    }, {
      "name": "C",
      "titles": "C"
    }
  ],
  "primaryKey": "A"
}
```

We used the repeated tests to calculate mean time of validation for every test. The following tables contain 50 results. It means that we performed 500 validations, and we collected 500 data points. Table 7.1 shows the results with the simple descriptor. Table 7.2 shows the results with the primary key descriptor. The tables show the results in relation to the memory configuration and the number of rows in the validated table. The tables show mean times of validation. The following scatter plots show every measurement. They show together all 500 data points. Figure 7.4 corresponds to Table 7.1, and Figure 7.5 corresponds to Table 7.2. We also measured the maximum heap memory usage in the validation of the table with 10^7 rows. The out-of-memory configuration used 141 MB, and in-memory configuration used 3404 MB.

The results show expected values, but they are still interesting. We can see that a primary key significantly increases validation time. The values in the primary key must be unique. The check of uniqueness has quadratic time complexity, and possible optimizations use precious memory. If a primary key is not specified, it does not waste time. We can see that validations of in-memory tables are faster than validations of out-of-memory tables. This effect is very strong with the primary key descriptor because the primary key check performs many more read operations with the table. We used local tables in our measurements. On-line tables can increase the difference between the in-memory and out-of-memory configuration. We did not measure these results because unpredictable internet connection makes them unreliable, and it is difficult to upload large files to the internet.

Number of rows $\times 10^6$	In-memory	Out-of-memory
1	2.77	2.92
2	4.89	5.33
3	6.83	8.01
4	8.15	10.49
5	9.47	12.89
6	11.15	15.49
7	13.24	18.12
8	15.06	20.10
9	17.69	22.87
10	19.09	25.28

Table 7.1: Mean time of validation with the simple descriptor in seconds

Number of rows $\times 10^3$	In-memory	Out-of-memory
1	0.29	1.01
2	0.32	2.63
3	0.39	5.43
4	0.45	9.16
5	0.57	14.24
6	0.69	20.24
7	0.84	27.18
8	1.00	35.32
9	1.30	44.37
10	1.34	54.87
11	1.45	
12	1.66	
13	1.90	
14	2.14	
15	2.39	
16	2.66	
17	2.99	
18	3.32	
19	3.65	
20	3.94	

Table 7.2: Mean time of validation with the primary key descriptor in seconds

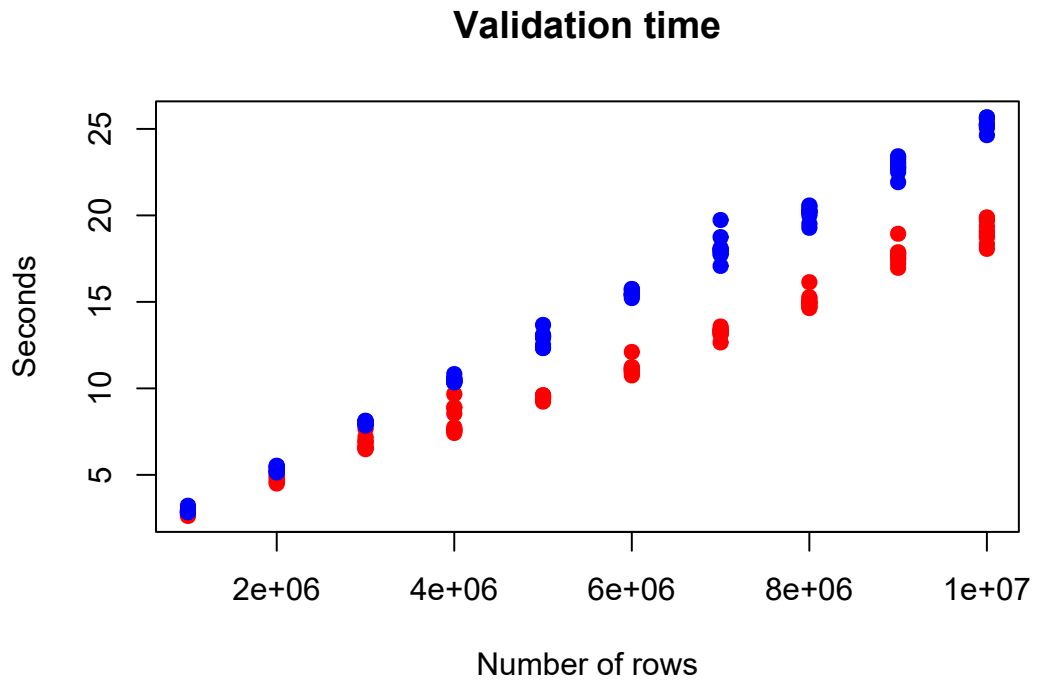


Figure 7.4: Validation time with the simple descriptor, in-memory is red and out-of-memory is blue

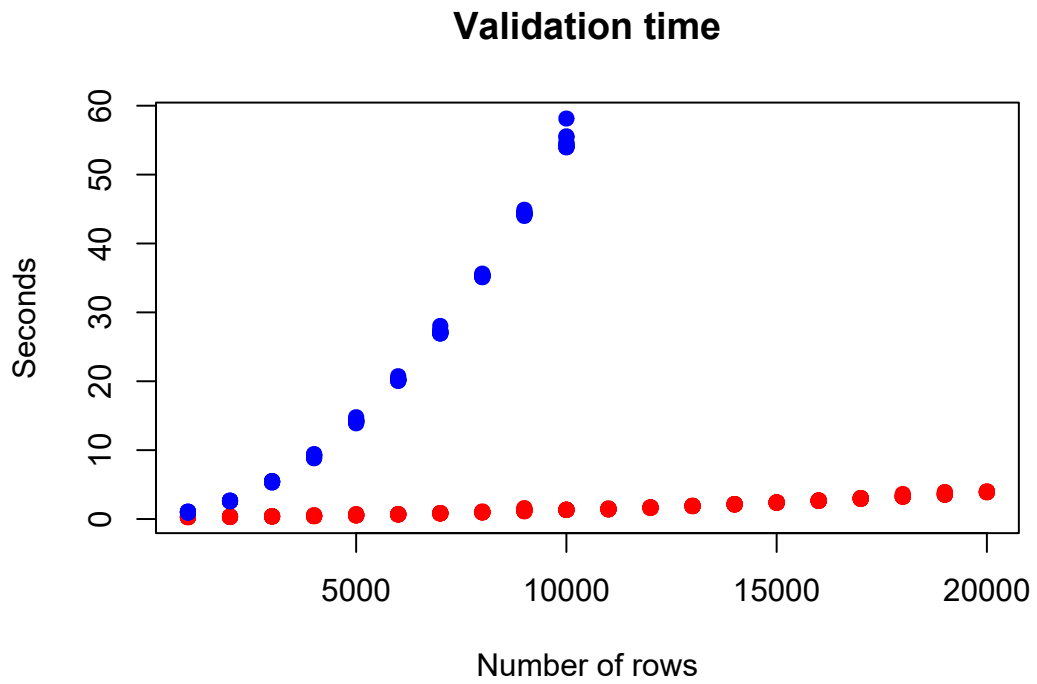


Figure 7.5: Validation time with the primary key descriptor, in-memory is red and out-of-memory is blue

This section shows the practical significance of our project. It was practically used many times with different input data and different configurations. This process unwittingly tested the functionality. We measured the time of validation duration in various circumstances. The results show the impact of our design decisions and the implementation. We see the importance of in-memory and out-of-memory tables. We understand that validations of large tables are not instantaneous. We also see that it was a good idea to separate the validation in the web application with a queue because the validation may take some time, and it cannot block the web server. Our solution has a good performance. It is able to validate large tables, and it can be practically used.

7.6 Summary

We evaluated our solution from several different perspectives. This evaluation shows the qualities and features in our solution. First, we focused on the formal assignment of this thesis. We described the solution with respect to the assignment. This evaluation also fulfils some requirements from the assignment. We evaluated individual parts of the solution and their roles. Each part has its unique and useful interface. There are practical examples in the evaluation. We recalled our requirements from the analysis and showed how we fulfilled them. We compared our solution with other projects and presented the value of our solution in a competitive environment. There are external criteria together with our judgement. Finally, we measured the performance of our solution and saw how powerful it is in various situations. This section thoroughly describes our accomplishments and the successful fulfilment of this thesis and its assignment.

8. Conclusion

In this thesis, we developed a CSV file validator according to the CSV on the Web W3C recommendations. We described the entire process of software development. We started with our technical standards and specifications because they define the domain of this thesis. Then, we analysed the situation, the assignment and the requirements. The analysis thoroughly specified the goal of this thesis. We gradually created a solution. The solution is a software project. The design describes the main principles of our solution. The implementation describes technologies, source code, and the software itself. We created a documentation for different roles of users. It concisely familiarizes new users with our work. The evaluation describes how we accomplished our goal and fulfilled our requirements.

The main value of this thesis follows from the standards. There are several complex technical standards and intricate recommendations. This thesis successfully brings them together and uses them to create the practical software. We managed the considerable complexity, combined the previous work of countless experts, and created an original contribution. We used various programming languages and technologies during the implementation. We learned how to use them and connect them. We chose the best tools for every particular purpose. This thesis is a sophisticated combination of many different things. It orchestrates them to accomplish its goal.

