Master Thesis

Semantic Form Editor

Bc. Tomáš Klíma
Supervisor: Mgr. Miroslav Blaško, Ph.D.

January 2021

Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Klíma Tomáš** Personal ID number: **459933**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Software Engineering**

II. Master's thesis details

Master's thesis title in English:

Semantic form editor

Master's thesis title in Czech:

Editor sémantických webových formulářů

Guidelines:

SForms is a javascript library for displaying interactive forms in the ReactJS framework. Unlike similar libraries, SForms is based on Semantic Web technologies, which allows form questions and answers to be mapped to knowledge stored in ontologies. The aim of this work is to create an SForms form editor that will help the user create forms based on current best practices and mapping to knowledge stored in ontologies. Associations with ontological knowledge can also be used to validate created forms.
Instructions:
1) become familiar with Semantic Web technologies for representation (OWL, RDF, JSON-LD) and validation (SHACL) of knowledge
2) explore existing form creation tools and current best practices for form design
3) analyze the requirements of the editor and SForms extension
4) design and implement a prototype editor, including the necessary extension of SForms
5) test the implemented prototype, including user testing on at least 3 users

Bibliography / sources:

- Wroblewski, Luke. Web form design: filling in the blanks. Rosenfeld Media (2008).
- Wood, David. 'What's New in RDF 1.1.' W3C Working Group Note (2014).
- Walke, Jordan. 'React-A JavaScript library for building user interfaces.' (2013).
- Ledvinka M., Blaško M., SForms (https://kbss.felk.cvut.cz/web/kbss/s-forms)

Name and workplace of master's thesis supervisor:

Mgr. Miroslav Blaško, Ph.D., Knowledge-based Software Systems, FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **07.09.2020** Deadline for master's thesis submission: **05.01.2021**

Assignment valid until: **19.02.2022**

Mgr. Miroslav Blaško, Ph.D. Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Abstract

Forms can be an excellent tool for collecting information effectively. However, there is no straightforward path to creating an ideally designed research instrument, able to gather data of sufficient quality. Building a form that is understandable for respondents and efficient at the same time, requires knowing its target audience, as well as following the form design guidelines. Nevertheless, the quality of data is not the only trait sufficing for a form to be efficient. Other characteristics of an efficient form can be the reusability or the integrability with existing information from different domains. These can be achieved by using Semantic Web technologies. A form editor is a useful tool that provides a user interface for creating forms. There are plenty of form editors available, but minimum of them is capable of operating on forms build upon the Semantic Web. The main goal of the thesis is to create an editor that allows form designers to build, adjust, and validate the Semantic Web based forms that are afterwards visualised with the SForms JavaScript library. The application is developed in React web framework Next.js with the use of TypeScript and Material-UI. Created forms are validated to comply with form best-practice guidelines using the constraint language SHACL. This master thesis further intends to analyse forms in terms of their digital representation and examine the best-practises of their design and following visualisation. The work moreover analyses eight existing solutions of form editors and validates them against the design guidelines. Taken together, this work aims to examine form design guidelines, analyse technologies concerning the Semantic Web and finally create a form editor capable of creating forms based on Semantic Web technologies.

Keywords: Form, Form best-practices, Form Editor, Web Application, JavaScript, React, Next.js, SForms, JSON-LD, SHACL, Semantic Web, Linked Data

Abstrakt

Formulář představuje důležitý prostředek pro sběr dat. Vytvořit formulář, který je efektivní a zároveň srozumitelný pro jeho respondenty, není jednoduché, jelikož to vyžaduje znalost cílové skupiny dotazovaných a osvědčených postupů pro návrh formulářů. Kvalita dat však není jediným znakem formuláře postačujícím pro efektivní sběr dat. Dalšími charakteristikami efektivních formulářů může být jejich opětovná použitelnost nebo integrovatelnost s existujícími informacemi pocházejícími z jiných domén. Toho lze dosáhnout pomocí technologií Sémantického webu. Editor formulářů je užitečný nástroj, který poskytuje uživatelské rozhraní pro vytváření formulářů. Formulářové editory jsou v současnosti běžně dostupné, ale minimum z nich je schopno pracovat s formuláři založenými na technologiích Sémantického webu. Hlavním cílem této práce je vytvořit editor, který umožní návrhářům formulářů vytvářet, upravovat a validovat formuláře založené na Sémantickém webu, které jsou následně vizualizovány pomocí JavaScriptové knihovny SForms. Aplikace je vyvinuta za použití webového frameworku Next.js s technologií TypeScript a knihovnou Material-UI. Vytvořené formuláře jsou ověřovány, aby byly v souladu s pravidly osvědčených postupů návrhu formulářů pomocí jazyka SHACL. Tato diplomová práce dále obsahuje analýzu formulářů z hlediska jejich digitální reprezentace a zkoumá osvědčené postupy jejich návrhu a následné vizualizace. Práce navíc analyzuje osm existujících řešení editorů formulářů a validuje je na základě ověřených postupů návrhu formulářů. Tato práce si v souhrnu klade za cíl prozkoumat osvědčené postupy pro návrh formulářů, analyzovat technologie týkající se Sémantického webu a nakonec vytvořit editor formulářů schopný vytvářet formuláře založené na technologiích Sémantického webu.

Klíčová slova: Formulář, Osvědčené postupy návrhu formulářů, Editor formulářů, Webová aplikace, JavaScript, React, Next.js, SForms, JSON-LD, SHACL, Sémantický web, Propojená data

Author statement for graduate thesis:

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date ....

signature

Acknowledgements

I would like to thank my supervisor Mgr. Miroslav Blaško, Ph.D. for his guidance and generous support during my final project and thoroughly testing my application during the implementation.

Contents

Introduction

In all kinds of industries and organisations, the documents with predefined blank spaces serving for filling data are used to capture various pieces of information in a specific structure. Essentially, it is possible to carry out the process of collecting data in two ways. Firstly, in the traditional way, through the use of paper forms and secondly, by using one of the types of digital forms. An example of the most used type of digital forms is nowadays a web form. It can be found on almost any website, ranging from social media applications to e-shops to tools for the management of medical reports. However, most of the forms are missing dynamic features, whether it is due to being a paper solution or an insufficient adjustment to computer capabilities. Additionally, the majority of forms, even the digital ones, do not use the Semantic Web technologies which makes them not readable for machines.

The Semantic Web[6] is a technology that extends the World Wide Web. It provides an option for publishing information in the format of Linked Data[6]. It also allows easier integration with existing web sources, such as medical, financial, or statutory vocabularies.

The Semantic Web is not yet widespread, neither are semantic forms, which are forms built upon principles and technologies of the Semantic Web. One of the possible reasons that semantic forms are not well-used can be the lack of tools supporting creation of such forms. Currently, there are multiple existing solutions of editors for building digital forms interactively, via a user interface. However, semantic forms cannot be created without significant understanding of the underlying Semantic Web technologies, which is a knowledge not every form designer has.

The main goal of the thesis is to create a form editor for creation of high-quality semantic forms, based on SForms. SForms is a library capable of visualising semantic forms, defined using the Semantic Web standards. The Semantic Form Editor, implemented as a result of this thesis, allows creating, editing, as well as validating of semantic forms using a user interface. Apart from creating and editing, validating is a key process when it comes to user experience, as well as the efficiency of data gathering. The Semantic Form Editor validates the created forms against form best-practice guidelines, which define proper behaviour of forms, in order to collect data of sufficient quality.

To conclude, the Semantic Form Editor should avoid the problems of existing solutions, support the features a form designer can rely on to be able to design forms. Overall, the final application should be able to allow form designers to build and modify complex semantic forms according to a form owner's requirements and validate them from the perspective of best-practice guidelines.

Chapter 1 of the thesis gives an introduction to the topic the work is based on. It discusses the problematics of forms and explains the technologies of the Semantic Web. The following Chapter 2 summarises the best-practice guidelines of how to design web forms. Chapter 3 presents the analysis of existing solutions used for building digital forms. Next, Chapter 4 describes the JavaScript SForms library from the perspective of usage, technology stack, and the semantic form representation. Afterwards, Chapter 5 demonstrates the design process of the form editor, which is realised based on the pre-

vious investigation. Moreover, modifications of the SForms library are proposed to be able to fulfil the requirements of the Semantic Form Editor. Chapter 6 describes the implementation of designed features in the application as well as in the SForms library. Furthermore, Chapter 7 discusses the methods of testing, the testing itself, and the evaluation of usage of the Semantic Form Editor. Finally, the conclusion summarises the overall result and the future of the work.

Chapter 1

Background

This chapter gives an introduction to the topics, the thesis is based on. The first part introduces form design topic and establishes related terminology used throughout the thesis. The second part describes the Semantic Web and the technologies behind it.

1.1 Form

A form is a conversation between two parties, the form owner and the user. It can be defined as a document with blank spaces serving for insertion of a requested piece of information.[30]

A form is a well-used technique for collecting data, which have been used, to some extent, for centuries. Forms are documented to have existed since the 19th century when preprinted documents were used by lawyers.[14]

Administrative and repetitive processes where information is collected or stored can become a burden if the information is requested or gathered in an unorganised way. Using reusable structured documents for requesting specific information instead, is an effective way to collect information.

Forms have been used in multiple spheres of our daily lives, ranging from office administration to medical reports to ordering in an e-shop. Forms can be categorised into paper and digital. This classification results from technology development, as in the past, the paper variant had been the only available one. To this day, paper forms are still used for multiple purposes, for instance, in offices, in schools or for elections. However, as various domains proceed to digitalisation, paper forms are gradually replaced by digital forms. For example, electronic medical reports, eCRF, respectively, are being preferred over paper medical reports.[21]

The data collection via a form is realised by asking questions by the form owner and stating answers by the respondent.

Questions can be generally classified into three types: open, closed and semi-closed questions. *Closed questions* can be answered by a limited range of answer options. For example, questions answered with either *yes*, or *no*, are considered as closed. *Open questions* do not provide any set of answer options from which can be chosen. Instead, respondents are required to use their own words to describe the answer based on their knowledge and experience.[37] *Semi-closed questions* is a combination of open and closed question types. Respondents can choose out of a set of answer options or provide a custom text answer in provided text field.[40]

Keeping in mind the difference between question types can play a significant role in effective collection of data.

1.1.1 Paper Form

A paper form also referred to as a printed form, is a traditional method for data collection. It consists of static elements, such as blank fields and context information.

The main benefits of paper forms are listed below:

1. Although many areas are continuously shifting towards digitalisation, not all processes support digital forms. Either they are not fully standardised or legalised. Therefore, a paper form is the only option to use.

2. No computer is needed for filling a paper form, as not all respondents have access to one.

3. Paper forms are easier to use. Some respondents neither have sufficient technical knowledge nor the willingness to use something they do not know.[21]

In spite of these benefits, paper forms lack many functionalities that digital forms have, therefore are generally less effective than digital forms.[41]

1.1.2 Digital Form

There are various instances of digital forms, for example, web forms, or PDF and ZFO[1] form formats. A digital form, in essence, simulates the paper form by using the same form elements. The reason is the ease of use since the paper form design is what users are used to working with. In addition to traditional form elements used in paper forms, for example, multiple-choice, single-choice and text fields, a digital form contains a few elements more, as can be seen in Section 1.1.3.1.[9]

Contrary to paper forms, digital form elements are interactive and has numerous advantages over its paper variant, for instance:

1. Based on respondent's previous answers, only relevant questions can be asked. This is more discussed in Section 1.1.3.2.

2. A digital form can automatically check for errors of omission and commission. A user might not have filled all the information required, that is the error of omission. The error of commission, on the other hand, is caused by violating specific field rules. This process is called validation.[9]

3. The collected data are represented in a digital form, so they can be immediately sent for analysis or stored in a database.

4. The whole process of a form lifecycle, from a form distribution to data analysis to data storage can be automated.

5. Some input fields might be pre-populated, for example, some answers serving solely for identification may be pre-filled, which reduces repetitive information filling. One example is eCRF, where the digitalisation can help with the automatic generation of the following fields: a study subject, a study number, a subject description, and date.[21]

A form lifecycle is a process involving many different parties. For the purpose of this thesis, I define four involved parties in Table 1.1.

[1] ZFO is a form file created in 602XML Form Server application. More information available at `https://filext.com/file-extension/ZFO` to 5/11/2020.

Table 1.1: Parties involved into forms

Term	Definition
Form owner	A person or an organisation using a form to collect data.
Form designer	A person designing a form.
Respondent	A person filling data in a form.
Form builder/editor	An application, which helps a form designer create forms using a user interface.

1.1.3 Web Form

A web form is defined as a form that respondents fill inside a browser.[9] It is typically built using Hypertext Markup Language (HTML[2]), Cascading Style Sheets (CSS[3]) and optionally JavaScript[4]. A HTML form consists of special elements called form controls. The lifecycle of the web form consists of the modification of its form controls and the following submission to an agent for processing, for example, a web server and a mail server.[19]

Source code 1.1: Basic example of an HTML form

```
<form action="http://jobs.com/apply-for-a-job" method="post">
    <label for="name">Name:</label>
    <input type="text" id="name">
    <label for="birth">Date of birth:</label>
    <input type="datetime-local" id="birth">
    <input type="radio" id="male" name="gender" value="male">
    <label for="male">Male</label><br>
    <input type="radio" id="female" name="gender" value="female">
    <label for="female">Female</label>
    <input type="radio" id="other" name="gender" value="other">
    <label for="other">Other</label>
    <button type="submit">Apply</button>
</form>
```

Source code 1.1 is an example of an HTML form which consists of a form element wrapping form controls, namely a text field, a datetime picker, radio buttons and a submit button. After the user answers all questions from the example form, he clicks on the submit button and the form is sent to a server for processing.

In a web form, a common question consists of a label, an HTML form element, an optional question-level help, an optional required/optional indicator and a validation message in case of error.

The example of a form created using the Bootstrap 4 framework[5] with all building parts of question mentioned in the preceding paragraph can be seen in Figure 1.1.

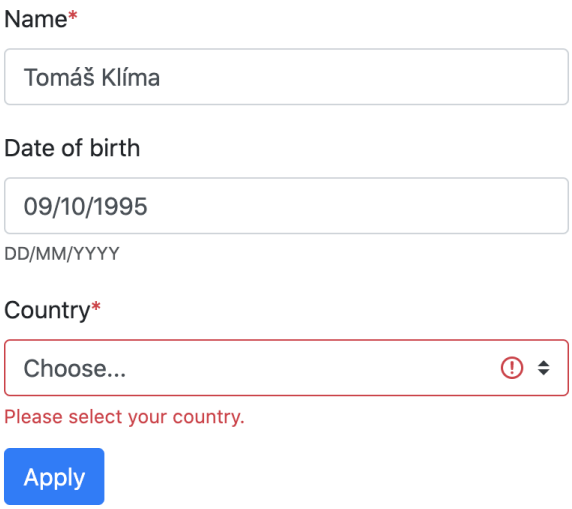
[2]More information about HTML available at https://html.spec.whatwg.org/ to 4/11/2020

[3]More information about CSS available at https://www.w3.org/Style/CSS/ to 4/11/2020

[4]More information about JavaScript available at https://www.w3.org/standards/webdesign/script to 4/11/2020

[5]More information about Bootstrap framework is available at https://getbootstrap.com/ to 4/11/2020

Figure 1.1: A web form created using the Bootstrap 4



1.1.3.1 Form Controls

For the purpose of this thesis, forms controls are divided into three categories: basic, custom and other.

The basic form element is a form control for collecting data which is built-in HTML5 standard[6]. The overview of the most used basic form elements, with the naming, corresponding HTML5 element and brief description, can be seen in Table 1.2.

Table 1.2: Basic form controls overview

Term	HTML5 element	Description
Text field	Input of type text	A single-line text field.
Text area	Textarea	A multiple-line text field.
Dropdown	Select	Choose one answer from a toggleable menu of options.
Single-choice	Multiple radio buttons	Choose only one answer from list of options.
Multiple-choice	Multiple checkboxes	Choose multiple answers from list of options.
Number field	Input of type number	A field for entering a number.
Email field	Input of type email	A field for entering an email.
Range	Input of type range	A slider with a draggable handle.
Datetime, Date, Time, Week, Month	Input of type datetime-local, date, time, week, month	Fields for entering time.
File uploader	Input of type file	A field for uploading a file.
Color	Input of type color	A field for choosing a color.
Telephone field	Input of type tel	A field for entering a phone number.
Autocomplete field	Input with attribute list, datalist	A dropdown with an option to type. Based on the typing, the options are filtered and displayed.

[6]More information about HTML5 standard available at `https://www.w3.org/TR/2014/REC-html5-20141028/` to 4/11/2020

The custom form element is a custom form control for collecting data which is not part of the HTML5 standard. The overview of the most used custom form controls, with the naming and brief description, can be seen in Table 1.3.

Table 1.3: Custom form controls overview

Term	Description
Fill in the blank	A text with text fields.
Image choice	Choose image from list of images.
Masked text	A field ensuring and displaying a predefined format, for example, DD/MM/YYYY.
Matrix of dropdowns	A 2D matrix serving for asking multiple questions in a compressed space. It contains dropdowns for collecting answers.
Matrix of single-choice	A 2D matrix serving for asking multiple questions in a compressed space. It contains radio buttons for collecting answers.
Matrix of multiple-choice	A 2D matrix serving for asking multiple questions in a compressed space. It contains check boxes for collecting answers.
Scale rating	A single-choice question allowing to rank on a scale represented by numbers.
Star rating	A single-choice question allowing to rank on a scale represented by stars.
Signature	Handwrite a signature in a box.
Spinner	A number field with plus and minus buttons.
Ranking	A ranking of options in the specific order.

The other form control is an element which does not serve for collecting data. The overview of the most used custom form controls, with the naming, corresponding HTML5 element and a brief description, can be seen in Table 1.4.

Table 1.4: Other form controls overview

Term	HTML5 element	Description
Button	Button	A button can serve, e.g. for submitting or resetting the form.
Heading	Heading	A title heading.
Paragraph	Paragraph	A paragraph of text.
Media content	Image, video, audio	A box with media content.
Section		A box for questions with a title.

1.1.3.2 Conditional Logic and Branching

Using the web forms, it is possible to benefit from the option to hide irrelevant questions. As an example, a respondent in a doctor's registration form states that he is a male. Thus, the questions about pregnancy are irrelevant and will not be shown. The pregnancy questions from this example are called *conditional questions*, and the gender question is called the *trigger question*.[9]

1.1.3.3 Form Types

There are different ways for how to format a form, in different words, how to organise the questions within the form. The most common are described below:

- **Classic form** is a form type, where all questions are displayed on one screen at a time. The respondent can scroll to navigate between questions.[18]
- **Multi-step form**, is a form type, where the form is divided into multiple steps having one step per one screen.[9] A user can use next and previous buttons to navigate between steps. The current state can be visible in progress bar.
- **Wizard form**, is similar to multi-step form. However, instead of progress bar, there is a *wizard* which allows a user to navigate directly between specific steps without completing the previous steps.
- **Card form** represents a form type, where only one question is shown at a time to the respondent. After the question is finished, the user can move to a next question.[18]

1.1.3.4 Form Editor

A form editor also referred to as a form builder is an application, which helps form designers build digital forms with minimum or no coding skills.[41] There are many different form editors available. They are either used as an application for building a structure of the form which is then integrated into a form owner's application or can also work as a powerful standalone application for whole lifecycle of forms. In case of a standalone application, the application can cover the whole data collection cycle, including the distribution of the form to respondents and an analysis of obtained answers.

As mentioned before, the form editor offers a solution not requiring any knowledge in programming. It usually contains an intuitive interface for adding individual form controls to a workspace to build a form. A Drag and Drop API[7] is often used for placing and moving form controls in a workspace. The added form controls can be further customised, for example in terms of a label, question-level help or validation rules. The resulting form can be exported in various formats, or, in case of the mentioned standalone form editors, hosted and distributed to respondents.

Note that form editors do not usually allow for customisation of how the form or individual form controls will look like. However, quality form editors often follow UX design and form best-practices guidelines, so traits, such as usability or accessibility, are typically kept.

1.2 Semantic Web

The current web is a series of connected documents, which are designed to be understandable by humans, but meaningless by computer programs. The Semantic Web is an extension of the web, which aims to interconnect all the Web information into one big database of data called Linked Data. In other words, it means to structure the information in a way to be understood not only for humans but also for computers. The Semantic Web technologies enable people to create data stores on the Web, write rules for handling data as well as build ontologies, also called vocabularies.[6, 25] The Semantic Web technologies, such as RDF, OWL, or SPARQL, will be further described later on in this chapter.

1.2.1 Linked Data

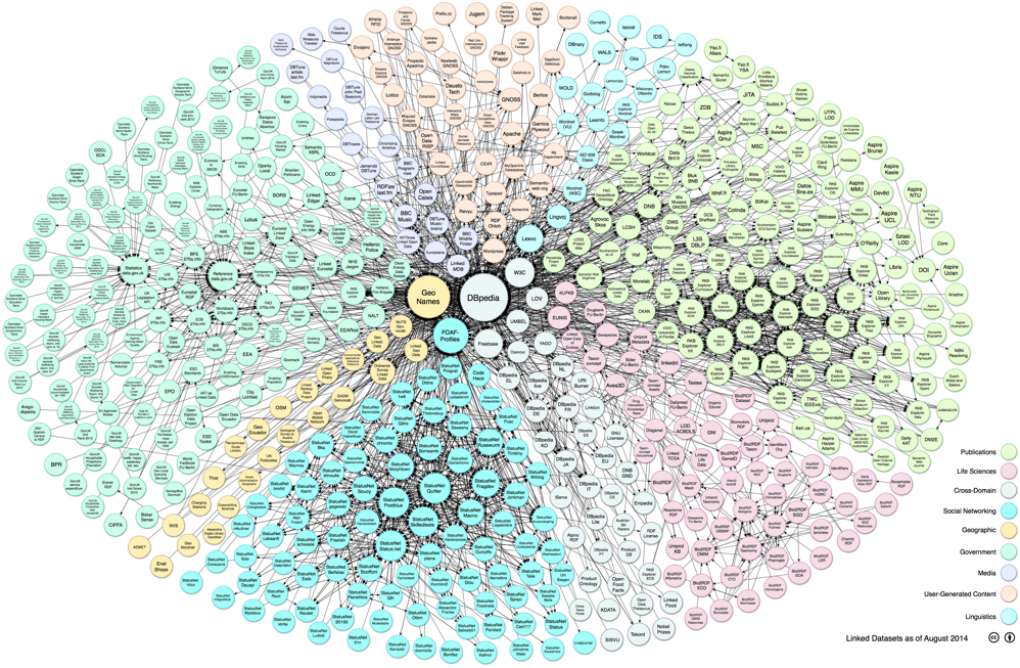
Linked data is a collection of interconnected data, which are specified in a standard format, such as RDF. In addition, those data are reachable and manageable by Semantic Web

[7] Allows to drag HTML element to different place. More information available at: `https://developer.mozilla.org/en-US/docs/Web/API/HTML_Drag_and_Drop_API`

tools, e.g. by SPARQL.[6]

Figure 1.2 shows, how the data, information and knowledge were interconnected in year 2014. Nowdays, the example would be even more complex.

Figure 1.2: Linked Open Data Cloud in 2014



Source: http://lod-cloud.net/versions/2014-08-30/lod-cloud.svg

1.2.2 Ontologies

Ontologies describe the semantics of data in a uniform way which enables communicating specific domain knowledge between different parties. For example, between people and between application systems.[42]

The ontology is represented by four main components:[42]

- A **concept**, also known as class, is an abstract group, a set or a collection of objects, which share common properties. This component is represented as a hierarchical graph where a concept can be represented by superclasses and subclasses. For example, the work of art can be represented as a superclass with many subclasses, such as music, a painting, and a sculpture.
- An **instance**, which defines the specific object of the concept. In case of the concept painting, it can be the Mona Lisa.
- A **relation**, which is used to express the relationship between two concepts within a specific domain. For example, Leonardo Da Vinci has a *creator* relation with the Mona Lisa painting.
- An **axiom**, is used to impose constraints on the values of concepts or instances to verify the consistency of ontology.

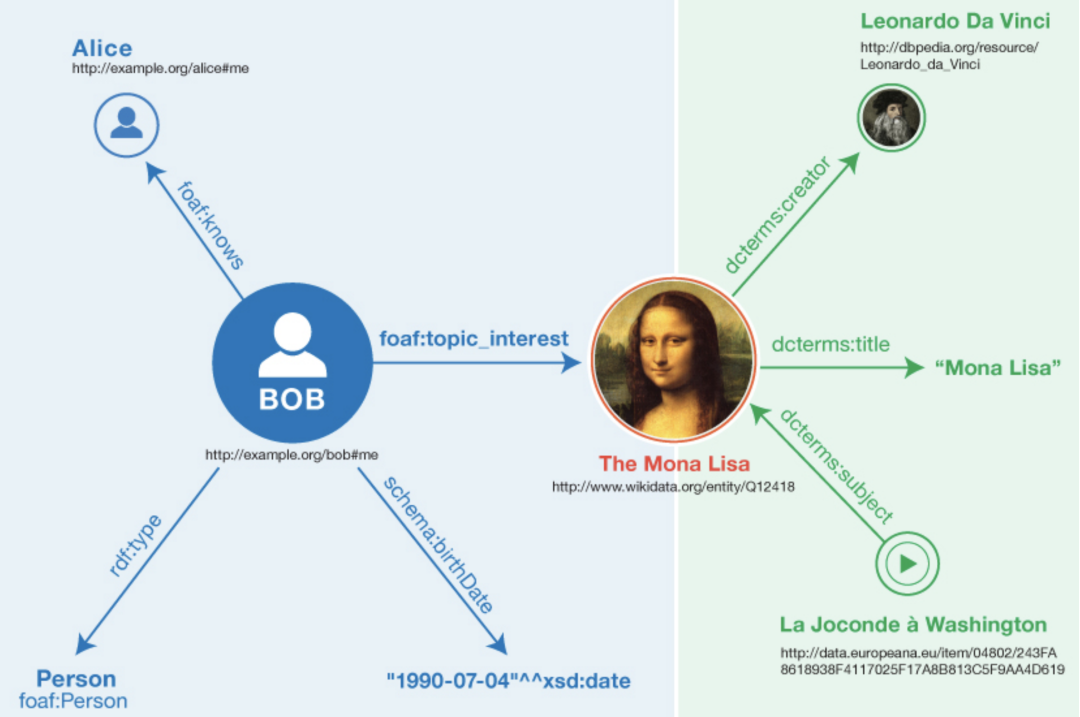
1.2.3 RDF

The Resource Description Framework, abbreviated to RDF, is a language serving for the representation of information about resources on the web. The resources are represented

by RDF Triples: Subject-Predicate-Object.[47] For instance, the artists (subject) paints (predicate) a painting (object). The subject, predicate and object can be identified by web identifiers called URIs[8]. In addition, subjects can also be identified by blank nodes and the objects can be identified by blank nodes as well as literals. Literals are used to identify values by a lexical representation, for example, numbers and dates, which can be plain or typed.[3] The blank nodes indicate the existence of a resource without using it or saying anything about it.[13]

Figure 1.3, represents visualised triples as a connected graph also referred as an RDF graph where Bob is a person, who is friends with Alice and has birthday on a specific date identified by a typed literal. He is interested in the painting of Mona Lisa.

Figure 1.3: Example of an RDF graph



Source: https://www.w3.org/TR/rdf11-primer/example-multiple-graphs-iris.jpg

The RDF graph can be serialised into the following formats:[47]

- Turtle family of RDF languages: **N-Triples**, **Turtle**, **TriG** and **N-Quads**. For example, the content of Figure 1.3 is expressed in N-Triples in Source code 1.2.
- **JSON-LD**, which will be more specified in Section 1.2.6.
- **RDFa**, which is a format of embedding triples into HTML and XML[9].
- **RDF/XML**, which is a format of RDF graph coded in XML.

[8]Uniform Resource Identifier. More information available at `https://www.w3.org/TR/uri-clarification/` to 5/11/2020

[9]Extensible Markup Language. More information available at `https://www.w3.org/XML/` to 5/11/2020

Source code 1.2: Representation of Figure 1.3 in N-Triples Turtle format

```
@prefix ex: <http://example.org/> .
@prefix wd: <http://www.wikidata.org/entity/> .
@prefix sh: <http://schema.org/> .
@prefix p: <http://purl.org/dc/terms/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dbpedia: <http://dbpedia.org/resource/> .
@prefix xml: <http://www.w3.org/2001/XMLSchema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:bob#me rdf:type foaf:Person> .
ex:bob#me foaf:knows ex:alice#me .
ex:bob#me sh:birthDate "1990-07-04"^^xml:date .
ex:bob#me foaf:topic_interest wd:Q12418 .
wd:Q12418 p:title "Mona Lisa" .
wd:Q12418 p:creator dbpedia:Leonardo_da_Vinci .
```

Source: https://www.w3.org/TR/rdf11-primer/ [Edited]

The RDF and RDFS vocabulary are collections of IRIs[10] designed for use in RDF graphs. They provide constructs to define the semantic characteristics of RDF data.[47] For example, the information that Bob from Figure 1.3 is a person is defined using the construct *rdf:type* from RDF vocabulary.

1.2.4 OWL

The Web Ontology Language, a so called OWL, is a part of the vocabulary family together with the RDF and RDFS vocabulary. OWL adds another semantics on how to describe RDF data. It contains some of the RDFS constructs as well as XSD datatypes and a lot more.[29]

1.2.5 SPARQL

SPARQL is an RDF query language for NoSQL graph databases, also known as RDF triplestores. SPARQL allows querying RDF data by applying similar constructs as MySQL does in relational databases.

The query consists of two parts. First, the query clause:[12]

- The **SELECT** form returns result in a table format.
- The **CONSTRUCT** form returns a single RDF graph specified by a graph template.
- The **ASK** form tests if the query pattern has a solution and returns a boolean.

The second part is the WHERE clause, which provides the pattern to match against the data graph. The example of a SELECT query on a sample data can be seen in Source code 1.3. The WHERE clause there also contains the FILTER function which restricts solutions only to those satisfying the expression.[12]

[10] International Resource Identifier. More information available at: https://www.w3.org/International/O-URL-and-ident.html to 5/11/2020

Source code 1.3: SPARQL query on RDF graph

```
PREFIX  ex: <http://example.com/ex/elements/1.0/>

:book1  ex:title  "Vlcice, kapitoly z dejin podkrkonosske obce" .
:book1  <http://example.com/ex/elements/year> 2019 .
:book2  dc:title  "Trutnov, toulky minulosti a soucastnosti" .
:book2  <http://example.com/ex/elements/year> 2002 .


SELECT ?title ?year
WHERE {
  ?x ex:title ?title .
  ?x <http://example.com/ex/elements/year> ?year.
  FILTER (?year < 2010) .
}
```

1.2.6 JSON-LD

JavaScript Object Notation for Linked Data, abbreviated to JSON-LD, is a serialisation format for RDF data. JSON-LD is primarily intended to be used in web applications as the JSON-LD is fully compatible with JSON. In addition, the existing JSON can be easily interpreted as JSON-LD with minimal changes.[26]

Source code 1.4: Representation of Figure 1.3 in JSON-LD format

```
{
   "@context":{
      "foaf":"http://xmlns.com/foaf/0.1/",
      "interest":{"@id":"foaf:topic_interest", "@type":"@id"},
      "knows":{"@id":"foaf:knows", "@type":"@id"},
      "birthdate":{"@id":"http://schema.org/birthDate",
      "@type":"http://www.w3.org/2001/XMLSchema#date"},
      "dcterms":"http://purl.org/dc/terms/",
      "title":"dcterms:title",
      "creator":{"@id":"dcterms:creator","@type":"@id"},
      "subject_of":{"@reverse":"dcterms:subject","@type":"@id"}
   },
   "@graph":[
      {
         "@id":"http://example.org/bob#me",
         "@type":"foaf:Person",
         "birthdate":"1990-07-04",
         "knows":"http://example.org/alice#me",
         "interest":"http://www.wikidata.org/entity/Q12418"
      },
      {
         "@id":"http://www.wikidata.org/entity/Q12418",
         "title":"Mona Lisa",
         "subject_of":"http://data.europeana.eu/item/04802/243FA
         8618938F4117025F17A8B813C5F9AA4D619",
         "creator":"http://dbpedia.org/resource/Leonardo_daVinci"
      }
   ]
}
```

Source: https://www.w3.org/TR/rdf11-primer/ [Edited]

Source code 1.4, shows the RDF graph from Figure 1.3 in JSON-LD format. This example shows a JSON-LD document in a top-level structure. It uses in-line context definition. The *@context* part describes how the document can be mapped to an RDF graph.

In the *@graph* part, each JSON object corresponds to an RDF resource and is described using the *@id* keyword. The first RDF resource is described as *http://example.org/bob#me*. This resource by construct *interest* is interested in the second resource, which is the Mona Lisa painting identified by URI *http://www.wikidata.org/entity/Q12418*.[47]

1.2.7 SHACL

The Shapes Constraint Language, a so called SHACL, is a language for describing RDF graphs and validating them against a set of conditions. The conditions can be represented as shapes and constraints. The constraints can be, for example, for validation of the pattern, the length, the numeric range or the datatype of graph properties. The constraints can be extended using SPARQL queries SELECT and ASK, or by using JavaScript programming language. Together with the constraints, it is possible to specify the severity and a message in case of a violation of a rule. Based on the validation process, the validation report is generated.[24]

The SHACL language can also be extended by Data Shapes Vocabulary (DASH). DASH is a collection of reusable extensions. It provides new constraints and target types, as well as components for representing test cases and suggestions for how to fix violated constraints.[22]

Source code 1.5, shows the shapes graph in Turtle format for validation of an RDF graph, in this example an ontology-based smart form. A node of type *doc:question*, either should not be a form control of type autocomplete or should have at least four answer options. If the constraint is violated, it is included in a validation report with specified message.

Source code 1.5: An example of SHACL validation constraint in Turtle format

```
:s09
  a sh:NodeShape ;
  sh:property [
    sh:or (
        [
          sh:path form:has-possible-value ;
          sh:minCount 4 ;
        ]
        [
          sh:not
            [
               sh:path form-lt:has-layout-class ;
               sh:hasValue "type-ahead" ;
            ]
        ]
      )
    sh:message "Autocomplete form control should have at least four answer
        options"@en;
  ];
  sh:targetClass doc:question ;
.
```

Chapter 2

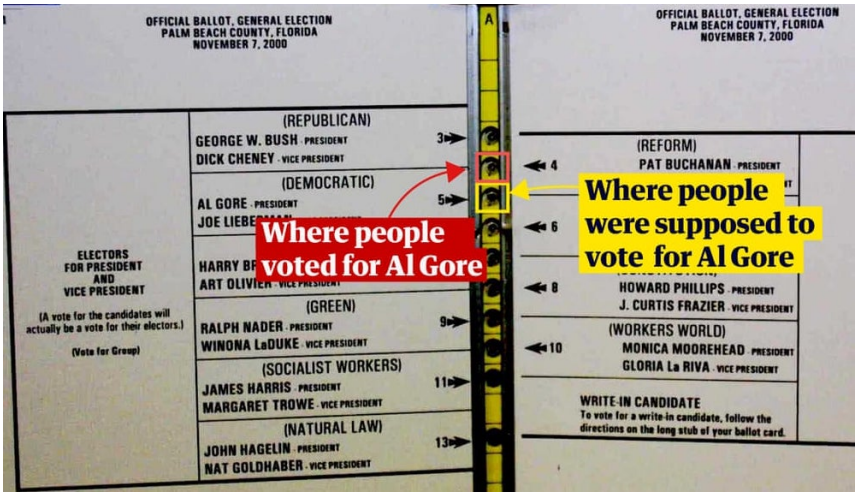
Form Design Guidelines

This chapter will analyse best-practice guidelines on how to design web forms. There are four fundamental goals to accomplish in web form design. The first goal is to make forms as simple as possible to minimise the struggling of responders while filling forms. The second principle is to illuminate a path to completion. The goal of the form owner is to get a form filled, so the form should state clearly how it can be filled. The next goal is to consider the context. There is always a reason why the form exists, and the reason should be considered. The last but not least is to ensure the consistency in communication. The form should overall speak to a user with one voice.[48]

When forms are designed poorly, the collected data can be of lower quality. The reasons are, for example, that a user is burdened with too many unimportant questions so he might overlook the important ones. Next, a user might misunderstand questions and therefore enter incorrect answers.

An excellent example of a bad form design with a considerable impact is the presidential elections in Florida in the year 2000. That time, Vice President Al Gore lost elections in Florida against George W. Bush by a 537-vote margin. Voters who wanted to vote for Al Gore, accidentally voted for Pat Buchaman.[32] The problem with the form is captured in Figure 2.1.

Figure 2.1: Butterfly ballot used for elections in Florida in the year 2000.



Source:
https://theguardian.com/us-news/2019/nov/19/bad-ballot-design-2020-democracy-america

Forms satisfying best-practices have a great potential to increase the number of successful first-trial submissions. In addition, they minimise the risk of leaving the form as a

consequence of unsuccessful submission. Moreover, completion times of forms following best-practices are between 18% to 33% faster.[38]

The outline of this chapter is inspired by the structure of the book *Designing UX: Forms* by Jessica Enders.[9] Parts marked with a star (*) can be used as best-practices validation rules in a form editor. Dagger (†) mark represents suggested features of a form rendering application.

Other information cannot be implemented without artificial intelligence. However, they should serve a potential form designer to design forms better.

2.1 Wording

Words are one of the foundation stones of the form design. They are used across the whole form. In questions, more specifically in question labels, answers, and question-level helps, and moreover, in all different types of headings, buttons and messages.

2.1.1 Wording of Questions

The question has to convey the information to a user accurately to avoid ambiguity, as well as to achieve sufficient quality of collected data. Without a correctly formulated question, users may not get the context right and answer incorrectly. Besides, validation often cannot check for those types of errors, so they should be prevented.

Answering a question is the process of four phases. First of all, respondents have to interpret the question to understand its meaning. That is called comprehension. The second phase is retrieval. Respondents need to retrieve specific information from their memory. In the next stage, respondents use that particular information to render a judgement. The final step, respondents use their decision to provide an answer. They have to select an answer from a prepared set of options or type it to be consistent with preceding answers or social desirability.[44]

2.1.1.1 Comprehension

During a question design, it is required to keep in mind that respondents will probably skim through the question instead of thoroughly reading it. A question label should contain all information needed by all users. Using question-level help for that purpose is insufficient. It is recommended to avoid jargon and abbreviation for the formulation. Only plain language should be used in order to make the question understandable for the user. To satisfy the language criteria, a form designer should use:[9]

- Short and simple words
- Short and simple sentences which express only one idea
- Active voice
- Words familiar to the group of respondents

Wording should be consistent. Full-sentence questions should not be mixed with brief prompts if it hinders comprehension*. The same applies for punctuation at the end of the label. It is possible to use question marks, colons or nothing, but it should be consistent across the form*.[9]

Personalisation words like *My* and *Your* should be avoided because they are redundant and add to the user's mental overload*. The only exception is when the form refers to more than two parties.[9]

2.1.1.2 Retrieval

A form should contain only the questions which a user can answer. If a user is unable to recall an answer from his memory, he should be informed in advance which information and equipment he needs to be able to respond to all questions in the form.[48]

2.1.1.3 Judgement

The form designer should be aware of what is appropriate to ask and the way it is asked within a particular context of the form. This can help to avoid getting the user into a situation where he struggles how and whether to answer. If an unpleasant question has to be asked, a question help should be added to explain why the information is needed.[9] If the answer is required in a specific format, then there should be a statement presenting the rule.[2]

Additionally, answer options should be precise, especially when expressing time frames, weights, money and other quantities. The reason is that every respondent can understand, for example, *rarely* as a different time unit.[9]

2.1.1.4 Answer

As already mentioned in Chapter 1, there are three types of questions, open, closed, and semi-closed. In an open question, the user can type, whatever the validation rules allow him. If the validation rules are specified, the question-level help should explain the format of the answer to avoid possible errors. In the closed questions, the user has to choose one from the predefined answers. A correctly specified closed question helps the user interpret the question correctly and avoid potential errors.[9]

A form designer should choose closed questions only if the right set of answer options is known. The answer options should be short, mutually exclusive, self-explanatory and unbiased. Furthermore, it should be complete, that means there must be an option for every user. If it is not possible to include all possible answer options, then it is suitable to use semi-closed question type. That means that one of this type of answer variants should be provided: *Other*, *Don't know*, *Not applicable* or a possibility to type individual answer*.[9] Finally, the answer options should be sorted or grouped in an intuitive way. If no meaningful order exists, options can be sorted alphabetically.[2]

2.1.2 Wordings of Other Parts of the Form

The wording of questions rules also applies for the following parts of forms, especially the ones involving language.

2.1.2.1 Headings

Every form should have a title heading*†. It should be brief, unique and descriptive with a maximum of five words*. In addition, it should avoid articles and redundant words like *form*, *online*, *web*, and *for**. Next type is a section heading†. It is applicable for longer form and helps a user notice the change of topic. The disadvantage is the prolongation of form. A good approach is to start without the section heading and add it only if it is beneficial for the respondent. The last type of heading is a step heading used in multi-step and wizard forms†. Step headings can be numbered. Nonetheless, it is not recommended for forms with a high number of steps, because it can call attention to the form length*.[9]

2.1.2.2 Buttons

Same rules as for titles apply for buttons. The button language should be brief, descriptive and prompt to click. It should express what will happen after the user clicks on the button. Buttons with the text *Place order* or *Sign up* are always better than the ones with plain *Submit*†.[9]

2.1.2.3 Messages

Messages are used for communication with a respondent. They can indicate status, highlight important information or prompt the user to act. It is a good habit to convert important instructions into questions, for example, instead of a note saying *Complete this section only if...* use conditional questions.[9]

Error messages should have polite and non-accusatory tone, definitely not blame the user and be rude. Their purpose is to convey the user where the error occurred and how to correct it.[9]

2.1.3 Required and Optional Fields

Questions can be required or optional. The *required question* means that an answer to this question is necessary to be filled. This type of question is often marked by * or *(required)* indicator. On the other hand, *optional question* can remain unanswered. This type of question is marked by *(optional)* indicator. If a form contains all fields of one type, required or optional, it should not be indicated in any way. In that case, it is unnecessary information which slows down the form completion process.[48] On the other hand, if required and optional questions are mixed, only the minority type of questions should be tagged by an indicator†. If the minority type is required, an instruction saying: *All questions are optional unless marked (required)* should be added to the top of the form†.[9]

However, if some questions are optional, a form designer should think about the fact, that the longer the form is, the lower the completion rate and respondent's satisfaction can be.[48]

2.2 Form Layout

Form layout is a graphical placement of form controls together with colours and typography.

2.2.1 Question Field Types

Most questions can be designed by one of the three basic form controls, specifically text fields, single-choice and multiple-choice questions†.[9]

Form designer should use as few form control types as possible to avoid confusion of the user*.[28] Form controls mentioned above are appropriate because respondents know them from paper forms. They are accessible and straightforward to develop because the HTML5 standard natively supports them. Other form controls known from web forms are only an enhancement of the three specified types. For example: [9]

- A dropdown is a list of radio buttons collapsed into smaller space.
- A time pickers are an alternative to text fields.
- Ranges and star ratings are an alternative to single-choice and text fields.

When these enhanced form controls are used, it is essential to develop them correctly, in order for them to be accessible, usable on touch devices and more effective compared to the basic ones. Accessibility includes a rule, that form elements and buttons should have at least 48 pixels height†[28] and the gap between form elements should be between 50% to 75% of the field height†[48]. In addition, radio buttons and checkboxes should allow interaction by clicking on their label†.