This thesis is a contribution to the worlds of data formats, open data, and open source software. These domains are usually poorly funded, and they rely on the enthusiasm of their communities. Despite the lack of funds, contributors are able to create respectable innovative solutions and advance entire fields of technology. This thesis performs practical tasks, but it also upholds greater principles. Users can conveniently validate CSV files and publish their data with higher quality. The data can be then transformed and linked. This has immense benefits and practical applications. People can easily explore the data and search connections. Connected data reveal hidden and unforeseen patterns. Possible uses go beyond imagination.

The developed project has many different powerful capabilities. It provides convenient user interfaces for various tasks. It performs the validation and offers other supporting features. Even though the project is very extensive, it still has a potential for extensions and improvements. Software projects generally cannot ever be finished because there are always new situations and requirements. Our project is ready for the unpredictable future. We created every little part of it with significant flexibility and extensibility. Developers can adjust the functionality with minimum effort in the future. We confidently expect long life of our project.

We diligently developed every part of the project and wrote every chapter of this text. Best practices are visible everywhere in the design and the implementation. We were not afraid of any interesting approach or design pattern. We were free to choose best technologies and integrate them in our implementation thanks to our design. The evaluation clearly shows the developed capability and usability. We fulfilled the assignment of this thesis, our goal, and every requirement. We can successfully conclude this thesis with pride.

Bibliography

- [1] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. RDF 1.1 Turtle, 2014. URL <https://www.w3.org/TR/2014/REC-turtle-20140225/>. [Online; accessed 4-June-2022].
- [2] Tim Berners-Lee. 5-star OPEN DATA, 2015. URL <https://5stardata.info>. [Online; accessed 5-May-2022].
- [3] Tim Berners-Lee. World Wide Web Consortium, 2022. URL <https://www.w3.org/>. [Online; accessed 4-June-2022].
- [4] Simon Brown. The C4 model for visualising software architecture, 2023. URL <https://c4model.com/>. [Online; accessed 19-October-2023].
- [5] Douglas Crockford and Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format, 2017. URL <https://datatracker.ietf.org/doc/html/rfc8259>. [Online; accessed 7-June-2022].
- [6] DCMI Usage Board. DCMI Metadata Terms, 2020. URL <https://www.dublincore.org/specifications/dublin-core/dcmi-terms/>. [Online; accessed 3-June-2022].
- [7] Digital Enterprise Research Institute. Prefix.cc, 2022. URL <https://prefix.cc/>. [Online; accessed 5-June-2022].
- [8] Directorate-General for Communications Networks, Content and Technology. European legislation on open data, 2022. URL <https://digital-strategy.ec.europa.eu/en/policies/legislation-open-data>. [Online; accessed 15-May-2022].
- [9] Docker Inc. Docker, 2023. URL <https://www.docker.com/>. [Online; accessed 30-October-2023].
- [10] Martin Dürst and Michel Suignard. Internationalized Resource Identifiers (IRIs), 2005. URL <https://datatracker.ietf.org/doc/html/rfc3987>. [Online; accessed 2-June-2022].
- [11] Facebook Open Source. Create React App, 2023. URL <https://create-react-app.dev/>. [Online; accessed 27-October-2023].
- [12] FasterXML. Jackson, 2023. URL <https://github.com/FasterXML/jackson>. [Online; accessed 5-October-2023].
- [13] Government Digital Service. Find open data, 2022. URL <https://data.gov.uk/>. [Online; accessed 15-May-2022].
- [14] Stuart Harrison et al. CSV Lint, 2022. URL <https://github.com/Data-Liberation-Front/csvlint.rb>. [Online; accessed 15-August-2022].
- [15] Jan Janda. OTAVA, 2023. URL <https://github.com/JanJanda/otava>. [Online; accessed 1-October-2023].

- [16] JSON-LD Community. JSON-LD Playground, 2022. URL <https://json-ld.org/playground/>. [Online; accessed 7-June-2022].
- [17] Gregg Kellogg et al. CSVW Implementation Report, 2015. URL <https://w3c.github.io/csvw/tests/reports/index.html>. [Online; accessed 17-August-2022].
- [18] Gregg Kellogg et al. RDF::Tabular, 2022. URL <https://github.com/ruby-rdf/rdf-tabular>. [Online; accessed 20-July-2022].
- [19] Vojtěch Malý. Validátor CSV souborů dle W3C doporučení CSV on the Web. Master thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, 2019. URL <http://hdl.handle.net/10467/82671>.
- [20] Meta Open Source. React, 2023. URL <https://react.dev/>. [Online; accessed 27-October-2023].
- [21] NPM Contributors. NPM, 2023. URL <https://www.npmjs.com/>. [Online; accessed 26-October-2023].
- [22] Office of Information Policy U.S. Department of Justice. Freedom of Information Act, 2022. URL <https://www.foia.gov/>. [Online; accessed 15-May-2022].
- [23] Open Data Institute. CSV Lint IO, 2022. URL <http://csvlint.io/>. [Online; accessed 16-August-2022].
- [24] Open Data Institute. About the ODI, 2022. URL <https://theodi.org/about-the-odi/>. [Online; accessed 15-May-2022].
- [25] Open Source. H2 Database Engine, 2022. URL <https://www.h2database.com/html/main.html>. [Online; accessed 12-July-2022].
- [26] OpenJS Foundation. Express, 2023. URL <https://expressjs.com/>. [Online; accessed 28-October-2023].
- [27] OpenJS Foundation. Node.js, 2023. URL <https://nodejs.org/en>. [Online; accessed 28-October-2023].
- [28] Oracle. MySQL, 2023. URL <https://www.mysql.com/>. [Online; accessed 30-October-2023].
- [29] Otevřená data. Portál otevřených dat, 2022. URL <https://data.gov.cz/>. [Online; accessed 15-May-2022].
- [30] Otevřená data. Legislativní prostředí otevřených dat, 2022. URL <https://opendata.gov.cz/legislativa:start>. [Online; accessed 15-May-2022].
- [31] Rufus Pollock, Jeni Tennison, Gregg Kellogg, and Ivan Herman. Metadata Vocabulary for Tabular Data, 2015. URL <https://www.w3.org/TR/2015/REC-tabular-metadata-20151217/>. [Online; accessed 21-June-2022].