A form developer should use the HTML input type to benefit from an optimised keyboard on touch devices†. Furthermore, a form developer should specify the input name to profit from browser auto-fill feature on form fields with a specific name, such as *first name*.[43] Answer fields size should match with the expected length of the answers†.[48]

2.2.1.1 Single Checkbox

For questions designed with single checkbox, it is a better idea to use two radio buttons to avoid the mental calculation of linking checkbox state with the wording of the question*.[9]

2.2.1.2 Dropdown

While using a dropdown, a form owner should expect that many respondents do not know how to navigate through dropdowns with the keyboard. The options are initially hidden, and often, not all are visible at once. In a long list, it can be hard to find something. Thus it is a good idea to use autocomplete field instead.[9] On the other hand, if there are less than four options, the radio buttons are preferred over dropdown or autocomplete*.[46]

2.2.1.3 Time Pickers

Form elements which are related to time are a part of the HTML5 standard. However, they are not fully supported in main browsers[1].[34] That means that the developer of the form should use or develop a non-standard solution. Time pickers interface should not be the only way to input dates. A good alternative is a text field with a question-level help or a masked text*†.[38]

2.2.1.4 Ranges

Ranges are again part of HTML5 standard, nonetheless unsupported the same way as time pickers in main browsers.[34] It is a complicated way of providing a single value. It is better to use a text field or radio buttons for small ranges.[9]

2.2.2 Alignment

Questions should always be presented within the form in a single column.[2] For the label, there are four options on how to be placed within a field. Firstly above the field (*top-aligned*), secondly on the left of the field aligned to the left (*left-aligned*), next on the left of the field aligned to the right (*right-aligned*) and finally inside a field.

The top-aligned form in comparison to the left-aligned form can be filled more efficiently, with 33% time improvement, thanks to fewer fixations and saccades. However, it has to be taken into account that the time improvement was measured on form not only with changed alignment of labels, but also with some other changes, such as larger

[1] Latest version of most popular browsers which includes Chrome, Firefox, Safari, Opera and Edge. Available at: `http://www.w3counter.com/globalstats.php?year=2020&month=9`to 20/10/2020

spaces between questions, and transformation of a dropdown with two options to radio buttons.[38]

2.2.2.1 Top-Aligned Labels

The forms with top-aligned labels have better filling time than the others, because the respondents have to move only straight down.[38]. On mobile devices, it is the only way how to place the label. In case of devices with larger screens, top-aligned labels are especially useful in multilingual forms. Labels can be relatively short in one language, but in other languages twice as long. This fact can affect the form layout. On the other hand, this type of alignment takes more vertical space in comparison to the others.[48]

2.2.2.2 Left-Aligned Labels

Left-aligned labels make the reading of questions easy. Respondents can read a label in the left column and answer in the right column. However, if the labels are long, it extends the distance between the label and input field, and it takes longer to answer.[48]

2.2.2.3 Right-Aligned Labels

Forms with right-aligned labels can benefit from the fact that labels and fields are close to each other. However, most cultures read text from left to right, so users eyes prefer a hard edge along the left side.[48]

2.2.2.4 Labels Inside a Field

Label or question help should never appear within a field as a placeholder text. The only text there should be the answer. The main reason is that the user can forget the question or validation rules and is unable to reread it in case he already answered. This coheres with the fact that the user is unable to revise the form before submitting. Next, he can think that he already answered and leaves the field blank. Finally, it can be inaccessible, which depends on the browser used.[9]

2.2.3 Buttons

Buttons should visually express the importance of action. Buttons used for primary actions should be highlighted the most and left aligned under the last question†. Those rules apply especially for submitting button. Secondary action buttons should be less noticeable. Possibilities of how to distinguish buttons are by background colour, size, font size, font style and boldness.[48]

As for the buttons themselves, the reset or clear button should not be included. However, if it is required, the undo action should be provided†.[48]

2.2.4 Typography

An ideal typeface for forms is the one which is easy to read at small and larger sizes. Font style should be regular†. Italics and oblique styles should be avoided†. Bold font should be used only for titles and emphasis†.[9] The font size should be at least 16 pixels if the form is designed for touch devices because otherwise on iOS[2] devices, the form will be zoomed on each tapping†.[43]

[2] iOS - iPhone OS

Text in forms should be written in a sentence case[3]. Title case[4] and uppercase should be avoided because they are harder to read plus all uppercase is also hard to scan*.[1]

Optimal line length for general reading is between 60 to 75 characters. However, if the majority of labels in the form is short, the ideal length for a large screen is around 25 to 35 characters.[9]

2.2.5 Colours

Typically the red colour is used for errors, orange for a warning, green for success and blue for information. In forms, colours should be primarily used for messages and buttons. Messages should always be noticeable, and the colour should be in accompany with an icon and a text to highlight the problem area†.[2] The reason is that about 8% of men and 0.5% of women in the world suffer from colour blindness.[5]

It is crucial to have sufficient colour contrast. According to WCAG[5] 2.1, the contrast of non-text elements, which applies to form elements, should be at least 3:1 in Level AA[6]†. The contrast of text elements should be at least 4.5:1 for normal text in Level AA†.[4]

2.3 Flow of a Form

This section is about how the questions should follow each other, how they should be organised within a form and how to ensure the quality of data.

2.3.1 Question Order

Questions should be ordered the way things flow in the user's mind, from simple questions to the more complex ones. Related questions should be grouped together, and the same questions and options should be in a consistent order across the form.[9]

Navigating through the form should be able not only by mouse but also by a keyboard†. However, automatically navigating to the next field after completing the previous one should be avoided. Respondents do not expect it and it can confuse them.[43]

2.3.2 Form Types

To decide which form type is the best, it is useful to find the most used screen size of the majority of respondents and optimise the form in order to fit that screen size. If the form size fits less than four screens, the classic form should be used*. Otherwise, multi-step or wizard form type to have questions broken into multiple screens by natural grouping should be chosen. However, more than seven screens should be avoided.[9]

In addition, to multi-step form option, it is useful to use a progress indicator to display the respondent's progress. The progress indicator should serve to display the total number of steps and the current position within the form†.[48]

The card form type, is an effective way, how to keep users concentrated, without causing a distraction by other questions.[9] However, this type of form does not have an effect on the user's experience and completion rate, in comparison to an option with all questions on a single screen.[20]

[3]Sentence case - Only the first letter within a sentence is capitalised

[4]Title case - All words within a sentence are capitalised

[5]WCAG - The Web Content Accessibility Guidelines

[6]Level AA is a mid-range conformance level of accessibility. Available at: `https://www.ucop.edu/electronic-accessibility/standards-and-best-practices/levels-of-conformance-a-aa-aaa.html` to 20/10/2020

2.3.3 Before and After a Form

Forms, requiring a significant time to complete or knowledge which is unavailable to the user from his memory, should have a welcome screen†. The content of the welcome screen should include a brief explanation of the context of the form, external information needed to fill the form and the estimated time required to finish.[48]

After the user fills and successfully submits the form, a thanks page or a message should declare it. Depending on the case, the confirmation email can be sent, and information about what to do next can be provided†.[2]

If the form data have a significant impact on the user or the form is spread out into multiple screens, a review screen with all questions and answers provided should be considered, before submitting†.[9]

2.3.4 Validation

In case the error is caused by an error of omission, the user should be informed to provide an answer. On the other hand, if the error is caused by an error of commission, the validation message should convey what and where the error happened, together with the information on how to correct it. As mention in the layout section, it should be highlighted not only by colour but also by a symbol or a background colour†.[9]

The error information, how to fix the problematic question, should be in a normal case situated in close proximity to the problematic question. However, longer forms should also have a summary of all errors located at the top of the form†.[9]

Answers should be validated before submitting by client-side validation. If client-side validation evaluates the form errorless, it should always be validated on a server by server-side validation, prior to saving potentially dangerous data to the database.

Forms can be validated right away with an inline validation when the answer is provided. Nonetheless, many applications suffer from the problem of evaluating the answer by inline validation too soon, before the user finishes typing. Inline validation should be avoided unless the form has a small number of questions, and the majority of validation rules can be checked on the client-side†.[9]

Disabling primary action buttons is a bad practice because the form can act broken†. However, if the form is being submitted, the user should not be able to hit the button twice†. Moreover, if the process is not finished immediately, an information note about processing should be displayed†.[9]

2.3.5 Default Answers

Default answers are usually a bad idea, unless a form designer has data which shows with high probability the answers respondents will choose. Besides, if respondents do not change the default answers, it will have little impact on the quality of data*.[9]

However, a good concept is to use personalised default answers, which means the form is pre-populated by answers already known about the user by the provider of the form.[48]

2.3.6 Conditional Logic and Branching

The designer of the form should strive not to cause a dead end, by conditional questions*. A excellent example of using conditional questions is for screening the users. If the form is inapplicable for all users, it is recommended to start the form with the screening via eligibility questions and let respondents continue in filling the form in case they pass.[9]

Chapter 3

Existing Solutions

For many years, data were captured using paper forms. Forms were designed, printed and distributed to complete. Having collected the filled forms, researchers had to process massive amounts of data to obtain results. This chapter is dedicated to the analysis of the tools used for building electronic forms used in the digital sphere. The analysis will be beneficial for the design of the Semantic Form Editor.

For a more in-depth analysis, seven non-semantic web form editors were chosen. SurveyMonkey, JotForm, forms.app and Google Forms represent standalone web applications for design and management of forms. In addition, three solutions with the purpose of creating a structure of a form that has to be integrated into the form owner's application were selected. These were namely React Hook Form, Bootsnipp and formBuilder. As for the editor using the Semantic Web technologies, only one application was found. It is TopBraid EDG application which allows displaying forms generated from SHACL and DASH languages.

There are dozens of great form editor options to choose from on the market. Those seven form editor solutions were selected because they were easily found by Google[1] or Capterra[2] , and they differ from each other. They offer a wide range of different features. The user interfaces are easy to use, and either they are entirely free or offer free trials of premium features. The TopBraid EDG application was pointed out by my supervisor.

3.1 Standalone Form Solutions

A standalone form solution is a cloud-based software for efficient design and management of electronic surveys, polls and questionnaires. Form owners can use them to connect with their target group of respondents and collect relevant data for their businesses, researches and more. After the form is designed, it is ready to be used. Results can be analysed at any time during the collection process directly in applications. This applies for all analysed applications in this section except for the forms.app presented in Section 3.1.3.

3.1.1 SurveyMonkey

SurveyMonkey[3] was founded in 1999. It is an online cloud-based software for surveys, quizzes, and polls. It is well-known, and on average, 20 million people answer questions daily using SurveyMonkey.[18] The basic features are provided free of charge with

[1] Google search engine available at `https://google.com/` to 10/6/2020

[2] Capterra is an application used for finding and comparing of software solutions. Available at `https://www.capterra.com/survey-software/` to 10/6/2020

[3] SurveyMonkey available at `https://www.surveymonkey.com/` to 24/10/2020

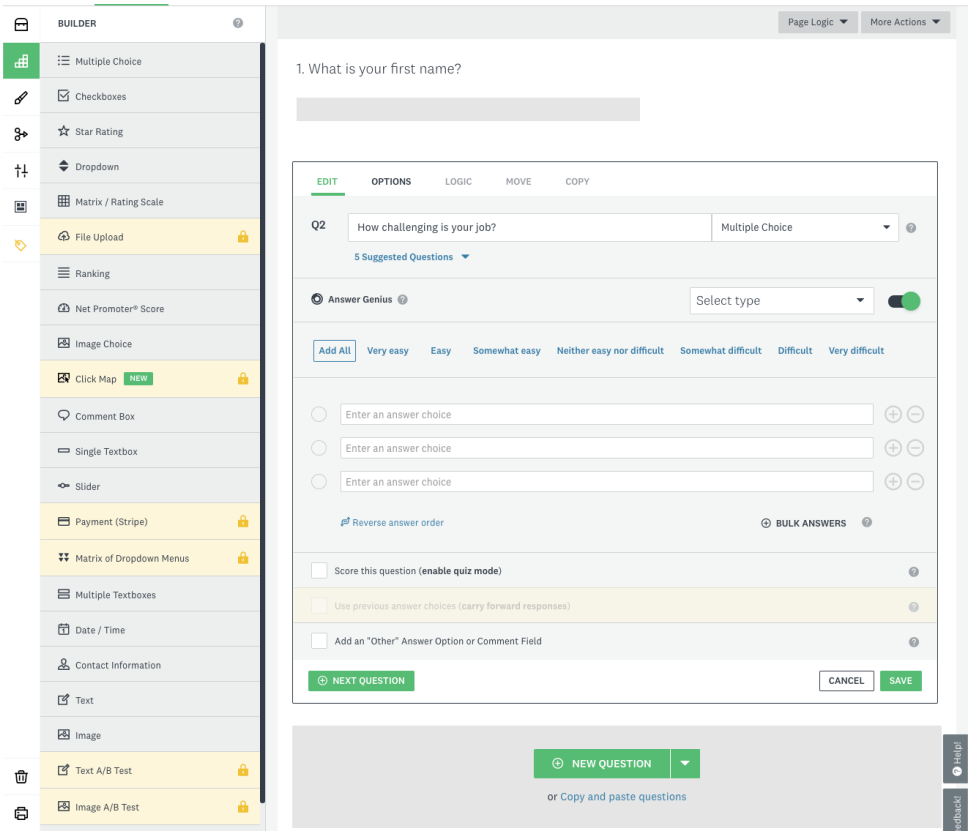
the limitation of 10 questions and 40 views of responses per form. The more advanced features are included in five different subscription plans.[18]

Form designers can build a form using four different methods. Firstly, by choosing one of the ready-to-use templates from different categories. Secondly, by using prewritten questions. These questions can be prepared, e.g. in a text-editor and must follow a syntax that the form-editor uses for such prepared questions inputs. With this feature, it is possible to add both simple text field questions and multiple-choice questions. The third option is to use a question bank with more than 2,000 prepared certified questions along with predefined answer options from different categories. Finally, the form designers can create a form by using Drag and Drop API for questions creation. All these building options can be combined together.

SurveyMonkey offers basic form controls as well as the custom ones. Some of the custom ones are image choice, star rating, ranking, matrix of single choice, matrix of multiple-choice, matrix of dropdowns or A/B testing[4] text field. Other form controls cannot be added to the form by the form designer. The title and subtitle are available at the top of each step, and the submission or navigation buttons are available at the bottom.

When the designer specifies the question and similar question is available in the question bank, a prompt message proposes to use that question. Furthermore, the proposal of different option answers is provided during closed questions design. It is also possible to add an *Other answer option*, which is described in Section 3.4. The form designer can configure many different attributes of each question, for example, answer validations, question fields length, randomisation of options or conditional logic. The labels are top-aligned.

Figure 3.1: SurveyMonkey in a process of adding multiple-choice question

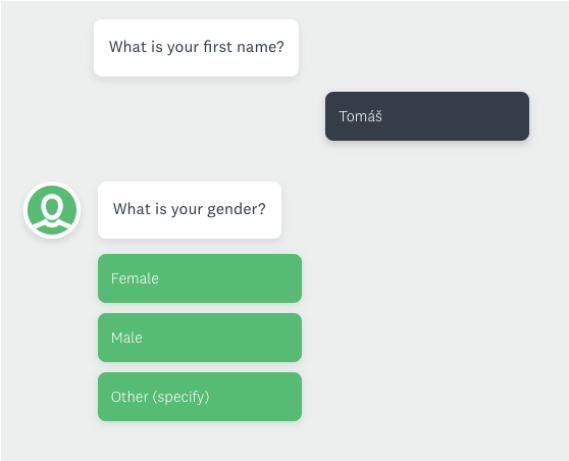


[4] A/B testing is a method used for comparing two versions of an application. More information available at `https://www.optimizely.com/optimization-glossary/ab-testing/` to 24/10/2020

Figure 3.1 shows the SurveyMonkey form editor with the process of adding a closed multiple-choice question. User can choose out of five suggested questions from a question bank or add predefined answer options.

There are three types of layouts: card form, classic form and multi-step form are presented as one type. The last type of layout, shown in Figure 3.2, is a conversation which is currently in the testing mode. The chatbot asks questions, and the user answers them.

Figure 3.2: A conversation form type in SurveyMonkey



A form designer can add an introduction and thanks page. In addition, displaying of required question indicators or randomisation of whole pages can be configured. Created forms can be multilingual. The graphical design can be customised by predefined themes and is fully responsive. The created form can be reviewed in the preview section, where the form designer gets recommendations, how to improve the form.

Team collaboration is provided and lets people on their team view, edit and comment forms in real-time. Created forms are hosted. Answers can be collected on SurveyMonkey web, in mobile apps, on researcher web using the embeddable view or printed on paper. Results can be analysed on the website, in the app or exported in PDF, XLS, CSV or PPTX formats. SurveyMonkey also provides REST API, which allows to, for example, create and modify surveys or get respondents' answers, without using SurveyMonkey application.

Based on the research, the application is user friendly from my point of view, and the user interface is intuitive and well-arranged. The form design follows the best practices from Chapter 2 except for a key one. It is impossible to specify question-level help or any help at all. Thus the user can unconsciously fill invalid answer and find it out only after submitting the form because inline validation is neither available. The application contains many useful features and can be used in a variety of ways.

The application was analysed in October 2020.

3.1.2 JotForm

JotForm[5] is an online form editor which allows creating forms and collection of data. It was founded in 2006 and is trusted by more than eight million users. All features are provided free of charge with limitations of five forms, 100 monthly submissions and 1,000 monthly form views in a free plan. To increase limits, JotForm offers three subscription plans.[17]

[5]JotForm available at `https://jotform.com/` to 24/10/2020

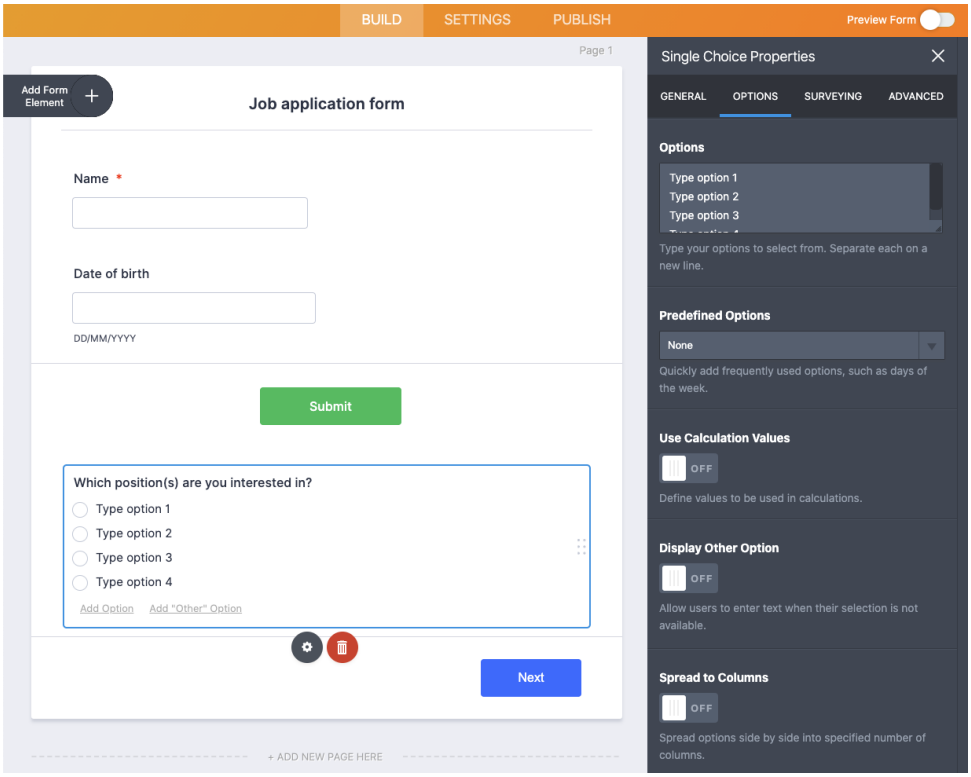
Form designers have four options on how to start building forms. The first option is to choose from about 5,000 of different prepared form templates. Next options offer to create a form from scratch or to clone an existing form created using JotForm. Finally, the last option is to import an existing PDF form which is then converted to an online form.

There is a vast amount of elements to choose from, that can be added to the form by dragging. Together with the basic and custom form fields, there are also widgets. Some of the custom form controls are, for example, scale rating, signature, spinner and fill in the blank. The designer can also put headings and button for submitting to any location of the form. Widgets integrate new functionalities to the forms. It is possible to add, for instance, a bar code scanner, a voice recorder or a code editor. A lot of different attributes of questions can be configured, such as label alignment, question fields width, answer validations and conditional logic and branching.

A form designer can create a classic or a multi-step form, which are presented as one type, as well as a card form. The design of the form is highly customisable. It can be styled by provided themes, colour schemes or CSS. It is also possible to change the font to one of the many provided. Created forms are fully responsive.

It is possible to collaborate with the team during the form building process. The application offers to create not only digital forms but also provides an editor for building PDF forms ready to be printed. Nevertheless, JotForm is not only a form editor, but also offers hosting of the forms and the following analysis and visualisation of collected data. Form owners can collect data for built forms using JotForm web or mobile apps, by embedding the form to the organisation website or by paper form.

Figure 3.3: Jotform in a process of adding a single-choice question



Moreover, it is possible to integrate over 400 third-party application add-ons. The examples are payment gateways for selling goods using the forms, Google Sheets to populate spreadsheets with the collected data or Google Drive to upload obtained files. Collected data can be exported in PDF, XLS and CSV format or used for creating visual

reports, which are available to download in PDF format or to be presented online.