- [32] React Bootstrap Contributors. React Bootstrap, 2023. URL <https://react-bootstrap.netlify.app/>. [Online; accessed 27-October-2023].

- [33] Remix Software, Inc. React Router, 2023. URL <https://reactrouter.com/en/main>. [Online; accessed 27-October-2023].
- [34] Yakov Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) Files, 2005. URL <https://datatracker.ietf.org/doc/html/rfc4180>. [Online; accessed 7-December-2023].
- [35] Andrey Sidorov. MySQL 2, 2023. URL <https://github.com/sidorares/node-mysql2>. [Online; accessed 29-October-2023].
- [36] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, Pierre-Antoine Champin, and Niklas Lindström. JSON-LD 1.1, 2020. URL <https://www.w3.org/TR/2020/REC-json-ld11-20200716/>. [Online; accessed 6-June-2022].
- [37] Jeni Tennison, Gregg Kellogg, and Ivan Herman. Model for Tabular Data and Metadata on the Web, 2015. URL <https://www.w3.org/TR/2015/REC-tabular-data-model-20151217/>. [Online; accessed 21-June-2022].
- [38] The Apache Software Foundation. Apache Maven, 2022. URL <https://maven.apache.org/>. [Online; accessed 12-July-2022].
- [39] The Apache Software Foundation. Apache Commons CLI, 2023. URL <https://commons.apache.org/proper/commons-cli/>. [Online; accessed 26-October-2023].
- [40] The Apache Software Foundation. Apache Commons CSV, 2023. URL <https://commons.apache.org/proper/commons-csv/>. [Online; accessed 5-October-2023].
- [41] The Apache Software Foundation. Apache Jena, 2023. URL <https://jena.apache.org/>. [Online; accessed 5-October-2023].
- [42] The JUnit Team. JUnit, 2023. URL <https://junit.org/junit5/>. [Online; accessed 26-October-2023].
- [43] The Linux Foundation. Openapi, 2023. URL <https://www.openapis.org/>. [Online; accessed 29-October-2023].
- [44] The Unicode Consortium. Unicode, 2022. URL <https://home.unicode.org/>. [Online; accessed 4-June-2022].
- [45] Štěpán Stenclák, Jan Janda, et al. Dataspecer, 2023. URL <https://github.com/mff-uk/dataspecer>. [Online; accessed 18-October-2023].
- [46] U.S. General Services Administration, Technology Transformation Service. Data.gov, 2022. URL <https://data.gov/>. [Online; accessed 15-May-2022].
- [47] Vaadin Ltd. Vaadin, 2022. URL <https://vaadin.com/>. [Online; accessed 12-July-2022].
- [48] VMware, Inc. Spring Boot, 2022. URL <https://spring.io/projects/spring-boot>. [Online; accessed 12-July-2022].

- [49] W3C Working Group. W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes, 2012. URL <https://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/>. [Online; accessed 26-June-2022].
- [50] W3C Working Group. Generating JSON from Tabular Data on the Web, 2015. URL <https://www.w3.org/TR/2015/REC-csv2json-20151217/>. [Online; accessed 1-July-2022].
- [51] W3C Working Group. Generating RDF from Tabular Data on the Web, 2015. URL <https://www.w3.org/TR/2015/REC-csv2rdf-20151217/>. [Online; accessed 28-June-2022].
- [52] W3C Working Group. CSV on the Web: A Primer, 2016. URL <https://www.w3.org/TR/2016/NOTE-tabular-data-primer-20160225/>. [Online; accessed 5-May-2022].
- [53] Wikipedia contributors. ASCII — Wikipedia, The Free Encyclopedia, 2022. URL <https://en.wikipedia.org/w/index.php?title=ASCII&oldid=1083442349>. [Online; accessed 5-May-2022].
- [54] Wikipedia contributors. UTF-8 — Wikipedia, The Free Encyclopedia, 2022. URL <https://en.wikipedia.org/w/index.php?title=UTF-8&oldid=1086175131>. [Online; accessed 5-May-2022].
- [55] World Wide Web Consortium. RDF 1.1 Concepts and Abstract Syntax, 2014. URL <https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/>. [Online; accessed 6-May-2022].
- [56] Evan You et al. Vite, 2023. URL <https://vitejs.dev/>. [Online; accessed 27-October-2023].

List of Figures

2.1	Example of an RDF triple	5
2.2	Example of an RDF graph with a blank node	8
3.1	Diagram of the anticipated users	17
3.2	Use case diagram of the system	21
3.3	Web application interface of the csvw-validator	22
3.4	Web application interface of the CSV Lint IO	24
4.1	Mock table validation form	34
4.2	Mock expert validation form	35
5.1	Generated graphical representation of the checks tree	41
5.2	Class diagram of the documents, gray interface is from <code>java.lang</code>	44
5.3	Simplified sequence diagram of the user's interaction with the library	47
5.4	Command line interface with the options	50
5.5	The main page of our web application	52
7.1	Web application with a descriptor validation	65
7.2	Web application with an expert validation	65
7.3	Command line interface options of the csvw-validator project	68
7.4	Validation time with the simple descriptor, in-memory is red and out-of-memory is blue	71
7.5	Validation time with the primary key descriptor, in-memory is red and out-of-memory is blue	71
A.1	Directory structure of the repository	80
C.1	System context diagram of the web application	84
C.2	Containers diagram of the web application	85
C.3	Components diagram of the web client	86
C.4	Components diagram of the web server	87
C.5	Components diagram of the web worker	88

List of Tables

3.1	Evaluated aspects of the solutions	27
3.2	Aspects of the proposed solution	27
7.1	Mean time of validation with the simple descriptor in seconds . .	70
7.2	Mean time of validation with the primary key descriptor in seconds	70

A. Attachments

This chapter lists the attachments of this thesis. The attachments cannot technically be included directly in this text because their formats are not suitable. The attachments are externally attached to this text, but they are still inseparable parts of this thesis.

A.1 Project repository

The Git repository of the project contains the developed software of the project and the development history. There are files with the source code, brief documentation, etc. It is also available from <https://github.com/JanJanda/otava>. Figure A.1 shows the directories of the repository. Our entire implementation is in this repository.



Figure A.1: Directory structure of the repository

B. Example of JSON-LD 1.1

There is the entire comprehensive JSON-LD example. We can use JSON-LD Playground [16] to explore it.

```
{
  "@context": {
    "ex": "http://example.org/",
    "dcterms": "http://purl.org/dc/terms/",
    "schema": "http://schema.org/",
    "submitted": {
      "@id": "dcterms:dateSubmitted",
      "@type": "http://www.w3.org/2001/XMLSchema#date"
    },
    "changed": {
      "@id": "dcterms:modified",
      "@type": "http://www.w3.org/2001/XMLSchema#date"
    },
    "title": {
      "@id": "dcterms:title",
      "@container": "@language"
    }
  },
  "@id": "ex:thesis123",
  "@type": "schema:Thesis",
  "submitted": "2021-07-20",
  "changed": [ "2021-07-18", "2021-07-15", "2021-07-10" ],
  "ex:reader": {
    "ex:rank": "professor",
    "ex:name": "John Doe"
  },
  "title": {
    "en": "Data monitoring",
    "cs": "Monitorování dat"
  },
  "dcterms:format": [
    {
      "@value": "paper",
      "@language": "en"
    },
    {
      "@value": "papír",
      "@language": "cs"
    }
  ],
  "ex:ready": true,
  "ex:copies": 3
}
```

C. Web application diagram

We created the diagram of the web application in the C4 model [4]. There is the source code of the diagram. The diagram can be effortlessly rendered and viewed with Structurizr (<https://structurizr.com/dsl>). Rendered diagrams are below. Everything is also included in the repository in the attachment A.1 in the directory `web-app/diagram`.