Based on the analysis, the application is from my perspective user friendly with intuitive and easy to use user interface. Forms can be created according to best-practices, except for two problems. The first problem is the missing of a progress bar in multi-step forms. The second one, JotForm does not guide the user to avoid problems. It is easy to build malfunctioning forms because the form editor allows for excessive customisation. For example, it is possible to create forms violating the basic principles as placing the submit button to any position within the form, without any warning from JotForm. An example of the form editor and the wrong placed submit button, can be seen in Figure 3.3. However, for experienced designers, it is a powerful tool which contains many useful features and can be used in all types of businesses and organisations.

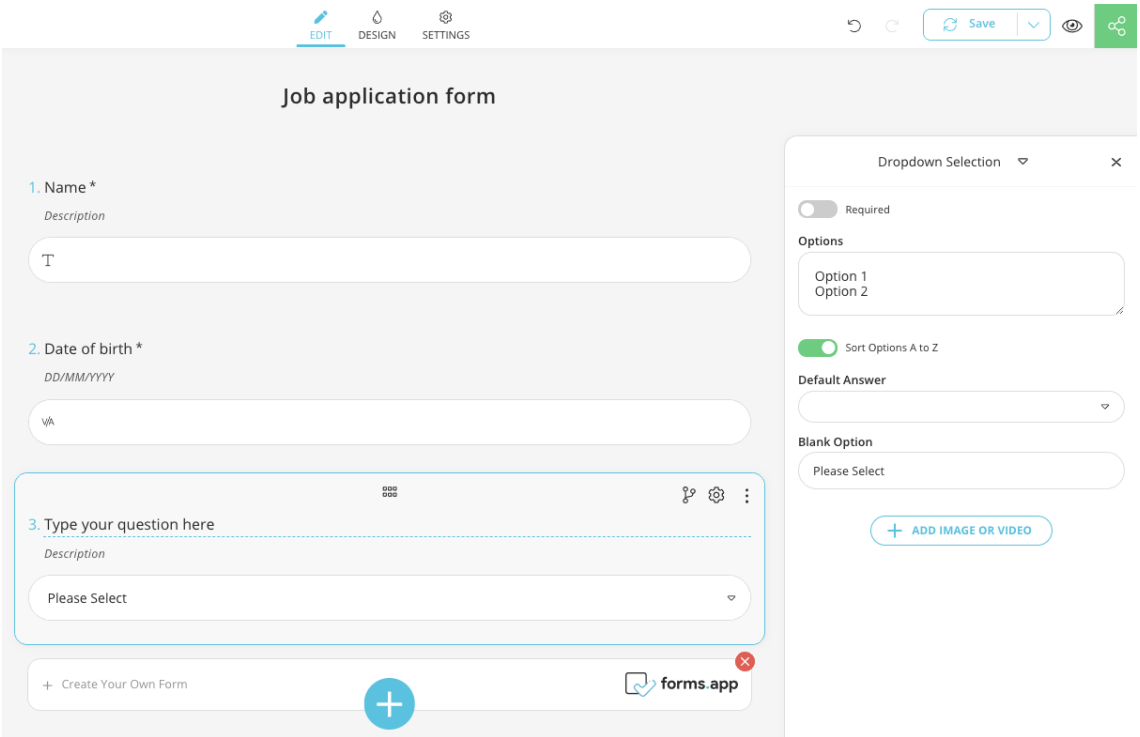
The application was analysed in October 2020.

3.1.3 forms.app

The form editor forms.app[6] was founded in 2018. All features are provided in a free plan, but there is a limitation of 250 form answers and 1,000 form views per month. Better limits are offered in three paid plans.[10]

The form editor supports only a classic form type and offers 13 templates to start with. There are 26 form elements to choose from. Together with the basic ones, there are scale rating, star rating, multiple-choice matrix, image choice, and for selling goods — a payment widget and a product basket. For each question, the editor allows configuring conditional logic and branching, validation rules, question-level help, and more. In Figure 3.4, the configuration of a dropdown question is shown.

Figure 3.4: Forms.app form editor in a process of adding a dropdown question



The configuration of the forms contains, for example, displaying of a form title, showing a field order number, a welcome and a thanks page. For design, a few predefined

[6] forms.app available at `https://forms.app/` to 27/10/2020

themes, many different fonts, and custom CSS can be used. Resulting forms have responsive design.

Collecting answers is possible on the website, in mobile applications or by embedding the form to the form owner's website. Answers can be afterwards exported in XLS format. In addition, it is possible to integrate over 300 third-party add-ons, such as Google Sheets to save collected data, Google Drive for storing collected files or project management apps like Asana, Jira, and Redmine.

After further analysis, the application misses many features in comparison to SurveyMonkey and JotForms. For example, it is impossible to create questions in a specific place. The new question is always added to the end of the form. However, it is possible to order already existing questions by Drag and Drop API. Having applied the best-practices from Chapter 2, I found one problem. It is impossible to customise the length of the field to fit the answer. The interface is, in my opinion, intuitive and user-friendly, but the colours are too light.

The application was analysed in October 2020.

3.1.4 Google Forms

Google Forms[7] started its service in 2008. It is entirely free of charge for personal use. For business, it is included in Google Workspace[8] plans.[16]

If a user does not want to begin with a blank form, there are 18 templates of forms to start with. The form designer cannot choose the form type in advance, but instead, he can arrange the questions to comply with the structure of a classic, a multi-step, and a card form.

There are 13 types of form controls to choose from. Except some of the basic ones, there are scale rating, single-choice and multiple-choice matrix and the possibility to add paragraph and media content, such as image and video. It is also possible to add an *Other answer option*. Another way to add questions is to import them from other existing forms. Conditional logic and branching is available only for multiple-choice and dropdown questions. However, in comparison with precedingly analysed tools, Google Forms does not allow displaying or hiding specific questions according to an answer. Instead, it is solved by navigating a respondent to a related form step based on the answer provided. The order of questions and answer options can be randomised.

The editor supports the collaboration of teams in real-time. Customising of the forms looks is limited. There are 12 theme colours, four background colours, four types of fonts, and a header image can be set.

Google Forms are a part of the Google Workspace ecosystem, and the collected data can be immediately sent to Google Docs and analysed. The form editor can be extended by third-party add-ons, for example, QR code scanner or Form Values[9] add-on, which adds functionality to import predefined answer options from Google Docs[10].

Based on the analysis, Google Forms have a basic feature set and aim for simplicity. However, for me, the form editor looks complicated and confusing in comparison to the three preceding tools. Questions can be added by clicking on the add button in a floating bar next to the questions. Afterwards, a new question is added under the currently selected question. Ordering of questions is possible by Drag and Drop API. The configuration of questions is directly in the question box. For each question type, the placement

[7] Google Forms available at `https://www.google.com/forms/about/` to 27/10/2020

[8] Google Workspace available at `https://workspace.google.com/` to 27/10/2020

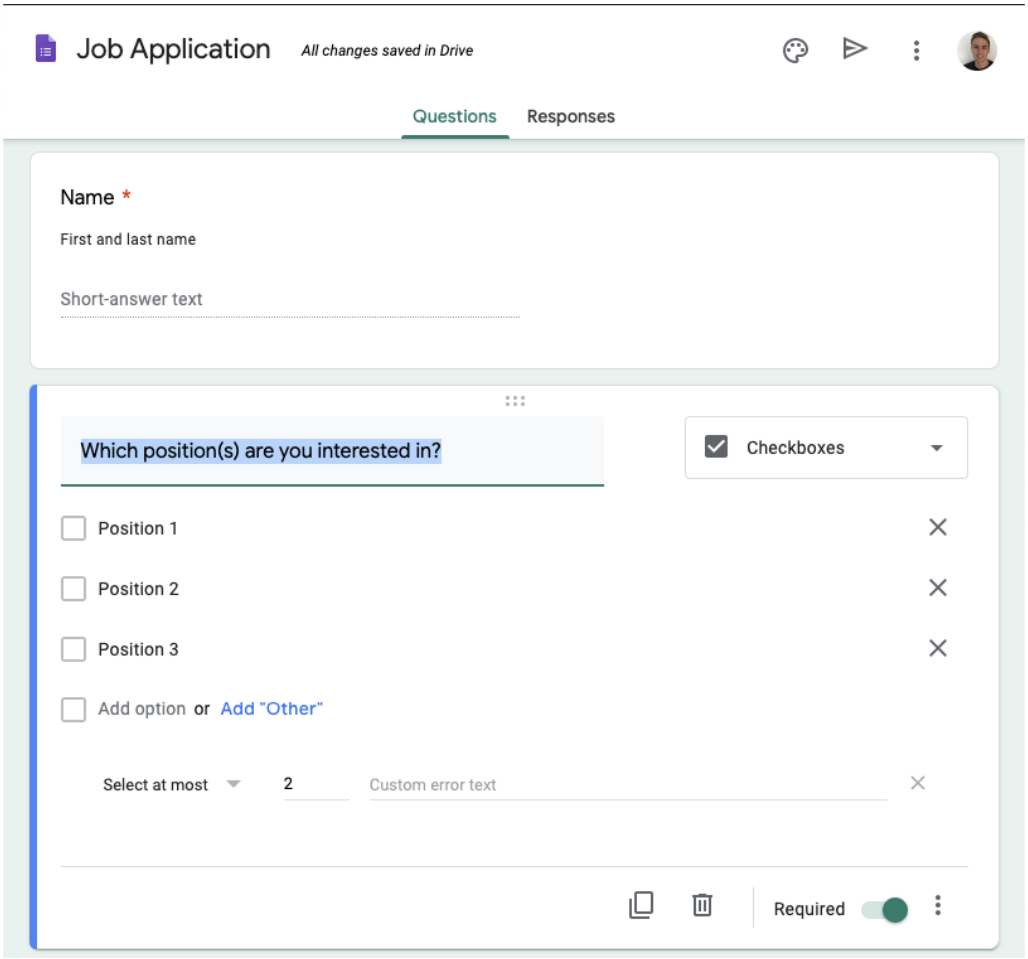
[9] Form Values add-on available at `https://gsuite.google.com/marketplace/app/form_values/675910425109` to 27/10/2020

[10] Google Docs available at `https://docs.google.com/` to 27/10/2020

of customising actions is inconsistent and difficult to navigate. The example of a form editor in the process of adding a multiple-choice question can be seen in Figure 3.5. When applying best-practices, it is impossible to set the length of the field according to the expected answer and display the progress bar in a multi-step form.

The application was analysed in October 2020.

Figure 3.5: Google Forms form editor in a process of adding a multiple-choice question



3.2 Form Editors

In comparison with standalone form solutions described in Section 3.1, the form editors discussed in this section are only for building forms in JSON, HTML, and React component format. The following development of an application for distribution of the form, collection of data, and analysis, all have to be resolved by form designers on their own.

3.2.1 React Hook Form

React Hook Form[11] is a React library which can be added for free to React projects through the package manager. It has around 480,000 weekly downloads from the npm[12], and new versions are released at least ten times a month. It is a library which allows a developer to create performant forms with less code.

[11] React Hook Form available at `https://react-hook-form.com/` to 28/10/2020

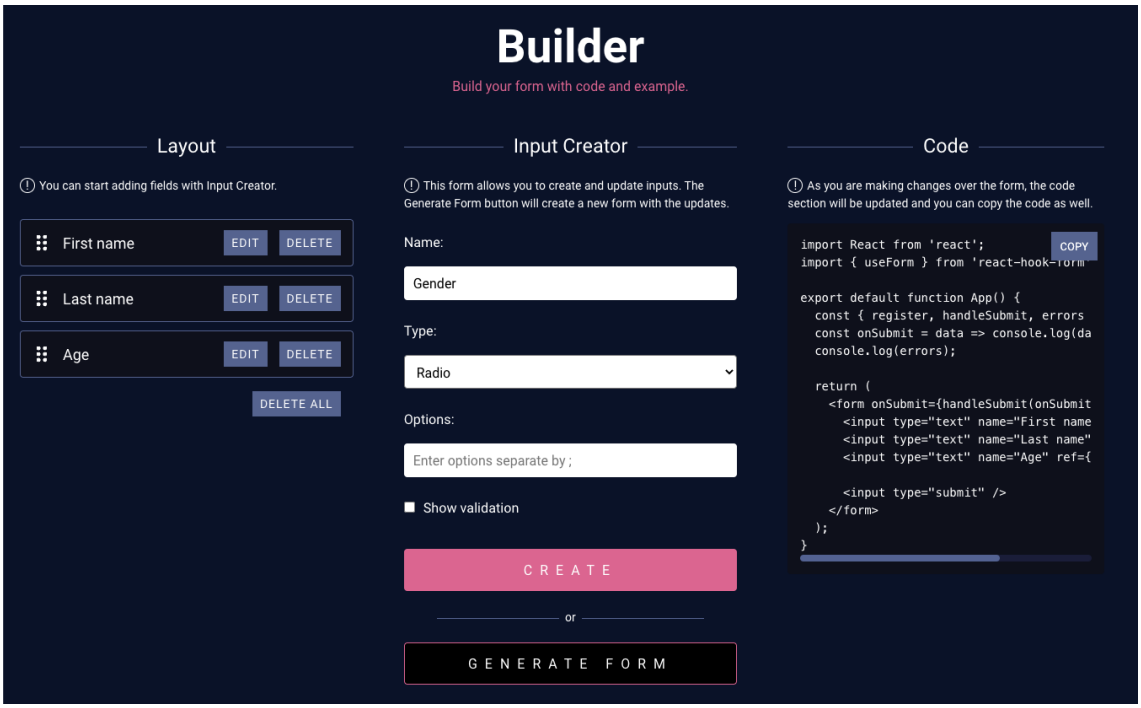
[12] Node package manager available at `https://www.npmjs.com/package/react-hook-form` to 28/10/2020

A developer can use a form editor available on a library website to design simple forms. The resulted form code can be afterwards copied to the developer's project. The developer can build a form out of 16 standard HTML input field types with configurable validation. It is possible to specify required questions, the maximum length of the input, regex pattern, minimum and maximum value. Other validation properties have to be added manually in the code. The Drag and Drop API can be used to organise form controls in the layout. The library is compatible with a large number of other form controls which can be added to the project with the help of other libraries. However, they are unavailable in the editor.

Based on the analysis, the form editor is simple and does not offer much functionality. There is only adding, editing, deleting and ordering of questions. However, for an undemanding user, the editor is sufficient or can be used for creating a skeleton of the form. The editor is intuitive, easy to use, and the developer can see the resulting code on the right side of the screen. The example of the form editor with three created questions can be seen in Figure 3.6. The documentation of the library is extensive and contains examples, as well as information about how to develop and further improve the forms. For instance, there is a tutorial, how to implement a wizard form type.

The library was analysed in October 2020 on 6.9.6 version.

Figure 3.6: Example of React Hook Form form editor



3.2.2 Bootsnipp

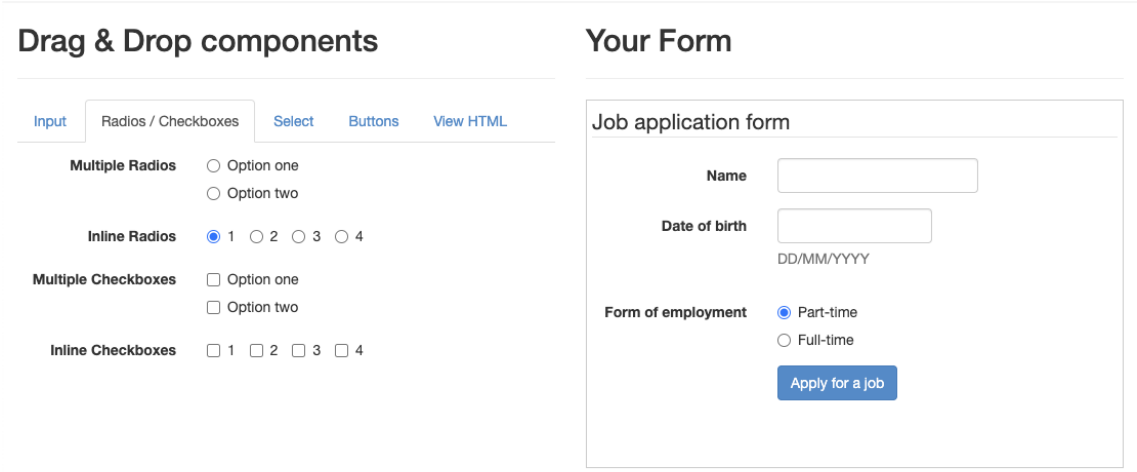
Bootsnipp[13] is a website with a gallery of code snippets and a playground for developers using Bootstrap. A developer can find here more than 1,000 different ready to use templates using the Bootstrap and useful tools — for example, page builder, form editor and button builder.

Using the Bootsnipp form editor, it is possible to build a classic form out of text fields, text areas, multiple-choice and single-choice questions, dropdowns and a file uploader.

[13] Bootsnipp available at `https://bootnipp.com/` to 28/10/2020

In addition, it is possible to add a single button or a double button, which are, in fact, two buttons next to each other, serving for two independent actions, such as save and cancel. On the left side of the screen, a form designer chooses an element with the specific layout and drags it to the right side of the screen to the required position. The example of a process of creating form can be seen in Figure 3.7. For an open question, it is possible to specify a label, placeholder, question-level help, input size and required answer. For a closed question, the developer can specify a label and answer options. When the form designer finishes building of the form, he can copy the resulting HTML code of the form with classes to his own project importing Bootstrap 3.

Figure 3.7: The Bootsnipp form editor example



After further analysis, the form editor is easy to use and does not have many functionalities. However, if the developer modifies the resulting code, applies some additional CSS and specifies validation rules, he can build a form which satisfies best-practices. The problem with the form editor is that it builds the form in Bootstrap 3. However, the current major version is Bootstrap 4 which is incompatible with the version the form editor is currently working with.

The form editor was analysed in October 2020.

3.2.3 formBuilder

The formBuilder[14] is a JavaScript library for building and displaying forms. It is an active project, which has 1,973 stars on GitHub[15].

The form designer can, using the formBuilder, create a classic or a wizard form. By dragging, it is possible to add ten different basic form controls, for example, the autocomplete question. In addition, it is possible to add headers, paragraphs and buttons. The number of form elements is increasable by coding.

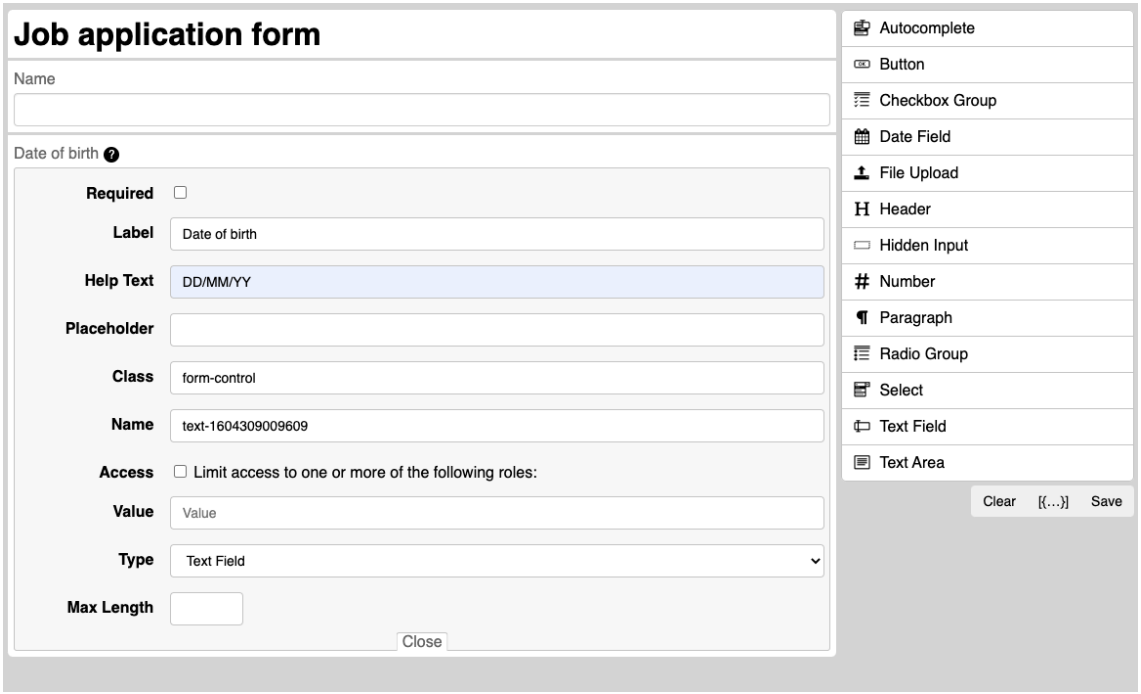
For each question, it is possible to specify a label, a placeholder, an HTML class for custom theming, a default value, a required answer and a help. From the validation point, for the text field, only the maximum length of the answer is configurable. For number field, it is the minimum and maximum number. The resulting form code is in JSON format which is then used for rendering of the form. The library is configurable

[14] formBuilder available at `https://formbuilder.online/` to 30/10/2020

[15] formBuilder on GitHub available at `https://github.com/kevinchappell/formBuilder` to 30/10/2020

and has extensive documentation. The formBuilder in a process of adding a text field is shown in Figure 3.8.

Figure 3.8: The formBuilder form editor in a process of adding a text field



From my point of view, the form editor is well designed and working as expected. However, I miss the possibility to set more validation rules for questions. When applying best-practices to created forms, the first problem is the question help. It is hidden under the question mark, which can unnecessarily slow down the completion of forms. Those problems can be solved by modifying the code of the library.

The form editor was analysed in October 2020 in version 3.6.1.

3.3 Semantic Form Editors

The semantic form editors are applications for creating forms based on the Semantic Web technologies.