```
workspace "OTAVA" {
  model {
    beginner = person "Beginner"
    expert = person "Expert"

    webApp = softwareSystem "Web Application" {
      client = container "Client" "" "React" {
        tableValPage = component "Table Validation" "" "" "page"
        descValPage = component "Descriptor Validation" "" "" "page"
        expertValPage = component "Expert Validation" "" "" "page"
      }
      server = container "Server" "" "Node.js" {
        router = component "Router"
        model = component "Model"
      }
      database = container "Database" "" "MySQL" "database"
      worker = container "Worker" "" "Java" {
        dbConn = component "Database Connector"
        lib = component "Validation Library"
      }
    }

    beginner -> tableValPage "Uses"
    beginner -> descValPage "Uses"
    expert -> tableValPage "Uses"
    expert -> descValPage "Uses"
    expert -> expertValPage "Uses"

    client -> router "Communicates"
    router -> model "Uses"
    model -> database "Reads & Writes"

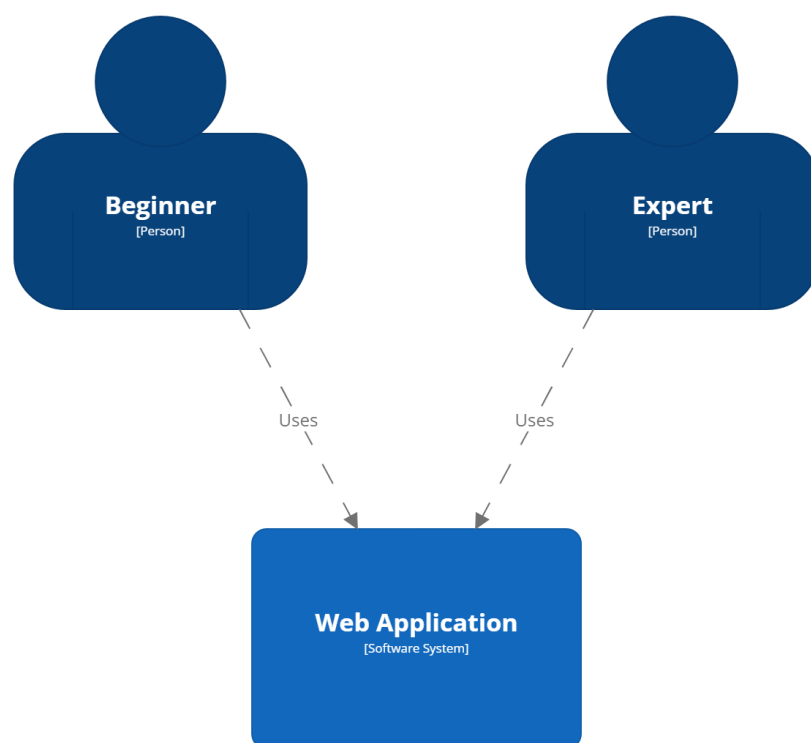
    dbConn -> database "Reads & Writes"
    dbConn -> lib "Uses"
  }

  views {
    systemContext webApp {
      include *
      autoLayout
    }
    container webApp {
```

```
    include *
    autoLayout
  }
  component client {
    include *
    autoLayout
  }
  component server {
    include *
    autoLayout
  }
  component worker {
    include *
    autoLayout
  }

  styles {
    element page {
      shape WebBrowser
    }
    element database {
      shape Cylinder
    }
  }

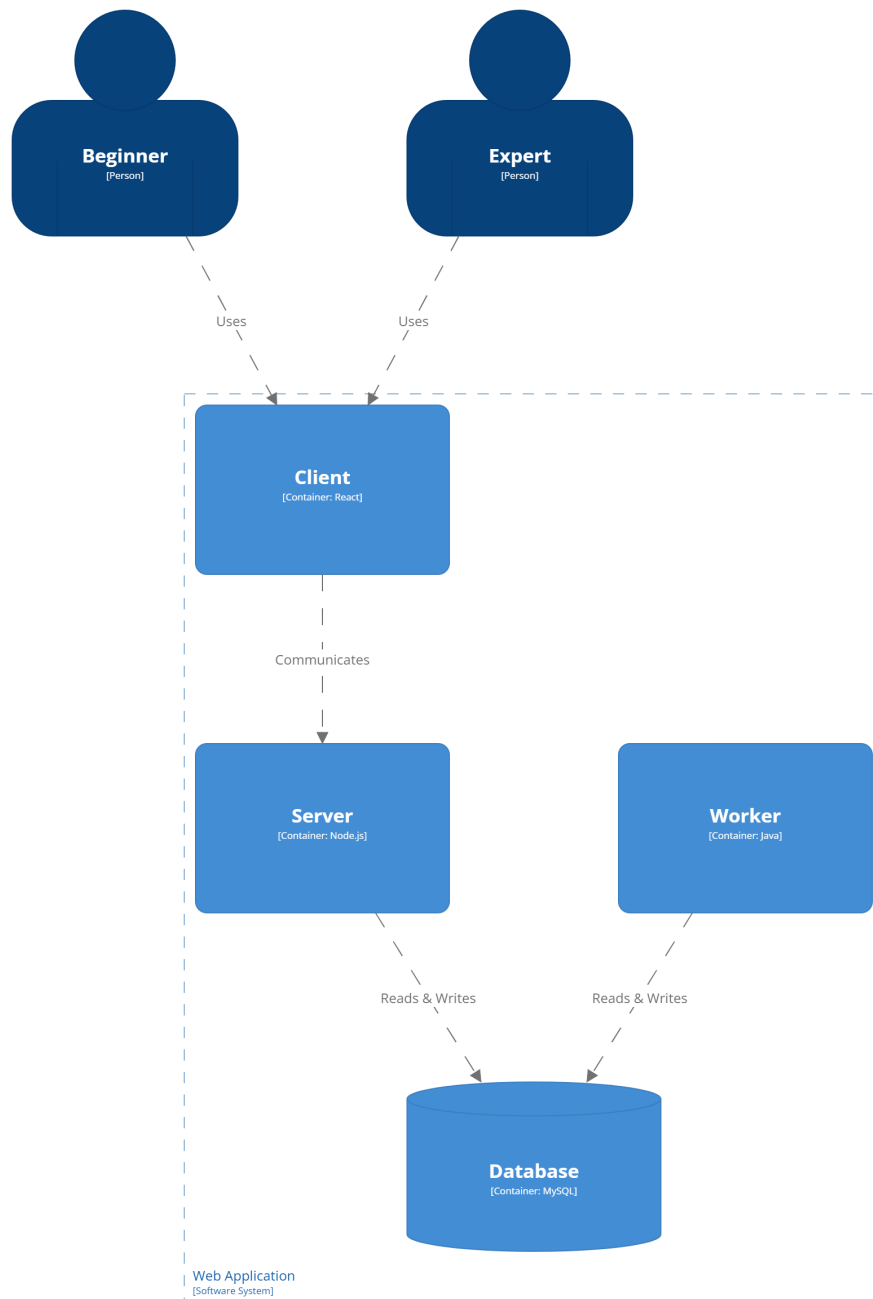
  theme default
}
```

[System Context] Web Application

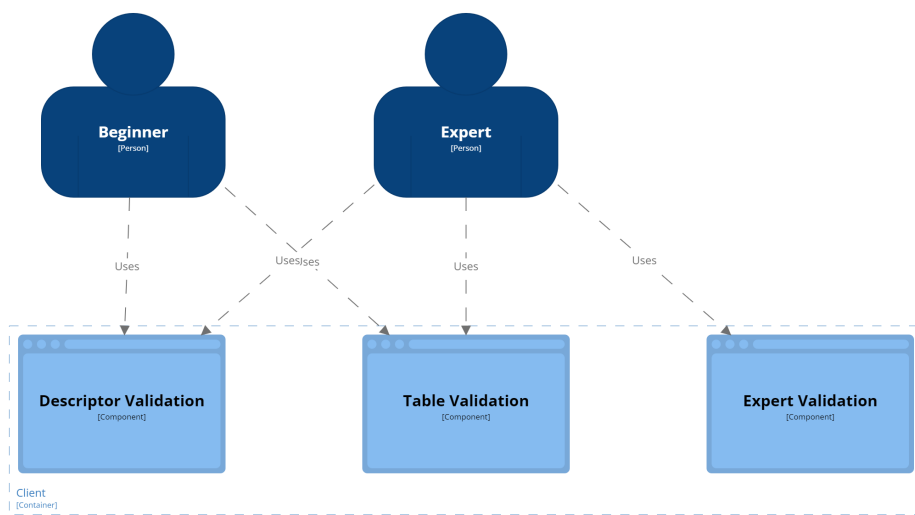
pátek 20. října 2023 v 10:42 středoevropský letní čas

Figure C.1: System context diagram of the web application



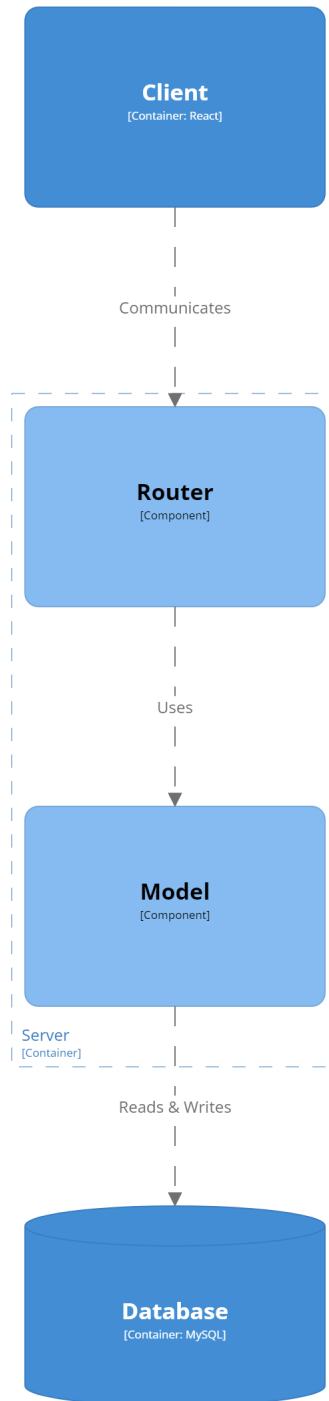
[Container] Web Application
 pátek 20. října 2023 v 10:42 středoevropský letní čas

Figure C.2: Containers diagram of the web application



[Component] Web Application - Client
 pátek 20. října 2023 v 10:42 středoevropský letní čas

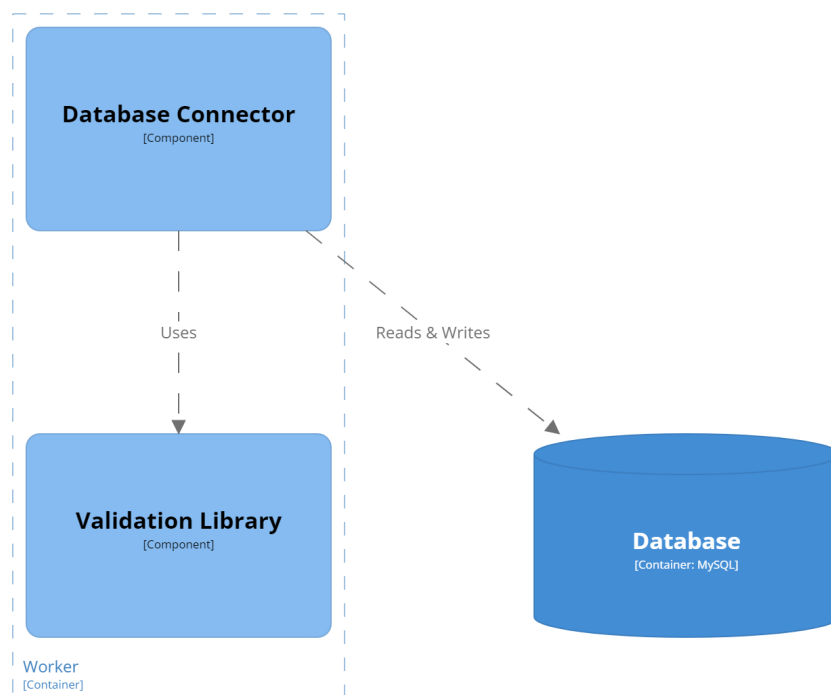
Figure C.3: Components diagram of the web client



[Component] Web Application - Server

pátek 20. října 2023 v 10:42 středoevropský letní čas

Figure C.4: Components diagram of the web server



[Component] Web Application - Worker

pátek 20. října 2023 v 10:42 středoevropský letní čas

Figure C.5: Components diagram of the web worker

D. Server in Openapi

We documented the web interface of the web service on the server with Openapi [43]. The shortened version of the documentation is below. It can be rendered with Swagger (<https://swagger.io/>). Everything is also included in the repository in the attachment A.1 in the file `web-app/server/openapi-docs.yaml`.