3.3.1 Generating of SHACL and DASH Forms for TopBraid EDG

TopBraid EDG[16] is a web application for creating and managing domain models and ontologies in the RDF, RDFS and OWL. Although SHACL and DASH are initially designed for data validation, it is possible to use them for building forms for TopBraid app.[23]

Generated forms are defined in a shapes graph using the SHACL vocabulary and a DASH namespace[17]. The DASH namespace introduces a variety of properties, classes, and instances useful for building forms and validating form data. One of the properties useful for generating forms is *dash:editor*. A form designer can use one out of 14 instances of *dash:editor* property. These instances represent form controls for filling the data and some of the available ones are, for example, an autocomplete, a text area, a datetime, a

[16] TopBraid EDG available at `https://www.topquadrant.com/products/topbraid-edg-gov-packs/`

[17] More information about *dash* namespace available at `http://datashapes.org/dash` to 30/10/2020

dropdown, a text field with a language dropdown, a text field for URI, and an HTML text editor with a language dropdown.[23]

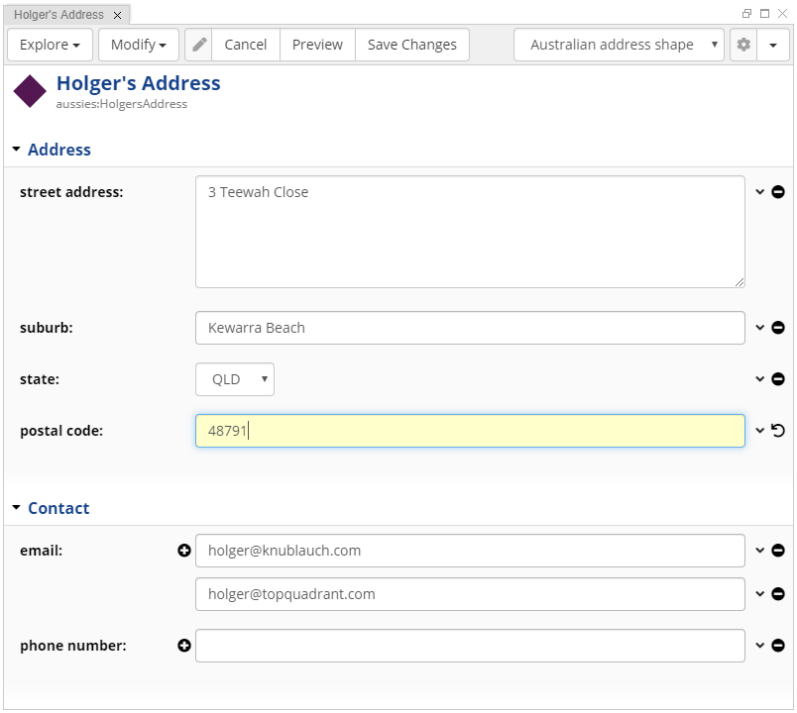
For diplaying of the data, there are form control viewers defined in *dash:viewer* property. Some of them are, for example, a HTML viewer, a viewer of text with language, and a URI viewer.[23]

The form controls can be either defined by the property *dash:editor* or alternatively resolved by a scoring system. For instance, if property *dash:editor* is of instance *dash:TextAreaEditor*, the text area is used. However, if property *dash:editor* is unspecified, the corresponding form control is selected by the scoring system in a following way. If property *sh:datatype* is specified as *xsd:string*, it counts as a score of ten, and if there is no other property increasing the score, the text field is chosen. However, if there is another property, e.g. *dash:singleLine*, defined as *false*, the text area form control is chosen instead. Some of the useful SHACL and DASH properties related to forms can be seen in Table 3.1.[23]

Table 3.1: SHACL and DASH form related properties

Property	Example values	Description
sh:datatype	xsd:boolean, xsd:date, xsd:string, xsd:datetime	One of the factors by which the form control can be chosen.
sh:order	"0"^^xsd:decimal	An order of form controls within a form.
sh:name		The label of form controls.
dash:editor	dash:AutoCompleteEditor, dash:TextAreaEditor	It specifies which form control should be used for editing of data.
dash:viewer	dash:HTMLViewer, dash:LangStringViewer	It specifies which form control should be used for viewing of data.
dash:singleLine	true, false	One of the factor by which the form control can be chosen.

Figure 3.9: Example of TopBraid EDG with the form generated using SHACL and DASH



Source: http://datashapes.org/forms.html

The example of TopBraid EDG application with generated form using SHACL and DASH can be seen in Figure 3.9. The corresponding part of a shapes graph used for rendering a street address text area form control can be seen in Source code 3.1.

Source code 3.1: Shapes graph rendering a text area form control

```
ex:StreetAddress
    a sh:PropertyShape ;
    sh:path ex:address ;
    sh:datatype xsd:string ;
    dash:singleLine false ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:name "street address" ;
```

The analysis is based on the unofficial draft[18] and was performed in November 2020.

3.4 Summary

The research helps to identify important functionalities that a form editor should have as well as helps avoid mistakes and incorrect solutions during the form editor design.

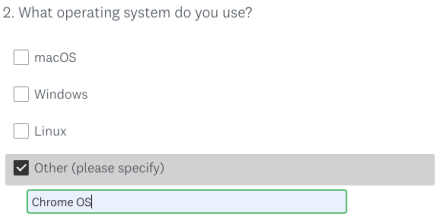
In Table 3.2 and Table 3.3, there are comparison summaries of all analysed solutions in this chapter. The overview includes some of the requirements of features of the Semantic Form Editor, which will be discussed in Chapter 5. For better understanding, the properties are explained below:

- **Form types** enumerate, which form types from Section 1.1.3.3 are supported by a specific form editor.
- **Reusability of questions** specify a possibility to reuse an existing question from the current form, from a different form or from a predefined list of questions.
- **Reusability of answer options** discuss the possibility to include already defined answer options from a different question in the form, from a different form or from a predefined list of answer options.
- **Validation of form design** mean that a form editor provides a possibility to validate a form against errors and best-practices of form design.
- **Forms satisfying form design guidelines**, mean that a form editor allows to create forms, which satisfy the form design best-practices from Section A.2.
- **Customisable design of questions and answer options** means that a form editor allows setting visual appearance to individual questions and answer options.
- **Conditional logic and branching** mean that a form editor supports the functionality described in Section 1.1.3.2.
- **The tree structure of forms** mean that a form editor allows the multi-level nesting of questions. In other words, the section form control is provided and can be used for nesting of questions.
- **Ordered and unordered questions** mean that a form editor allows setting fixed order of specific questions and a random order (defined, e.g. by sorting algorithm) of the rest.

[18] Available at `http://datashapes.org/forms.html` to 3/11/2020

- **Ordered and unordered answer options** mean that a form editor allows setting fixed order of specific answer options and a random order (defined, e.g. by sorting algorithm) of the rest.
- **Other answer option** means that a form editor supports adding the *Other answer option* to closed questions, which transforms the closed question to the semi-closed one, which allows respondents to type an answer, which is unavailable in the set of answer options. An example of this answer option can be seen in Figure 3.10.

Figure 3.10: The other answer option



Source: https://www.surveymonkey.com/

- **Multilingual forms** mean that a form editor supports creating a form in multiple languages.
- **The Semantic Web forms** mean that a form is defined by using the Semantic Web technologies and therefore can profit from Linked Data.

3.4.1 Summary of Standalone Form Solutions

Table 3.2 shows comparison of selected properties of analysed standalone form solutions.

Table 3.2: Comparison of selected properties of standalone form solutions

	SurveyMonkey	**JotForm**	**forms.app**	**Google Forms**
Form types	Classic; Card; Multi-step; Conversation	Classic; Multi-step; Card	Classic	Classic; Multi-step
Reusability of questions	Question bank; By duplication	By duplication	By duplication	Import from other form; By duplication
Reusability of answer options	Predefined options	No	No	By add-on
Validation of form design	Yes	No	No	No
Forms satisfying form design guidelines	No	No	No	No
Customisable design of questions and answer options	Yes	No	No	No
Other answer option	Checkbox with a text field or only a text field	Checkbox with a text field	Checkbox with a text field	Checkbox with a text field
Conditional logic and branching	Yes	Yes	Yes	Yes
The tree structure of forms	No	No	No	No
Ordered and unordered questions and answers	No	No	No	No
Ordered and unordered answer options	No	No	No	No
Multilingual forms	Yes	No	No	No
The Semantic Web forms	No	No	No	No

3.4.2 Summary of (Semantic) Form Editors

Table 3.3 shows comparison of selected properties of analysed (semantic) form editors.

Table 3.3: Comparison of selected properties of (semantic) form editors

	React Hook Forms	**Bootsnipp**	**formBuilder**	**Generating SHACL and DASH forms in TopBraid EDG**
Form types	*	*	Classic; Wizard	Classic
Reusability of questions	No	No	By duplication	By reusing a code of shapes graph
Reusability of answer options	No	No	No	By reusing a code of shapes graph
Validation of form design	No	No	No	By specification in shapes graph
Forms satisfying form design guidelines	*	*	No	By specification in shapes graph
A customisable design of questions and answer options	*	*	Yes	?
Other answer option	No	No	No	?
Conditional logic and branching	*	*	No	Yes
The tree structure of forms	No	No	No	Yes
Ordered and unordered questions and answers	*	*	No	?
Ordered and unordered answer options	*	*	No	?
Multilingual forms	No	No	No	Yes
The Semantic Web forms	No	No	No	Yes

* - Depends on the implementation of a rendering application.
? - An unknown information.

Chapter 4

S-Forms

This chapter analyses the SForms library actively developed by Mgr. Miroslav Blaško Ph.D. and Ing. Martin Ledvinka at KBSS[1] at FEE CTU in Prague. The library is used for visualisation of ontology-based smart forms defined in JSON-LD format. In this format, questions, answers, answer options, as well as the behaviour of a form, such as a conditional logic and branching, are defined as graph nodes.

The analysis of the application in this chapter is carried out in version 0.1.8[2] of the SForms library to date 3/4/2020.

4.1 Current State

The SForms is a library, not a standalone application. Therefore it has to be integrated into an existing application. One of the example usages is Study Manager.[21] The Study Manager is an eCRF management application used for retrospective clinical trials. It is based on the Semantic Web technologies. The application consists of a static part which includes users, institutions and patient records management. The dynamic part uses the SForms library for rendering externally defined eCRFs which are a part of patient records. They are fetched using the REST API from backend. The backend obtains the data from a semantic database called *triple-store* and uses SPipes[3] to assembly the form into a final shape.

As already mentioned, in order to use the SForms library, it is required to integrate it into an existing application. The library is stateless, and the data have to be provided and managed by a container application using the Reflux[4] state management library. In addition, only the classic form type view is available in the library. However, the Study Manager application provides a wrapper for a wizard form type, which can be used with the horizontal or vertical orientation of a wizard navigation bar.

Figure 4.1 shows an example of the Study Manager application with the horizontal wizard form visualised using SForms. Only the section *Inclusion criteria* with its content is rendered using the library, everything else, including the buttons, is a part of the Study Manager application.

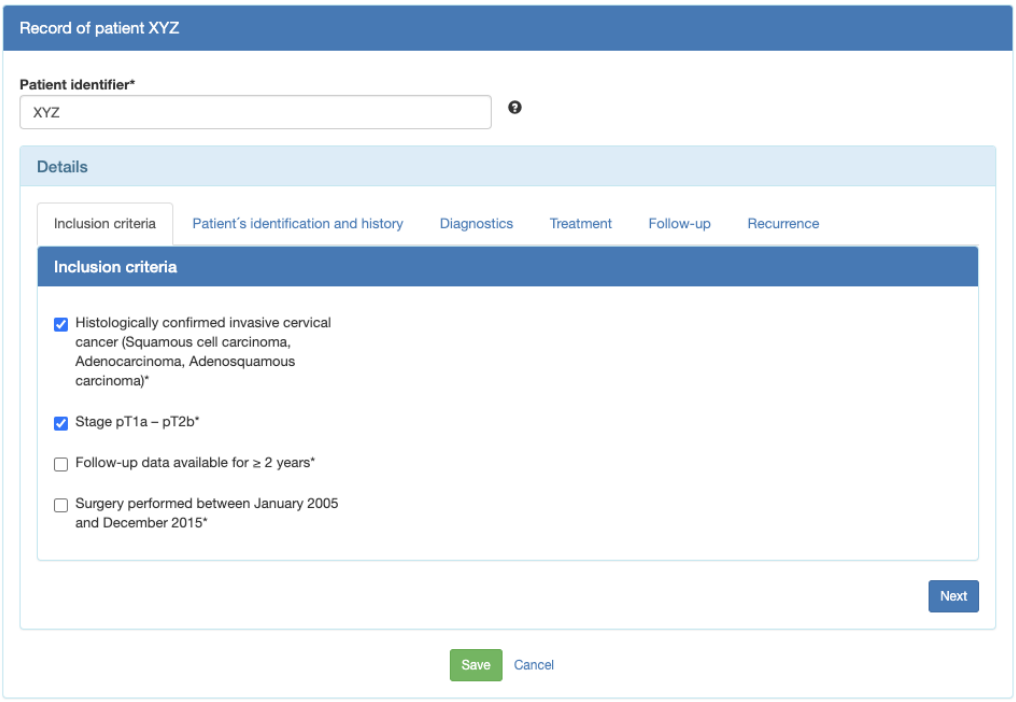
[1] Knowledge-based and Software Systems Group. More information available at `https://kbss.felk.cvut.cz/web/kbss` to 15/11/2020

[2] Source code of the SForms library in version 0.1.8 available at `https://github.com/kbss-cvut/s-forms/tree/40a9bed6dcffd94a637cf277674a386e104d89d3` to 15/11/2020

[3] SPipes developed by KBSS. More information available at `https://kbss.felk.cvut.cz/web/kbss/s-pipes` to 15/11/2020

[4] More information about Reflux state manager available at `https://www.npmjs.com/package/reflux` to 15/11/2020

Figure 4.1: The example of SForms in the Study Manager application.



Applying the best-practices from Chapter 2, the form visualisation has many shortcomings. For example, there is no question-level help, only the help displayed on mouseover. Next, the form controls and buttons have insufficient height, and the font size is smaller than 16 pixels.

4.1.1 Technology Stack

The library is developed in JavaScript using frameworks and libraries:

- **React 15**
- **React Bootstrap**[5] in version 0.30.5 which uses Bootstrap 3
- **Babel**[6] **6** used for code compilation
- **Browserify**[7] used for code bundling
- **Jasmine**[8] for testing

In addition, it uses the Semantic Web libraries such as *jsonld*[9] or *jsonld-utils*[10]. As well as, libraries for autocomplete[11] and time picker form controls[12]. The codebase is

[5] More information about react-bootstrap library available at `https://react-bootstrap-v3.netlify.app/` to 15/11/2020

[6] More information about Babel available at `https://babeljs.io/` to 15/11/2020

[7] More information about Browserify available at `http://browserify.org/` to 15/11/2020

[8] More information about Jasmine available at `https://jasmine.github.io/` to 15/11/2020

[9] jsonld library available at `https://www.npmjs.com/package/jsonld` to 15/11/2020

[10] jsonld-utils library developed by KBSS available for downloading at `https://kbss.felk.cvut.cz/dist/` to 15/11/2020

[11] react-bootstrap-typeahead library developed by KBSS available for downloading at `https://kbss.felk.cvut.cz/dist/` to 15/11/2020

[12] react-bootstrap-datetimepicker library developed by KBSS available for downloading at `https://kbss.felk.cvut.cz/dist/` to 15/11/2020

written in JavaScript ECMAScript 6 (ES6[13]) standard combined with ECMAScript 5 (ES5) standard.

4.1.2 Form Controls

The SForms library supports form controls displayed in Table 4.1.

Table 4.1: SForms form controls overview

Form control	Note
Text field	A text field is converted to a text area if it contains more than 50 characters.
Number field	
Text area	
Dropdown	
Autocomplete	An autocomplete field is provided by an external library *react-bootstrap-typeahead*.
Masked text	A mask is provided by an external library *inputmask-core* and implemented into a text field.
Single checkbox	Usable for multiple-choice in combination with a section.
Time pickers	A date, a time and a datetime are available. The form controls are provided by an external library *react-bootstrap-datetimepicker*.
SPARQL field	A parser for displaying and modifying SPARQL queries provided by an external library *yasgui-yasqe*[14] and implemented into a text area.
TURTLE field	A parser for displaying and modifying TURTLE semantic format provided by an external library *yasgui-yasqe* and implemented into a text area.
Media content	Renders HTML iframe[15] with content specified by URL.
Section	Can be collapsed or expanded and can contain other form controls except for a wizard step.
Wizard step	A step of a wizard form type.

All form controls (except for media content) can be set hidden or disabled. In addition, the section can be collapsed or expanded by default.

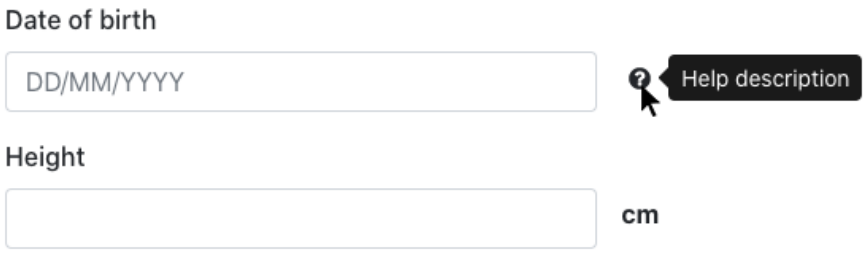
The form control consists of a top-aligned label, a field and optionally a question help hidden under a question mark and a unit. All mentioned parts of the form control can be seen in Figure 4.2. The first question is a masked text with a specified mask and a help displayed on mouseover. The second question is a text field with a defined unit.

[13] More information about ECMAScript standards available at `https://www.w3schools.com/js/js_versions.asp` to 15/11/2020

[14] More information about yasgui-yasqe available at `https://www.npmjs.com/package/yasgui-yasqe` to 15/11/2020

[15] More information about HTML iframe available at `https://www.w3schools.com/html/html_iframe.asp` to 15/11/2020

Figure 4.2: An example of a masked text with a help and a text field with a defined unit



Screenshot was taken in SForms in version 0.2.1.

Questions can be ordered or unordered, as well as both combined together. The same applies to wizard steps and answer options of closed questions. All of them can be, as already said, unordered due to each other, or ordered in a way, that one question (answer option, respectively) can have more than one preceding questions (answer options, respectively) set. In the library, the unordered ones are alphabetically sorted, and the ordered ones are organised using *tsort*[16] library. Questions can be displayed or hidden, based on conditional logic and branching. The labels, answer options, and other properties containing text can be multilingual.

4.1.3 Validation

The available validation of the form controls is the following:

- All form controls, except for media content, can be required or optional.
- A number field answer can be either validated for greater or equal than, greater than, less or equal than, less than or specified as a positive, non-positive, negative and non-negative integer.

4.2 Ontology-Based Smart Form Representation

The SForms library requires the ontology-based smart form represented in JSON-LD format. After the source code of the form is loaded to the library, the JSON-LD code is serialised into a flattened structure with missing *@context* due to mapping of properties to absolute URIs.

The flattening process transforms a graph into a list of nodes. It collects all properties of one node in a graph and transforms them into a single JSON-LD object. The same applies to blank nodes, which are labelled with blank node identifiers. This representation simplifies the structure for processing JSON-LD documents in applications.[39]

Afterwards, the shape of data is further modified to a tree structure and saved to a Reflux state manager of an external application.

The simple form consisting of three nodes, represented in JSON-LD format in a flattened structure with absolute URIs and prefixed names, can be seen in Source code 4.1.

[16] More information about tsort library available at `https://www.npmjs.com/package/tsort` to 15/11/2020

Source code 4.1: Simple ontology-based smart form in a flattened structure

```
{
  "@graph": [{
      "@id": "doc:question/form-root",
      "@type": "doc:question",
      "doc:has_related_question": {
        "@id": "doc:question/section-5686"
      },
      "form-lt:has-layout-class": "form",
      "http://www.w3.org/2000/01/rdf-schema#label": "form"
    }, {
      "@id": "doc:question/section-5686",
      "@type": "doc:question",
      "doc:has_related_question": {
        "@id": "doc:question/text-2736"
      },
      "layout:has-layout-class": "section",
      "http://www.w3.org/2000/01/rdf-schema#label": "Section"
    }, {
      "@id": "doc:question/text-2736",
      "@type": "doc:question",
      "layout:has-layout-class": "text",
      "form:requires-answer": {
        "@type": "http://www.w3.org/2001/XMLSchema#boolean",
        "@value": true
      },
      "http://www.w3.org/2000/01/rdf-schema#label": "Text"
    }
  ]
}
```

doc - Prefix label of *http://onto.fel.cvut.cz/ontologies/documentation/*.
form - Prefix label of *http://onto.fel.cvut.cz/ontologies/form/*.
form-lt - Prefix label of *http://onto.fel.cvut.cz/ontologies/form-layout/*.

4.2.1 Properties for Representation of Forms

Table 4.2 shows the most important properties of form nodes. A simple form consists of four types of form nodes: a question, an answer, an answer option, and a condition. The question type node should have at least four properties defined: *@id*, *@type*, *label*, and *has-layout-class* (from now on referred to as layout class). The layout class serves as the main distinction between form control types. All of the form controls except for a dropdown and a number field from Table 4.1 are defined using layout class property. The dropdown is used if layout class is not set to autocomplete form control, but the answer options are defined using *has-possible-value*. The number field is used if the layout class is missing or set to non-existing form control and a *has-datatype* is set to one of many variants of a numeric type. Note that, all form controls from Table 4.1 except for media content are represented as a question node. The media content is a *has-media-content* property of question nodes.

A form requires to have a root node of type question with a layout class set to *form*. Since the ontology-based smart forms have a tree structure, every question can have multiple subquestions. This is applicable to sections and wizard steps. However, any question can have subquestions which is used especially for conditional logic and branching, e.g., the subquestions of a single checkbox form control are visible only if the checkbox is checked. References to subquestion nodes are defined using *has_related_question* property. The *wizard-step* layout class represents a step of a wizard form and it can be defined only on nodes directly below a root node. For a classic form, the *wizard-step* is not used.