```
openapi: 3.0.0
info:
  title: OTAVA Web server
  description: The web interface of the OTAVA server...
  version: 1.0.0
tags:
  - name: Results
    description: Provides validation results and other data...
  - name: Forms
    description: Accepts data from web forms.
paths:
  /validation-data/{validationID}:
    parameters:
      - name: validationID
        in: path
        description: ID of validation to return
        required: true
        schema:
          type: integer
    get:
      tags:
        - Results
      summary: Returns data for specified validation.
      responses:
        '200':
          description: A JSON object with validation data
          content:
            application/json:
              schema:
                type: object
                properties:
                  id:
                    type: integer
                  language:
                    type: string
                  style:
                    type: string
                  passive-tables:
                    type: string
                  active-tables:
                    type: string
                  passive-descriptors:
                    type: string
                  active-descriptors:
```

```

        type: string
    description:
        type: string
    form:
        type: string
    request-time:
        type: string
    finish-time:
        type: string
    state:
        type: string
    outcome-text:
        type: string
    outcome-json:
        type: string
    outcome-turtle:
        type: string
/submit-table-validation:
  post:
    tags:
      - Forms
    summary: Accepts data from table validation form.
    requestBody:
      required: true
      content:
        application/x-www-form-urlencoded:
          schema:
            type: object
            properties:
              tableUrl:
                type: string
              active:
                type: string
              language:
                type: string
            required:
              - tableUrl
              - active
              - language
    responses:
      '302':
        description: Empty response. Redirects to a page...
        headers:
          Location:
            description: /result/{newID}
            schema:
              type: string
/submit-descriptor-validation:
  post:
    tags:
      - Forms

```

```

summary: Accepts data from descriptor validation form.
requestBody:
  required: true
  content:
    application/x-www-form-urlencoded:
      schema:
        type: object
        properties:
          descUrl:
            type: string
          active:
            type: string
          language:
            type: string
        required:
          - descUrl
          - active
          - language
responses:
  '302':
    description: Empty response. Redirects to a page...
    headers:
      Location:
        description: /result/{newID}
        schema:
          type: string
/submit-expert-validation:
  post:
    tags:
      - Forms
    summary: Accepts data from expert validation form.
    requestBody:
      required: true
      content:
        application/x-www-form-urlencoded:
          schema:
            type: object
            properties:
              style:
                type: string
              passiveTables:
                type: string
              activeTables:
                type: string
              passiveDescriptors:
                type: string
              activeDescriptors:
                type: string
              language:
                type: string
            description:

```



```
        type: string
required:
  - style
  - passiveTables
  - activeTables
  - passiveDescriptors
  - activeDescriptors
  - language
  - description
responses:
  '302':
    description: Empty response. Redirects to a page...
    headers:
      Location:
        description: /result/{newID}
        schema:
          type: string
```

E. W3C tests

There are results of the W3C validation tests from the CSVW Implementation Report [17]. The tests are implemented in the file `W3CTests.java` in our project. The following snippet shows the important part of the output. It shows the results of the W3C validation tests.

```
[INFO] Running io.github.janjanda.otava.library.W3CTests
[ERROR] Tests run: 281, Failures: 80, Errors: 0, Skipped: 0
[ERROR] Failures:
[ERROR] Test 012: tree-ops example with directory metadata
[ERROR] Test 013: tree-ops example from user metadata
[ERROR] Test 014: tree-ops example with linked metadata
[ERROR] Test 015: tree-ops example with user and directory metadata
[ERROR] Test 016: tree-ops example with linked and directory metadata
[ERROR] Test 023: dialect: header=false
[ERROR] Test 027: tree-ops minimal output
[ERROR] Test 032: events-listing.csv example
[ERROR] Test 033: events-listing.csv minimal output
[ERROR] Test 109: titles with invalid language
[ERROR] Test 116: file-metadata with query component not found
[ERROR] Test 118: directory-metadata with query component
[ERROR] Test 119: directory-metadata not referencing file
[ERROR] Test 120: link-metadata not referencing file
[ERROR] Test 121: user-metadata not referencing file
[ERROR] Test 139: @type out of range (as node type) - string
[ERROR] Test 142: @value with @language and @type
[ERROR] Test 143: @value with extra properties
[ERROR] Test 144: @language outside of @value
[ERROR] Test 145: @value with invalid @language
[ERROR] Test 146: Invalid faux-keyword
[ERROR] Test 148: title incompatible with title on language
[ERROR] Test 155: number format (valid combinations)
[ERROR] Test 162: numeric format (consecutive groupChar)
[ERROR] Test 164: decimal datatype with exponent
[ERROR] Test 165: decimal type with NaN
[ERROR] Test 166: decimal type with INF
[ERROR] Test 167: decimal type with -INF
[ERROR] Test 169: invalid decimal
[ERROR] Test 185: boolean format (value not matching format)
[ERROR] Test 186: boolean format (not matching datatype)
[ERROR] Test 191: date format (bad format string)
[ERROR] Test 192: date format (value not matching format)
[ERROR] Test 194: duration format (value not matching format)
[ERROR] Test 196: values with wrong length
[ERROR] Test 197: values with wrong maxLength
[ERROR] Test 198: values with wrong minLength
[ERROR] Test 199: length < minLength
[ERROR] Test 200: length > maxLength
[ERROR] Test 210: date value constraint not matching minimum
[ERROR] Test 211: date value constraint not matching maximum
```

[ERROR] Test 212: date value constraint not matching minInclusive
[ERROR] Test 213: date value constraint not matching minExclusive
[ERROR] Test 214: date value constraint not matching maxInclusive
[ERROR] Test 215: date value constraint not matching maxExclusive
[ERROR] Test 216: minInclusive and minExclusive
[ERROR] Test 217: maxInclusive and maxExclusive
[ERROR] Test 218: maxInclusive < minInclusive
[ERROR] Test 219: maxExclusive = minInclusive
[ERROR] Test 220: maxExclusive < minExclusive
[ERROR] Test 221: maxInclusive = minExclusive
[ERROR] Test 222: string datatype with minimum
[ERROR] Test 223: string datatype with maxium
[ERROR] Test 224: string datatype with minInclusive
[ERROR] Test 225: string datatype with maxInclusive
[ERROR] Test 226: string datatype with minExclusive
[ERROR] Test 227: string datatype with maxExclusive
[ERROR] Test 230: failing minLength with separator
[ERROR] Test 244: invalid datatype @id
[ERROR] Test 249: http normalization
[ERROR] Test 258: foreign key multiple referenced rows
[ERROR] Test 259: tree-ops example with csvm.json
[ERROR] Test 260: tree-ops example with {+url}.json
[ERROR] Test 261: maxLength < minLength
[ERROR] Test 269: 'format' for a boolean datatype is a string
[ERROR] Test 279: duration not matching xsd pattern
[ERROR] Test 280: dayTimeDuration not matching xsd pattern
[ERROR] Test 281: yearMonthDuration not matching xsd pattern
[ERROR] Test 282: valid number patterns
[ERROR] Test 284: valid number patterns (grouping)
[ERROR] Test 288: invalid #,#00 1
[ERROR] Test 289: invalid #,#00 1234
[ERROR] Test 292: invalid #,##,#00 1
[ERROR] Test 293: invalid #,##,#00 1234
[ERROR] Test 296: invalid #0.# 12.34
[ERROR] Test 297: invalid #0.# 1,234.5
[ERROR] Test 298: invalid #0.0 1
[ERROR] Test 299: invalid #0.0 12.34
[ERROR] Test 300: invalid #0.0# 1
[ERROR] Test 301: invalid #0.0# 12.345