The form nodes of type answer are referenced from question nodes using *has_answer*

property and the condition type nodes are referenced by *is-relevant-if* property. The condition node contains at least *@id*, *@type*, *accepts-answer-value*, and *has-tested-question* properties. The *accepts-answer-value* defines the accepted answer as RDF object literal or RDF resource. The *has-tested-question* property defines a relation between conditional and trigger questions. The answer node contains at least *@id*, *@type*, and one of *has_data_value* or *has_object_value* properties. Both properties serve for the respondent answer. The *has_data_value* represents the answer by using RDF literal value and the *has_data_value* represents the answer by an RDF resource. The answer option node type serves for closed questions and consists of at least *@id* and *label* properties.

Table 4.2: Selection of SForms properties

Property	Description	URI
@id	Identification of a node.	
@type	Type of a node.	
label	Label of a question (an answer, respectively). Can be multilingual.	rdfs:label
comment	Comment of a question. Can be multilingual.	rdfs:comment
has_related_question	Subquestions defined as a list of ids of question nodes.	doc:has_related_question
has_answer	Id of an answer node.	doc:has_answer
has_data_value	Specifies value of answer.	doc:has_data_value
has-layout-class	A list of form control type and properties of a question.	form-lt:has-layout-class
has-possible-value	Answer options defined as a list of ids of answer option nodes.	form:has-possible-value
has-preceding-question	Used for ordering. Preceding questions defined as a list of ids of question nodes.	form:has-preceding-question
has-question-origin	The node identification in Linked Data.	form:has-question-origin
is-relevant-if	List of ids of condition nodes.	form:is-relevant-if
has-tested-question	Id of validated question by specific constraint.	form:has-tested-question
accepts-answer-value	Requires a specific answer to fulfil a validation condition.	form:accepts-answer-value
has-datatype	Used for setting a type of number.	form:has-datatype
description	Specifies a question help. Can be multilingual.	http://purl.org/dc/elements/1.1/description

doc - Prefix label of *http://onto.fel.cvut.cz/ontologies/documentation/*
form - Prefix label of *http://onto.fel.cvut.cz/ontologies/form/*.
form-lt - Prefix label of *http://onto.fel.cvut.cz/ontologies/form-layout/*.
rdfs - Prefix label of *http://www.w3.org/2000/01/rdf-schema#*.

4.2.2 Form Building

There is no existing form editor for designing ontology-based smart forms for the SForms library. The design process currently consists of generating nodes using custom shell scripts and afterwards editing those in text editors, or in generic RDF editors, such as TopBraid EDG or Protégé[17], and then converting to JSON-LD format.

[17] More information about Protégé available at `https://protege.stanford.edu/` to 15/11/2020

Chapter 5

Application Design

This chapter deals with the design of the Semantic Form Editor application. It discusses the requirements of the editor, as well as the required modifications of the SForms library are presented.

5.1 Analysis of the Design of the Semantic Form Editor

The Semantic Form Editor will consist only from a frontend application without a backend. The designed application will be used for building ontology-based smart forms, which can be exported in JSON-LD format. These forms can be afterwards used in applications for form distribution and data collecting, which use SForms for visualisation of forms. More information about ontology-based smart forms is available in Section 4.2.

The absence of a backend part of the application was decided, for the reason that it will be developed as a part of another master thesis. The tool will be built as a standalone web application without the necessity of a server or a database. However, it should provide a mechanism to be usable with server applications for the management of forms.

For a better understanding of this section prior to deeper analysis, it is required to state basic information about planned parts of the application and the user roles.

Based on the analysis of form builders in Section 3.2, the lifecycle of the form consists of five parts, creation of a form, design of a form, preview of a designed form, collecting data from users and a following analysis of the collected data. Therefore, the Semantic Form Editor, which serves only for designing forms, which can be afterwards used in applications for collecting data, will consist of four parts:

1. New / Import mode
2. Edit mode
3. Preview mode
4. Export mode

The system is planned to distinguish two user roles. Their functionalities will be described later in this chapter. The user roles are defined below:

- **Expert** is a user who has access to all features of the form editor. Typically, it is a knowledge engineer that builds forms from existing ontologies. He is aware of underlying Semantic Web technologies, mainly understanding format JSON-LD.
- **Non-expert** is a user who has minimal or no knowledge about the underlying Semantic Web technologies. The features containing the JSON-LD code should be hidden and only basic features should be available.

5.1.1 Software Requirements

The software requirements of the Semantic Form Editor are divided into functional and non-functional ones. For better referencing, every requirement has a unique identification number. For better clarity, the functional requirements related to one feature are grouped together under one greater requirement. In addition, each requirement (except for the greater ones) have set priority using the MoSCoW method, based on analysis from previous chapters and the requirements of the supervisor of this work. The requirements are marked by the first letter of the MoSCoW priority scale, which is described in further detail below. Next, requirements contain a design proposal or a reason why a specific requirement will not be analysed or implemented. If none of that is stated, the requirement is out of scope for the prototype version of the application. For better clarity, the requirements which will be fully implemented are marked with a star (*), the partially implemented ones are marked with a dagger (†).

MoSCoW prioritisation technique divides the requirements into four categories:[31]

- Must Have (**M**)
  - A critical requirement, there is no point in implementing the solution without this requirement, because the application would not be viable, legal or safe to use.
- Should Have (**S**)
  - An important requirement, without it, it is possible to use the application with some eventual workaround.
- Could Have (**C**)
  - A wanted or desirable requirement, it is less important than *Should Have* and has less impact on the overall solution. If a problem occurs or a deadline is at risk, those requirements are the first one to be removed from a plan.
- Won't Have this time (**W**)
  - For this requirement, it was agreed that it would not be implemented. It is better to focus on more important requirements in this version.

5.1.1.1 Functional Requirements

For an illustration of the possible usage of the functional requirements during the process of designing forms, activity and use case diagrams created using UML 2.0 showing the requirements in different scenarios, are presented in this section.

Moreover, the colours used in diagrams represents if the specific steps in scenarios will be implemented in the prototype version. The green colour expresses complete implementation, the orange colour partical implementation, and the red colour missing implementation. In addition, the dark blue colour represents the step is performed outside the form editor application. Finally, the light blue colour represents that the step will not be implemented in the prototype version but can be fulfilled by a workaround.

Design process of a form

FR1: Create a new form / import an existing source code of a form

M FR1.1: System allows the user to start with a new form*

M FR1.2: System allows the user to load an existing form from a file*

S FR1.3: System allows the user to load an existing form from an external application*
S FR1.4: System allows the user to load a form by pasting its source code to the editor*
S FR1.5: System displays a source code of a loaded form*

- This requirement is a part of the creational phase in the form lifecycle. Therefore, it will be implemented in the prototype version.

FR2: Create forms of different form types

M FR2.1: System allows the user to create a classic form*
W FR2.2: System allows the user to create a multi-step form
M FR2.3: System allows the user to create a wizard form*
W FR2.4: System allows the user to create a card form
S FR2.5: System allows the user to change a form type during design of a form*

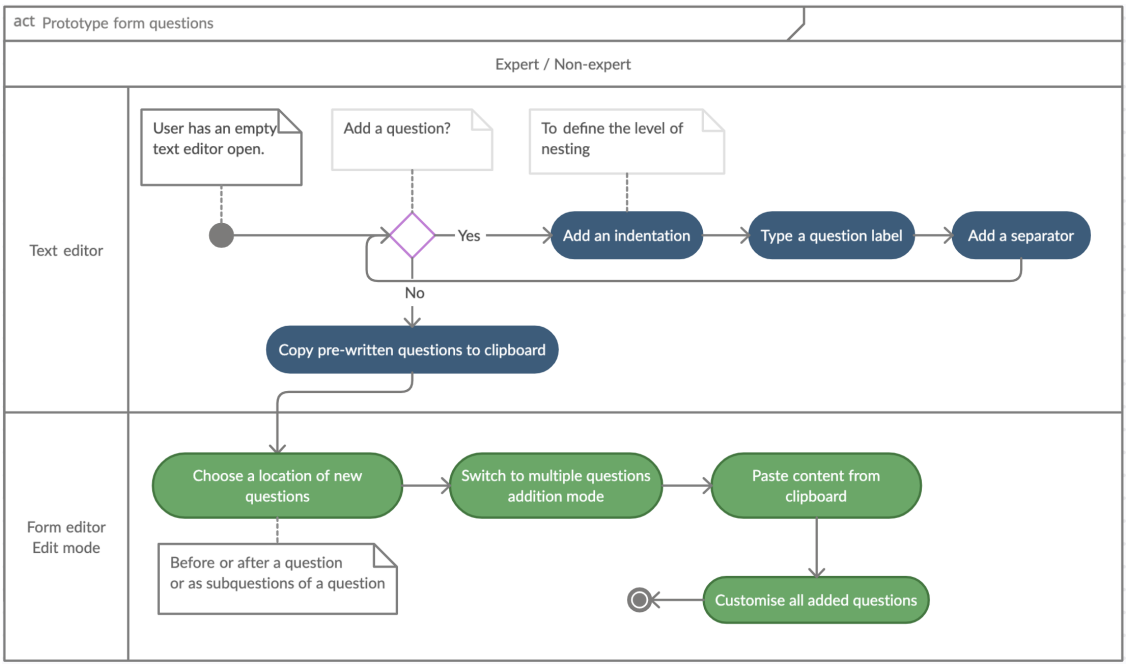
- The SForms is currently used only with a classic form and a wizard form type. Therefore, only FR2.1 and FR2.3 together with FR2.5 are usable and will be implemented in the prototype version. However, to be able to use a classic and a wizard form type easily in the SForms library, the modifications are required. More information in Section 5.2.

FR3: CRUD of questions in a form

M FR3.1: System allows the user to create a question*
M FR3.2: System allows the user to show question properties*
M FR3.3: System allows the user to update question properties*
M FR3.4: System allows the user to delete a question*
S FR3.5: System allows the user to add multiple questions at once*

- This is another essential requirement, each user has to be able to use to design forms. FR3.5 allows a user, e.g. to pre-write in advance a form structure in a text editor and copy-paste it to a form editor, an example scenario using this requirement can be seen in Figure 5.1

Figure 5.1: Scenario of adding multiple questions to a form at once



FR4: Reusability of questions

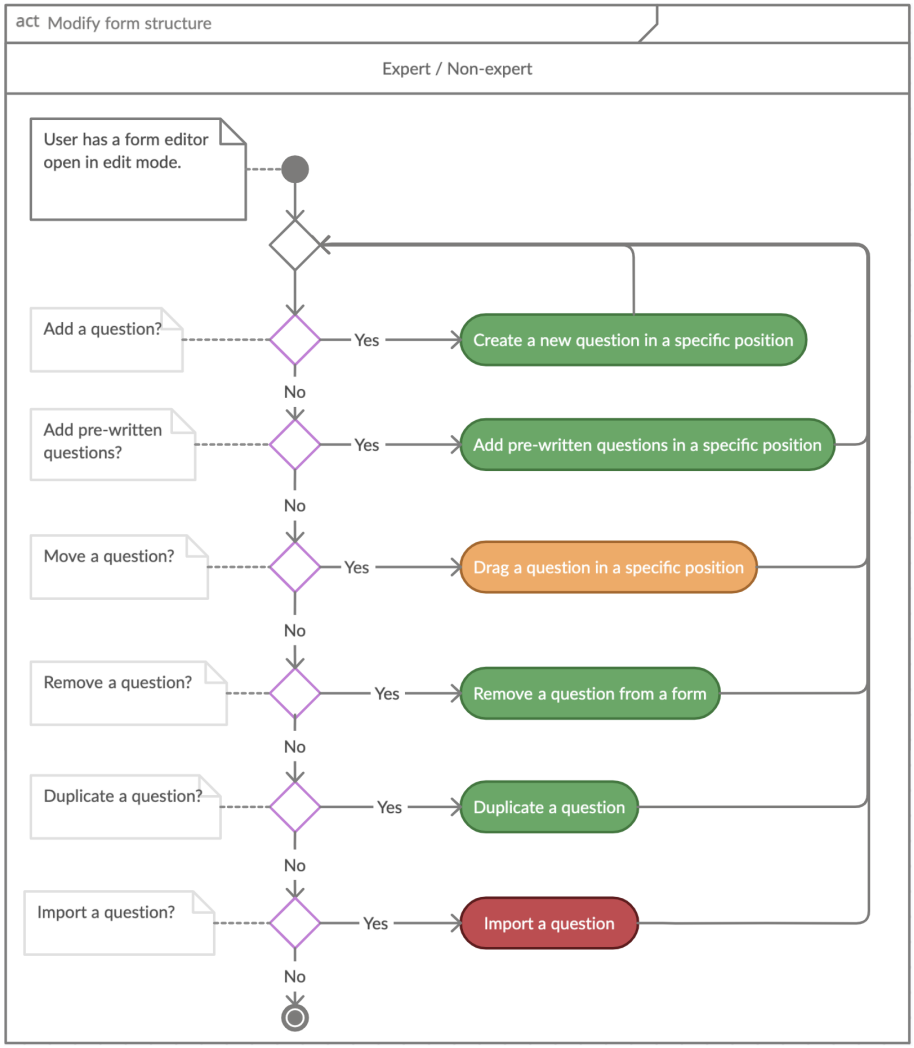
M FR4.1: System allows the user to duplicate questions with all its properties*

W FR4.2: System allows the user to import existing questions from other forms

W FR4.3: System allows the user to import questions from a question bank

- FR4.1 will be implemented and should allow a user to duplicate all questions with all its properties and answer options. On the other hand, FR4.2 and FR4.3 are not planned in the prototype version, due to the absence of a back-end side of the application. Together with other functionalities, the activity diagram using FR4.1 and FR4.2 in modification process of a form can be seen in Figure 5.2.

Figure 5.2: Modification process of a form structure in a form editor



FR5: Order of questions

S FR5.1: System allows the user to a create an ordered (unordered, respectively) question at a specific place of a form†

S FR5.2: System allows the user to drag a question to a specific place in an order (unorder, respectively) manner using Drag and Drop API†

C FR5.3: System allows the user to drag multiple questions at once to a specific place in an order (unorder, respectively) manner using Drag and Drop API

S FR5.4: System allows the user to unorder a question with a single click*

- This requirement should use the *has-preceding-question* property defined in Section 4.2.1 to set the order of questions within a form. FR5.1 and FR5.2 will be implemented only partially. It is possible to use the *has-preceding-question* property to define more than one preceding question per question. However, for the prototype version, only a variant with one preceding question defined per question will be implemented due to time constraints. Form designers can use a workaround of FR6. FR5.3 will not be implemented in the prototype version. On the other hand, FR5.4 will be implemented completely.

FR6: Edit a source code of a form during a design process

M FR6.1: System allows the user to modify a source code of a form*

W FR6.2: System allows the user to choose a formatting style of a source code of a form

- FR6.1 allows for editing the properties of a form without having to use the user interface of the editor where some requirements concerning the editing of a form may not have been implemented. FR6.2 will not be implemented in the prototype version due to time constraints.

FR7: SForms form controls

M FR7.1: System allows the user to create all types of form controls from SForms†

- SForms form controls can be seen in Table 4.1. All form controls except for dropdown, number field, SPARQL field and TURTLE field will be supported in the prototype version. The reason for not supporting a dropdown and a number field is because those form controls cannot be set by one layout class property as the other do. More information is available in Section 4.2.1. A SPARQL field and a TURTLE field will not be supported in the prototype version, due to the problems with compatibility of libraries used. More information can be found in Section 5.2.

FR8: CRUD of answer options of closed questions

M FR8.1: System allows the user to add new answer options to a question*

M FR8.2: System allows the user to remove answer options from a question*

M FR8.3: System allows the user to update answer options of a question*

M FR8.4: System allows the user to show answer options of a question*

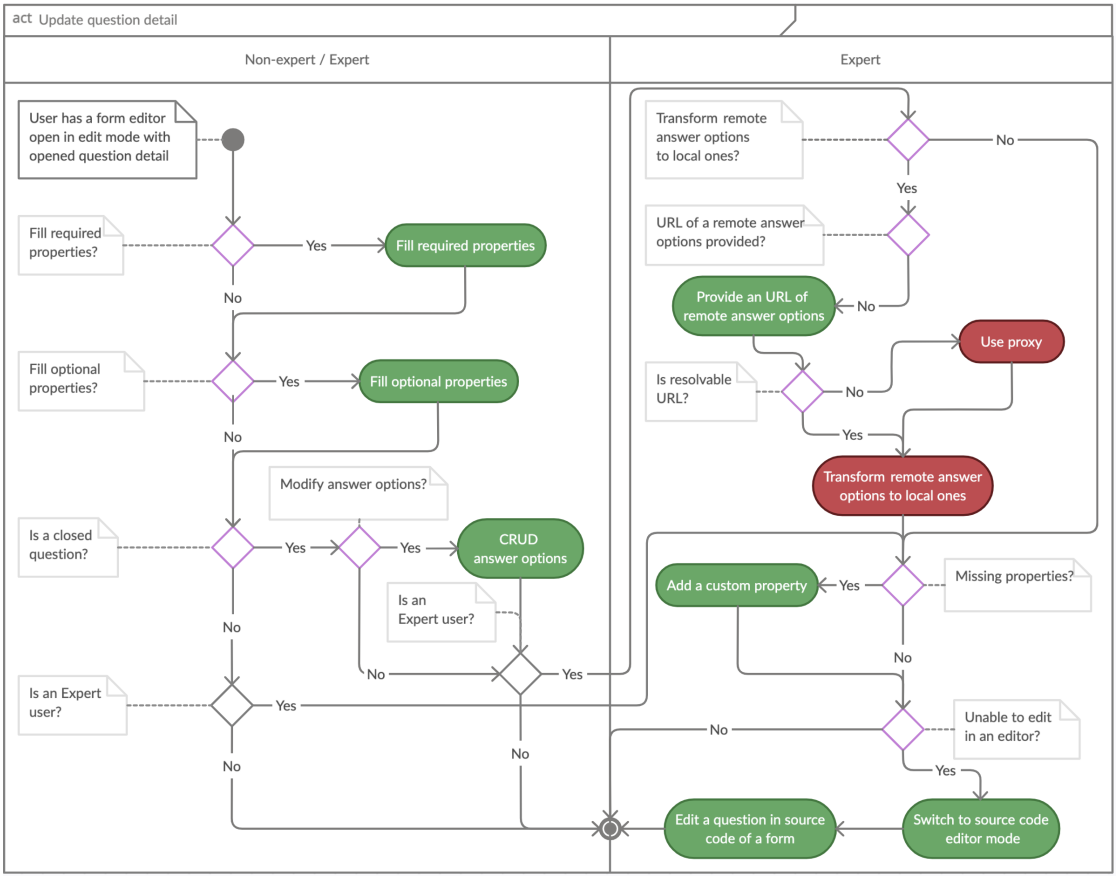
C FR8.5: System allows the user to add *Other answer option*

C FR8.6: System allows the user to transform remote answer options to local ones

C FR8.7: System allows the user to set proxy to resolve URL of remote answer options

- Closed questions and their answer options are an essential part of a form. Therefore, it is necessary to implement FR8.1 to FR8.4. FR8.5 implementation is dependent on modifications of the SForms library. For this reason, it will not be implemented in the prototype version of the editor. FR8.6 and FR8.7 will not be implemented in the prototype version due to time constraints. However, the activity diagram in Figure 5.3 displays the processes were these two requirements are involved.

Figure 5.3: Process of updating a question detail



FR9: Order of answer options of closed questions

C FR9.1: System allows the user to a create an ordered (unordered, respectively) answer option at a specific place

C FR9.2: System allows the user to drag an answer to a specific place in an order (unorder, respectively) manner using Drag and Drop API

C FR9.3: System allows the user to drag multiple answers at once to a specific place in an order (unorder, respectively) manner using Drag and Drop API

W FR9.4: System allows the user to unorder answer option with a single click

W FR9.5: System allows the user to display a preceding answer option of a specific answer option

- Due to time constraints, this requirement will not be implemented in the prototype version.

FR10: Reusability of answer options

W FR10.1: System allows the user to use answer options from other questions

W FR10.2: System allows the user to import existing answer options from other forms

W FR10.3: System allows the user to choose one of a predefined set of answer options

- Due to time constraints and a lack of a back-end, this requirement will not be implemented in the prototype version.

FR11: Multilingual forms

M FR11.1: System allows the user to create a form localised in any number of languages*

M FR11.2: System allows the user to create non-localised forms*

M FR11.3: System allows the user to add and remove localisation during the process of building a form*

M FR11.4: System allows to localise all localisable properties into configured languages†

- Since forms can be used by different nations, it should be possible to create forms in more languages. The SForms library is ready for multilingual forms, and therefore, the requirement will be implemented. FR11.4 will be implemented for a label, a help and answer options in the prototype version.

FR12: Preview a form in the SForms library

M FR12.1: System allows the user to preview a modified form*

M FR12.2: System allows the user to navigate to a specific question in preview mode from edit mode*

M FR12.3: System allows the user to preview a modified form in all configured languages*

M FR12.4: System allows the user to preview a wizard form in horizontal and vertical orientation*

M FR12.5: System allows the user to CRUD answers of questions in a form*

C FR12.6: System allows the user to navigate from preview mode of a specific question to edit mode of the question

- This requirement is a part of the preview phase in the form lifecycle. It is a phase in which a form designer can review a designed form. To implement this requirement, the SForms library will be used. However, the modifications of SForms listed in Section 5.2 are required to achieve it. FR12.6 will not be implemented due to other necessary modifications in SForms and time constraints.

FR13: Distribution of a form

M FR13.1: System allows the user to download a source code of a form*

M FR13.2: System allows the user to copy a source code of a form to a clipboard*

S FR13.3: System allows the user to publish a form to an external application*

S FR13.4: System optionally publishes a form on each change to an external application*

- This requirement participates in realising the data collection phase of a form lifecycle which serves for exporting of a source code of a form. FR13.3 and FR13.4 should be activated in a form editor together with the FR1.3. If the form is loaded from an external application, it should be published back to the same application after the process of modification of a form ends. The distribution of a form is captured in Figure 5.4. The possible usage of FR13.3 and FR13.4 is captured in scenario in Figure 5.5.

Figure 5.4: Scenario capturing distribution of a form

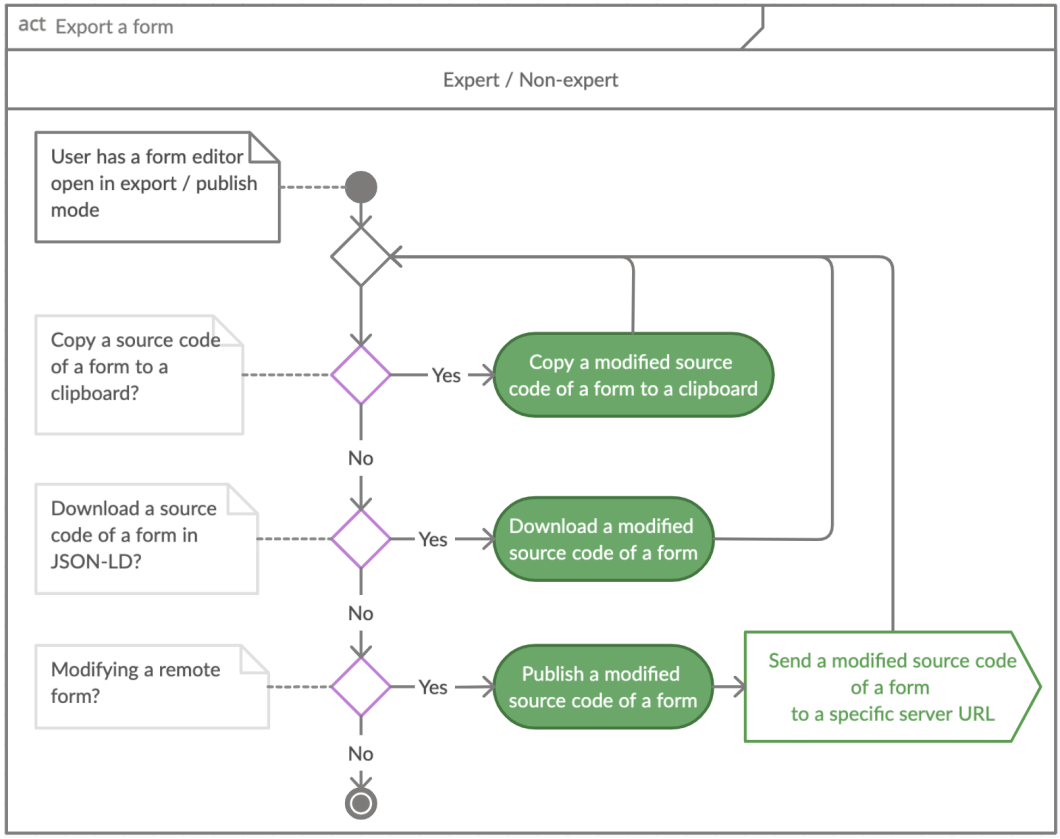
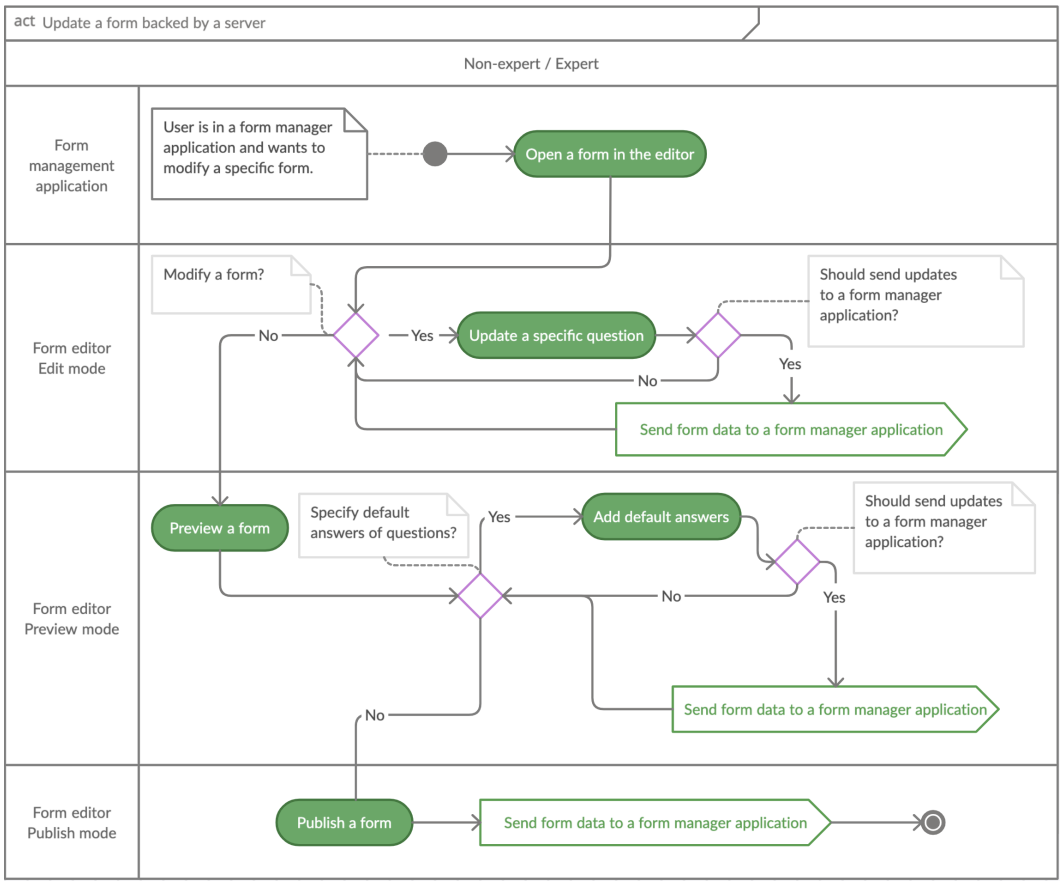


Figure 5.5: Scenario capturing a form editor with a form loaded from an external source



FR14: Conditional logic and branching

C FR14.1: System allows the user to set conditional logic and branching of questions

FR15: Welcome and thanks page

W FR15.1: System allows the user to add and customise the welcome page

W FR15.2: System allows the user to add and customise the thanks page

- This requirement needs specific modifications of the SForms library to be able to have a page before a form and after a form, as well as a support for at least a heading and paragraph form controls. The requirement will not be implemented in the prototype version.

FR16: Randomisation of questions and answer options

W FR16.1: System allows the user to set randomisation of order of questions

W FR16.2: System allows the user to set randomisation of order of answer options

- This requirement needs modifications of the SForms library to be able to configure randomisation of questions and answer options. Therefore the requirement will not be implemented in the prototype version.

FR17: Customisation of form design

W FR17.1: System allows the user to customise question design

W FR17.2: System allows the user to customise answer options design

W FR17.3: System allows the user to customise form design

- This requirement needs modifications of the SForms library to be able to display styles of the form controls, answer options or the form itself. Therefore the requirement will not be implemented in the prototype version.

FR18: Form according to best-practices

M FR18.1: System allows the user to create a form according to best-practices

- This requirement needs modifications of the SForms library to satisfy the list of the best-practices of form design from Section A.2. Those changes will not be implemented in the prototype version.

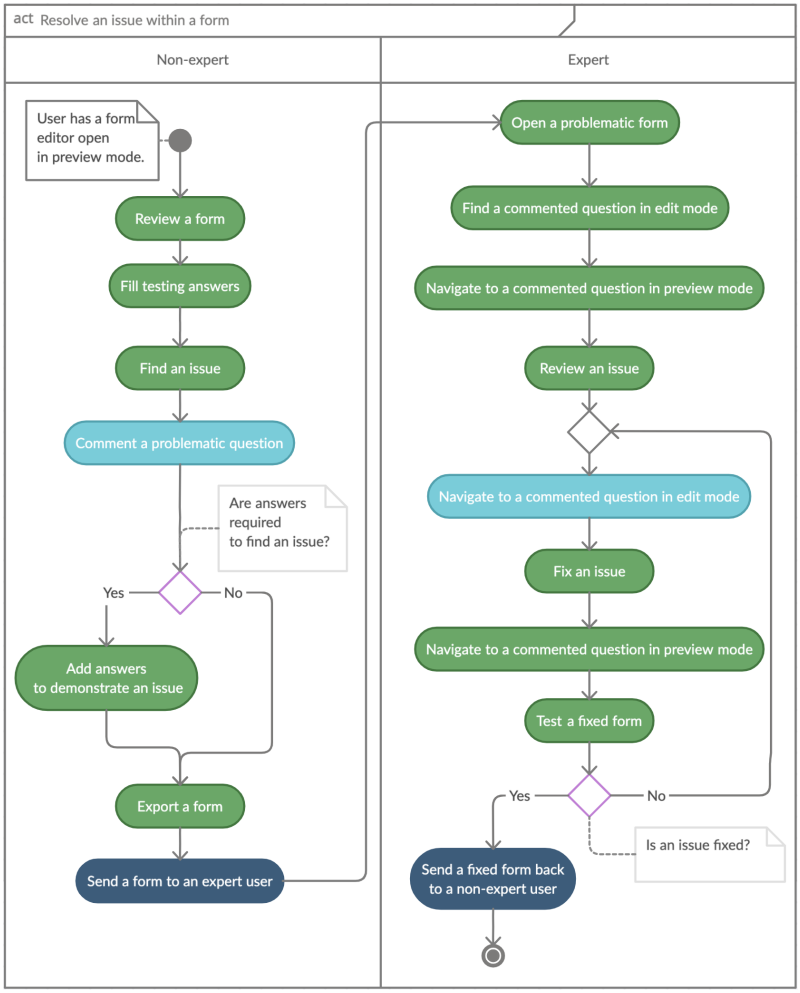
FR19: Comment on problematic questions

S FR19.1: System allows the user to comment on questions in preview mode

S FR19.2: System indicates the problematic questions*

- FR19.1 requires modifications of the SForms library and will not be implemented. It will be possible to comment questions in the edit mode using FR3.3. FR19.2 will be implemented using FR20.6. This requirement can benefit from FR12.5, because some issues of the form can be hidden until specific answers are provided. An example scenario applying this requirement can be seen in Figure 5.6.

Figure 5.6: Scenario of possible usage of the commenting questions requirement



Properties of a form editor

FR20: Question properties overview without an interaction with a question

S FR20.1: System indicates a form control type of a question*

S FR20.2: System indicates if an answer of a question is required*

S FR20.3: System indicates if a question has a preceding question and shows the preceding question on demand*

S FR20.4: System indicates if a section form control is by collapsed by default*

S FR20.5: System indicates if a question has a help*

S FR20.6: System indicates if a question has a comment and displays it on demand*

S FR20.7: System indicates if a question is disabled*

S FR20.8: System indicates if a question is hidden*

- This requirement speeds up the process of building a form. Therefore, it will be implemented in the prototype version.

FR21: Search questions

W FR21.1: System allows the user to find a question with a specific label

W FR21.2: System allows the user to filter questions by properties

- This requirement is unnecessary in the prototype version. However, as a workaround of FR21.1, a form designer can use the native find functionality of browsers.

FR22: Collapsible and expandable sections

C FR22.1: System allows the user to collapse or expand section questions*

C FR22.2: System allows the user to collapse or expand all section questions at once*

- This requirement speeds up the process of building forms by increasing clarity in the form editor. It will be implemented in the prototype version.

FR23: Undo changes

W FR23.1: System allows the user to undo and redo changes made

FR24: Real time collaboration

W FR24.1: System allows users to collaborate on the design of the form

FR25: Visibility of system status

C FR25.1: System informs the user about pending, successful and unsuccessful actions*

C FR25.2: System highlights questions which are being modified or moved*

- According to ten usability heuristics for user interface design by Jakob Nielsen, applications should always keep users informed about what is happening.[36] This requirement will be taken into account during the implementation of the application.

FR26: Validate a form

C FR26.1: System allows the user to validate a form according to ontology-based rules*

M FR26.2: System allows the user to validate a form according to best-practices*

M FR26.3: System allows the user to preview the validation result*

M FR26.4: System indicates invalid questions*

S FR26.5: System allows the user to CRUD validation constraints within a user interface

C FR26.6: System allows the user to validate a form by a specific validation constraint

C FR26.7: System allows the user to ignore a specific validation constraint for all questions

C FR26.8: System allows the user to ignore a specific validation constraint for a specific question

W FR26.9: System provides the user a manual on how to correctly design forms

- This is one of the requirements of this master thesis. FR26.1 and FR26.2 should be implemented using the SHACL language. The validation constraints planned for a validation of forms according to best-practices can be found in Section A.1. For the reason that it is a requirement of the master thesis, FR26.1 and FR26.2 will be implemented, as well as FR26.3 and FR26.4. FR26.5 will not be implemented in the prototype version. However, it should be possible to modify validation constraints in a source code of an application and afterwards deploy a new version of a tool with modified validation constraints. FR26.6 to FR26.9 will not be implemented in the prototype version. A possible way of using validation of a form can be seen in Figure 5.7 and Figure 5.8.

Figure 5.7: Scenario of possible usage of a form validation feature

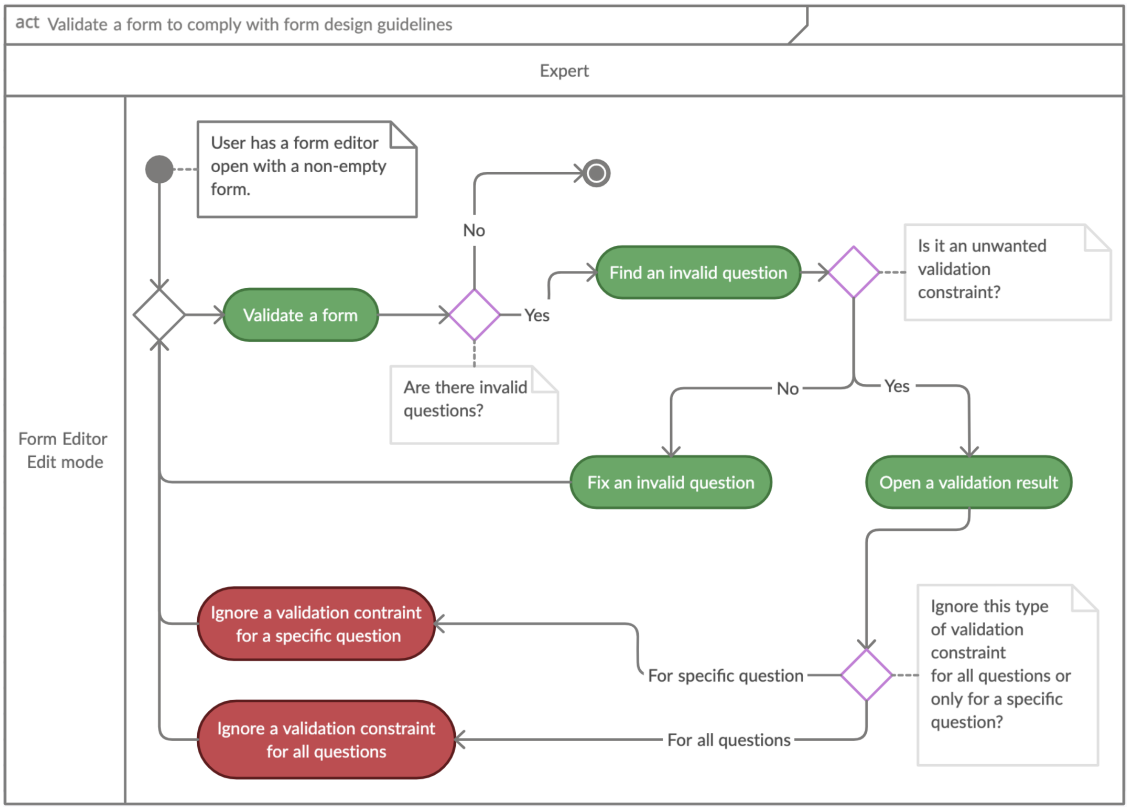
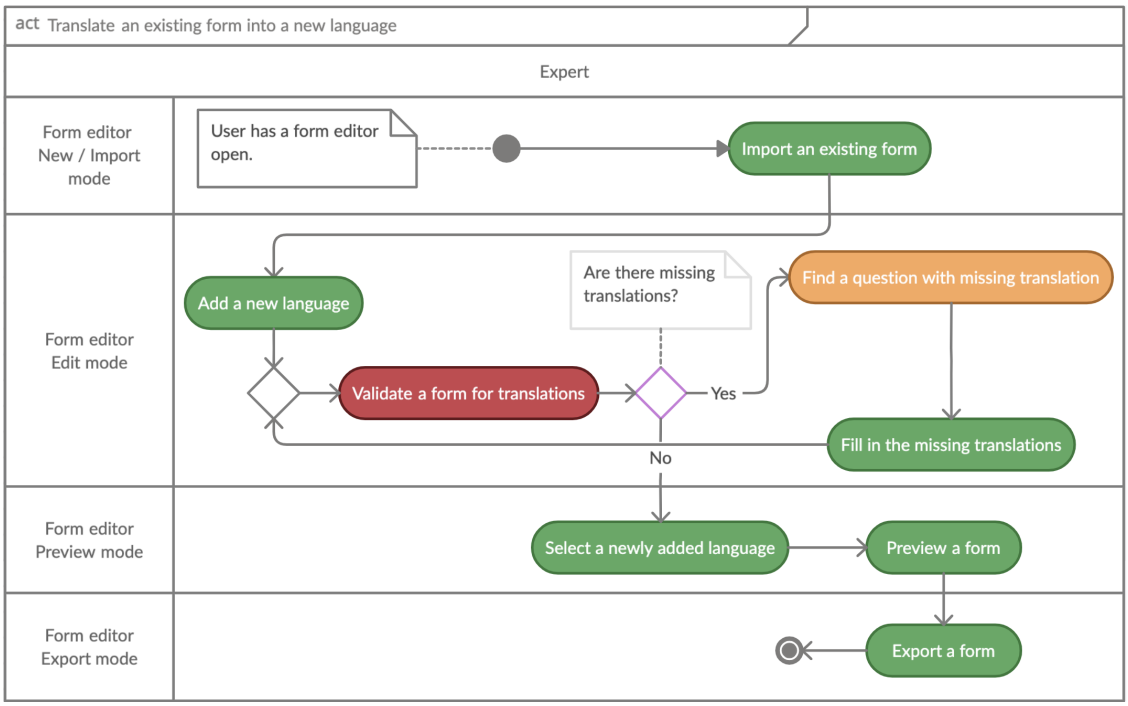


Figure 5.8: Process of translating a form into a new language



FR27: User roles

W FR27.1: System allows the user to choose a user role between Expert and Non-expert

- The user roles will not be implemented in the prototype version. However, the planned functionalities of each user role can be seen in the use case diagram in Figure 5.9.

Figure 5.9: Overview of some of the functionalities of users and the system in the editor

Others

FR28: Detect unsupported browser

C FR28.1: System warns the user about a usage of an unsupported browser

FR29: Log application errors

S FR29.1: System logs errors occurred during a process of building a form*

- This functionality will be implemented to help detect potential errors in the application.

5.1.1.2 Non-Functional Requirements

All of the non-functional requirements listed in this section will be taken into account during the implementation and testing of this work.

Portability and compatibility

M NFR1: System will be working in the newest versions of browsers*

- The application will be working in the last three versions of Google Chrome, Firefox and Safari to 22/11/2020.

M NFR2: System will be working on devices with display width >= 1000 pixels*

Localisation

M NFR3: System will be in English*

Accessibility

M NFR4: System allows creating of questions using keyboard*

Maintainability

M NFR5: System will be tested by unit tests*

- At least 75% of form logic functionalities should be covered by unit tests*

M NFR6: System will be tested by end-to-end tests*

- At least 75% of functionalities should be covered by end-to-end tests*

M NFR7: System will be tested by user testing*

- System should be tested on at least 3 users.

M NFR8: System will be created in newest versions of mainstream web technologies*

M NFR9: System will be extensible and maintainable*

5.2 Analysis of Required Modifications of SForms

This section discusses the modifications of the SForms library, which are required to be done to be able to fulfil the software requirements of the Semantic Form Editor.

In the next lines, the requirements of the Semantic Form Editor dependent on the modifications of SForms are listed. They are separated into two categories according to the necessity of implementing for the prototype version of the Semantic Form Editor.

Will be implemented in the prototype version

FR2.1, FR2.3: Require SForms to be able to handle the classic and the wizard form type

- Based on the analysis from Chapter 4, the SForms library supports only the classic form type, and the wrapper of a wizard form type is implemented in the Study Manager application. To satisfy FR2.1 and FR2.3, there are two options that can be made. The first option requires to duplicate the code of the wrapper from the Study Manager into the Semantic Form Editor. Also, it needs to be duplicated to every other application which will use the SForms library. The other option is to move a wizard form type functionality to the SForms library itself. The second option mentioned was decided to choose.

FR12.1, FR12.2: Require to provide an option to preview the designed form and to allow the form designer to navigate from a specific question in edit mode to the specific question in preview mode

- According to FR12, it is required to use the SForms library in the Semantic Form Editor for previewing the created and modified forms. However, according to NFR8, the Semantic Form Editor should be developed using the newest versions of the mainstream web technologies. This leads to a problem because based on the analysis from Chapter 4, the library is developed in React 15 with React Bootstrap in version 0.30.5, but the React Bootstrap library in this version is not compatible with the newest version of React. This incompatibility requires the SForms library to be updated to work with the newest version of React Bootstrap. The required modifications are demonstrated in more detail in Section 5.2.1.

Will not be implemented in the prototype version

FR7: Requires to support all SForms form controls

- It is required to support a dropdown and a number field as a part of the layout class property. In addition, it is required to find a solution for SPARQL and TURTLE fields, which use library *yasgui-yasqe*, which is unsupported in new version of React, as discussed in Section 5.2.1.

FR8.5: Requires *Other answer option* in semi-closed questions

FR15: Requires to provide a form with welcome and thanks pages

FR16: Requires a randomisation of questions and answer options

FR17: Requires a customisable design of a form and its questions and answer options

FR18: Requires SForms to comply with best-practice guidelines listed in Section A.2

FR19.1: Requires to comment questions in the preview mode using SForms

5.2.1 Update of Technology Stack

To be able to use the SForms library in the Semantic Web Editor, it is required to:

- Update React Bootstrap to a version supporting React 16, which is the version 1.0.0 and newer. React Bootstrap in version 0.30.5 uses Bootstrap 3, but the version 1.0.0 of the React Bootstrap library requires Bootstrap 4. There are many breaking changes in Bootstrap 4 which have to be taken into account and changed. In addition, it was decided to replace the *react-boostrap-datetimepicker* and the *react-bootstrap-typeahead* libraries also dependent on Bootstrap 3 by *react-datepicker* and *react-select*, which support React 16.

- Update React to version 16 or newer. This update requires to rewrite few concepts of the source code to avoid warnings, more concretely *componentWillMount* and *componentWillReceiveProps*. In addition, it is required to update the Babel library from version 6 to 7 and also update the code bundler Browserify. However, in the case of Browserify, it was decided to replace it in favour of Webpack code bundler. Next, it was agreed to replace the Jasmine test runner by Jest[1]. Both of the replacements are planned to be made to accomplish better compatibility and maintainability.

- Migrate the remaining parts of ES5 code to ES6 to achieve easier updating and navigating in the source code of the library.

- Move the wrapper serving for a wizard form type from the Study Manager to SForms. This includes moving a state manager to the SForms library and replacing

[1] More information about Jest test runner library available at `https://jestjs.io/` to 20/11/2020

the Reflux state manager by the Context API, which is a build-in state management system in React since version 16.3.[15]

Those modifications will result in a stand-alone SForms library, able to be imported to any React project. Developers will be able to start using it by providing a source code of an ontology-based smart form in JSON-LD format and an optional configuration.

Chapter 6

Implementation

This chapter is divided into two parts. The first part describes the modifications made to the SForms library. The second part discusses the implementation of the Semantic Form Editor.

6.1 The SForms Library

This section describes the process of how the SForms library was updated, as well as how SForms can be imported and used in React-based applications. The update of SForms preceded the implementation of the Semantic Form Editor due to incompatibilities of versions of used technologies.

6.1.1 Updating Process of SForms

The changes that had to be made to the SForms library were summarized during the design process of the application in Section 5.2.1. The updating process of SForms was performed to be able to fulfil FR2.1, FR2.3 and FR12.1 of the Semantic Form Editor.

The process started by refactoring code that was outdated or was not in compliance with the modern standards such as converting of *var* declarations of variables to *let* and *const*. The process continued by preparations for updating React to version 16. One of the necessary steps was adding *prop-types*[1] library to the project, as it is no longer part of React 16. Another important change, was to transform all outdated React lifecycle functions *componentWillMount* and *componentWillReceiveProps* into *componentDidMount* and *componentWillUpdate*.

After that, React and React Bootstrap was updated to new versions. React was updated to version 16.9.0 and React Bootstrap was updated to 1.0.1. As mentioned earlier in Section 5.2.1, React Bootstrap 1.0.1 uses Bootstrap 4, but React Bootstrap 0.30.10 uses Bootstrap 3. These two versions are not fully compatible, and therefore the migration guides available on React Bootstrap website[2] and Bootstrap website[3] were followed and the code appropriately updated.

After updating the React framework and resolving the updating of React Bootstrap, some other libraries such as Babel were updated. Also, bundling library Browserify was replaced by Webpack. When Webpack was added to the project, it was configured to run

[1] More information about prop-types available at `https://www.npmjs.com/package/prop-types` to 26/11/2020

[2] Migration guide of React Bootstrap available at `https://react-bootstrap.github.io/migrating/` to 26/11/2020

[3] Migration guide of Bootstrap available at `https://getbootstrap.com/docs/4.3/migration/` to 26/11/2020

the SForms library as a stand-alone application in the browser for development purposes. Next, the Jasmine test runner library was replaced by Jest library and tests were updated to be able to run using Jest.

As the most advanced step, the wrapper for a wizard form type was moved from the Study Manager application to SForms due to reasons discussed in Section 5.2. This step also contained removing Reflux state manager dependency and migrating to the Context API, the React build-in state manager. A context was created for storing a form structure, described in Section 4.2, and remote answer options of the autocomplete form control.

However, the process of updating SForms did not end with the last-mentioned step but instead continued during the implementation of the Semantic Form Editor, when bugs were encountered, or small modifications were required.

One of the required modifications was to be able to realise FR12.2 which in short allows a user to select a specific question in edit mode and navigate to it in preview mode. This adjustment required adding a configuration property serving for informing the SForms system to start with a question with a specific *@id* property. If the configuration property is provided, SForms finds the question with specific *@id* in the form structure and determines the initial wizard step in case of a wizard form type. Then, it renders a form, finds a question with specific *@id* in the DOM[4] of a page, scrolls to it and highlights it.

6.1.2 Example Usage of SForms in Application

Shortened example of usage of the SForms library in the Semantic Form Editor can be seen in Source code 6.1. The *SForms* component is imported from the library and requires a source code of a form as a *form* property, configuration object as an *options* property and *ref*, which allows accessing the *getFormQuestionsData* function which serve for accessing of SForms form structure data with filled answers by a user.

Source code 6.1: Shortened example of usage of the SForms library in the Semantic Form Editor

```
1  import { useRef } from 'react';
2  import SForms from 's-forms';
3  import useExportedForm from '../hooks/useExportedForm/useExportedForm';
4
5  const EditorPreview = () => {
6    const sformsContainer= useRef(null);
7    const form = useExportedForm();
8    const options = {
9        modalView: false,
10       horizontalWizardNav: true,
11       wizardStepButtons: false,
12       enableForwardSkip: true
13   };
14   return (
15     <SForms
16       ref={sformsContainer}
17       form={form}
18       options={options}
19     />
20   );
21 }
```

[4]Document Object Model. More information available at https://www.w3schools.com/whatis/whatis_htmldom.asp to 26/11/2020

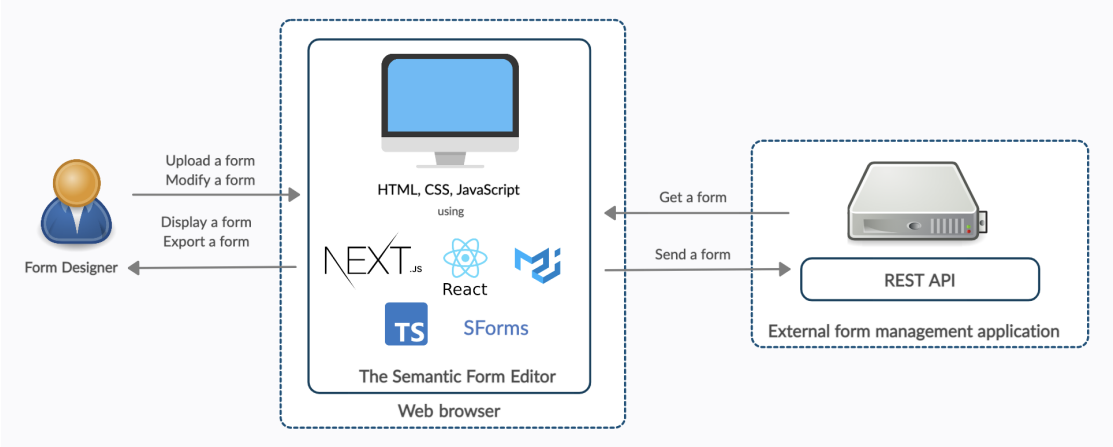
6.2 The Semantic Form Editor

This section is dedicated to the implementation of the application prototype. Firstly, the technology stack for the implementation is described. Furthermore, this section discusses the structure of the application and the method of representation of the data in the application, as well as how the individual functional and non-functional requirements have been implemented.

6.2.1 Technology Stack and Architecture

The application is developed in JavaScript. The used application architecture is displayed in Figure 6.1.

Figure 6.1: The application architecture



The core frameworks and libraries used are:

- **React** is a component-based JavaScript framework for building user interfaces. Using React, it is possible to develop a single page frontend application, which allows changing parts of the website without reloading of the whole page.[15]
- **TypeScript** is a language which extends JavaScript by adding static typing. Using TypeScript, the development of the application is faster, thanks to catching the bugs and errors before the code is compiled.[33]
- **Next.js** is a frontend framework which makes the development of React applications easier. The advantage of this framework for this project is the zero-configuration of compilation and bundling of the code, as well as the out of box support of TypeScript. Another benefit is the automatic deployment and hosting of the application.[45] The deployed application will be used for the demonstration of the progress of the implementation to the supervisor and for user testing.
- **Material-UI** is a user interface framework providing a large amount of prepared React components, which can be easily imported into the project and make the development of an application faster.[27]
- **Cypress** is a frontend testing tool, which can be used for testing any application running in a browser. It allows writing end-to-end, integration and unit tests in JavaScript. The running tests can be seen and in case of error debugged in a browser or analysed in the Cypress management system which provides results and recordings of tests. The service provides free of charge pricing plan, which is sufficient for this project. [7]

Another noticeably used library in the Semantic Form Editor is the SForms library. Some of the other libraries used are, for example, libraries for the validation of forms using SHACL language. The other libraries chosen are mentioned in Section 6.2.4 about the implementation of specific requirements.

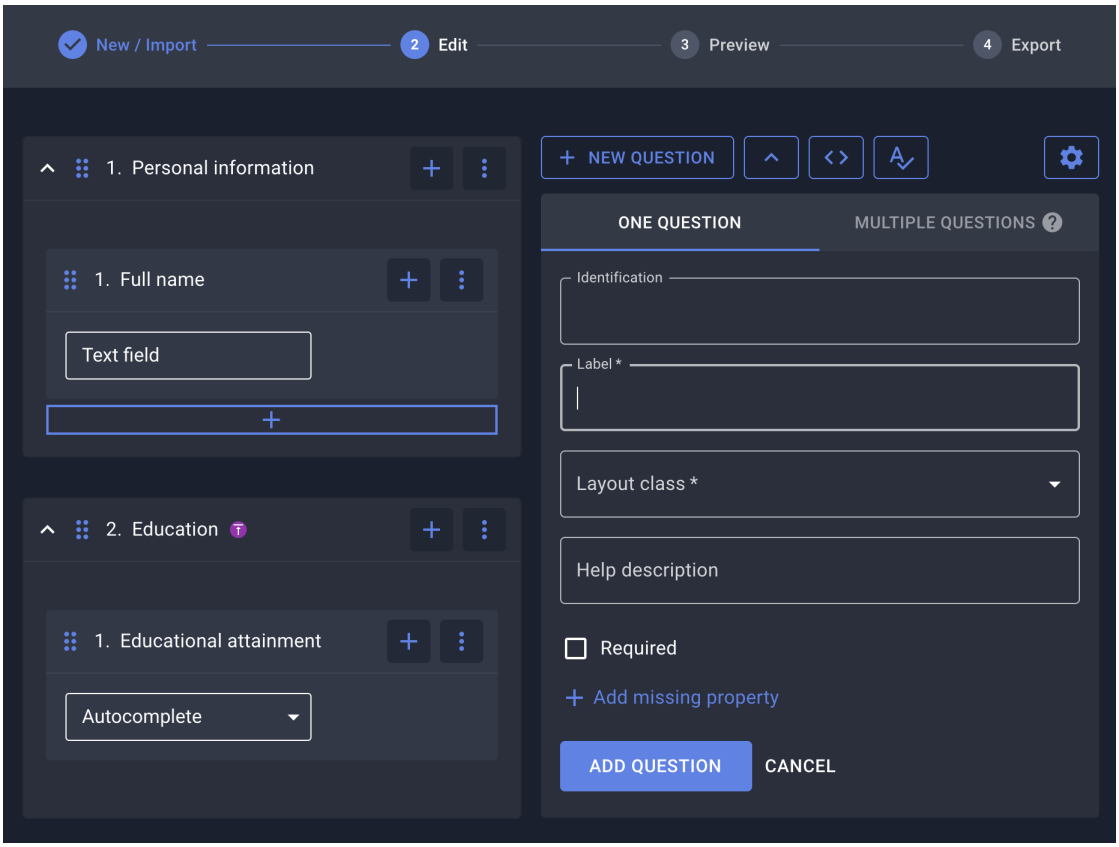
6.2.2 Layout

As planned in Chapter 5, an application is divided into four modes that represent the phases in the process of designing a form:

- **New / Import mode**, where a user can create a new or import an existing form.
- **Edit mode**, where a user can edit a form in a user interface or in a code editor.
- **Preview mode**, where a user can preview the form just created or modified.
- **Export mode**, where a user can export the form just created or modified.

The edit mode in a user interface can be seen in Figure 6.2. There is a navigation bar, in the top of the application, which displays the four modes and also serves for navigating between them. A form designer cannot move from New / Import mode to the next mode without creating or importing a form. In terms of navigating between the other nodes, a form designer can navigate as soon as at least one question is created.

Figure 6.2: The Semantic Form Editor in edit mode with four questions and the form bar opened



Next, in edit mode, situated on the left side, there are rendered questions of the form in a tree structure format having the same order as in SForms. Each question consists of a body and a header with label, indicators, and actions. On the right side, if a question is chosen or a form designer wants to add a new question, a form bar for filling the

question properties is shown. It serves for modifications as well as for the displaying of the question properties. Above the form bar, an action bar is located.

6.2.3 Structure of Data

Right after a form source code is loaded to the application, the source code is transformed into the same format as in SForms, mentioned in Section 4.2. The form is serialised into a flattened structure with a mapping of properties using *@context* to absolute URIs. The form in flattened structure can be seen in Source code 4.1. Afterwards, the shape of data is further modified to a tree structure using the *expandStructure* algorithm from the SForms library.

The *@context* part of a form source code is stored for later use, to be able to reverse the process of flattening, which is required for exporting the form back to the initial format.

The transformed source code of a form is then stored in an object of a FormStructure type. The FormStructure object has a pointer to a root node object, which consists of a question of the form layout class, and a map with all nodes identified by an *@id* of a question for constant access. Example of the FormStructure class can be seen in Source code 6.2. The node is of type FormStructureNode, which stores a pointer to its parent node and a data representing properties of a specific question from a transformed source code of a form.

Source code 6.2: An example of class representing a form in the Semantic Form Editor

```
class FormStructure {
  public root: FormStructureNode;
  public structure: Map<string, FormStructureNode>;

  constructor(root: FormStructureNode) {
    this.root = root;
    this.structure = new Map<string, FormStructureNode>();
    this.structure.set(root.data['@id'], root);
  }

  addNode(node: FormStructureNode) {
    this.structure.set(node.data['@id'], node);
  }
...
}
```

The data in the application are stored using React build-in state manager, the Context API. If the data are updated, the components using the specific context data, are re-rendered. However, to make it re-render, the updated data has to be an entirely new object and not the old one with only updated properties. For that, a *cloneDeep* function, which recursively clones the object, from *lodash*[5] library is used.

6.2.4 Implementation of Functional Requirements

This section presents which functional requirements were implemented and what means were used for implementing those requirements. The summary of all functional requirements can be found in Section 5.1.1.1.

Each requirement begins with the list of implemented, partially implemented or not implemented subrequirements of the requirement.

[5]More information about Lodash available at `https://lodash.com/` to 25/11/2020

FR1: Create a new form / import an existing source code of a form

- Implemented: FR1.1, FR1.2, FR1.3, FR1.4, FR1.5

The first screen of the application, create / import mode consists of buttons and an editor of a code. If a form designer uses *New Form* button, the system loads an empty form template of ontology-based smart form. The source code of the empty form is located in the separate file in the Semantic Form Editor project and can be replaced prior to compilation of the application. The template consists of a *@context* part which contains the basic properties and a *@graph* part with the root form node.

The button *Import existing form* allows a user to load a JSON-LD file with a source code of a form from his device. This is implemented using an input form element of type *file* with the restriction to allow only *application/json* and *application/ld+json* formats.

Another part is an editor of a code in JSON-LD format, which displays the created or imported form source code. A user can use it for editing the created or imported form, or he can paste there a source code of a form from the clipboard. The editor is implemented using *jsoneditor*[6] library, which allows viewing, editing, as well as formatting and validating the code in JSON (JSON-LD, respectively) format. The last button *Save and start editing* navigates the user to edit mode.

The form can also be loaded from an external source. If the pathname of the URL of the application contains a query parameter *formUrl*, the application tries to load the form using the GET HTTP request method[7]. It is also possible to use it in combination with WebDAV[8] server, which can be used to edit source code of forms directly from the file system. In addition, there is optional query parameter *draftUpdate*, which is introduced in the implementation of FR13. An example pathname with query parameters can be seen in Source code 6.3.

Source code 6.3: A pathname with query parameters for loading a form from an external source

```
1 /?formUrl=http://example.com/newForm.json&draftUpdate=true
```

FR2: Create forms of different form types

- Implemented: FR2.1, FR2.3, FR2.5
- Not implemented: FR2.2, FR2.4

The application supports only classic and wizard form type because other form types are not supported by the SForms library.

For the classic form type, edit mode allows a user to create any question (except for a wizard step question) on any tree level of a form. In contrast to the wizard form type, the user can create only a wizard step type question on the root level and create the others as their subquestions.

A user can choose the form type having navigated to edit mode with an empty form. Alternatively the user chooses one during the design process by opening the configuration of the form from the action bar. If the user changes the type of a form during the design process from the wizard form to the classic form, the wizard steps are changed to sections. The opposite is from the classic form to the wizard form. This conversion depends on the layout class of questions at the root level. If questions at the root level

[6] jsoneditor library available at https://www.npmjs.com/package/jsoneditor to 26/11/2020

[7] More information about HTTP request methods available at https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods to 26/11/2020

[8] More information about WebDAV available at https://www.comparitech.com/net-admin/webdav/ to 26/11/2020

are all of the type section, then they are all transformed into a wizard step type. If not, all existing questions become a subquestion of a new wizard step question.

The form type is not stored in the source code of a form but dynamically decided based on the questions on the root level. If all of the subquestions are wizard steps, then the form type is a wizard form, if none of them are wizard steps, then the form type is a classic form. If the question types are mixed, the behaviour is not defined.

FR3: CRUD of questions in a form

- Implemented: FR3.1, FR3.2, FR3.3, FR3.4, FR3.5

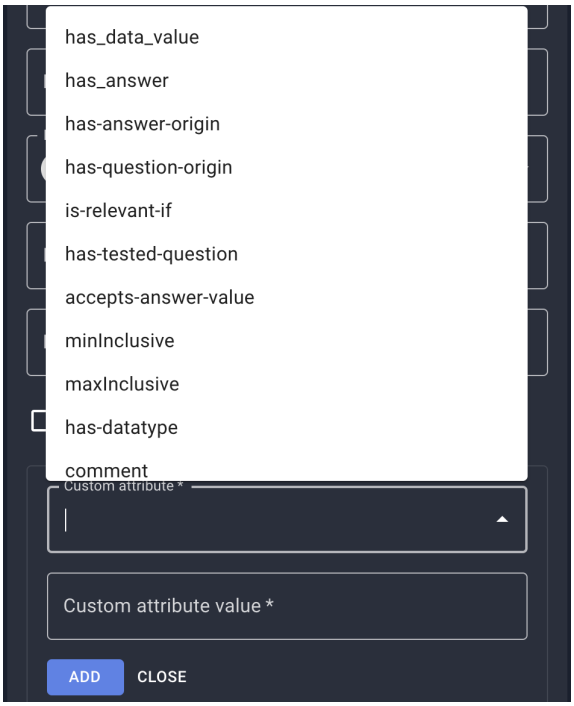
A user has three ways of creating a question, firstly by clicking on a plus button in the action bar, which creates a question on a root level. Next, by clicking on a plus button in the header of a question, which creates a subquestion of a question. The last option is to create a question at a specific place, which is introduced in the implementation of FR5.

When it is clicked on an add button or on a question, the form bar is opened. The first text field of the form bar is for *@id* of a question. This field cannot be modified. The *@id* property is generated from the provided label of a question. The generated *@id* is in Kebab case[9] with a random four-digit number at the end.

Some of the properties can be specified directly in its corresponding form element, such as a label, a layout class, a help, or a mask for a masked text form control. Other properties, which were defined in the *@context*, can be specified using *Add missing property* button, as it can be seen in Figure 6.3.

When the question is saved, the node is created and added to the map of a FormStructure object as well as the question is added as a subquestion to its parent question in the tree structure.

Figure 6.3: Adding of question properties not implemented in the form bar



Questions can be deleted using an action from the question menu, which appears on click on *three dots* button in the header of each question. In case of deleting a question

[9]Kebab case - All words within a sentence are lowercase and connected by a hyphen

containing subquestions, all of them are deleted. If the deleted question was preceding question of some other question, it loses the order.

A user can also add more questions at once by switching to the multiple question mode. A navigation bar between single question and multiple questions mode can be seen in Figure 6.2 in the top of the form bar. This navigation appears only during the creation of a new question. A user can type or paste questions into the provided text area. The format required for creating multiple questions is following. A form designer provides labels of questions, each on a new line. To create a subquestion of a question defined in the previous line, an indentation consisting of two spaces have to be added to the beginning of the label. This applies to any required level of a form. The questions which were followed by an indented label are created of type section or wizard step in case of a wizard form. The ones which were not followed by the indented label are of a type text field.

FR4: Reusability of questions

- Implemented: FR4.1
- Not Implemented: FR4.2, FR4.3

The duplication of a question can be done using an action from the question menu. The question is duplicated with all its properties. The only missing one is the preceding question property, which is removed.

FR5: Order of questions

- Implemented: FR5.4
- Partially implemented: FR5.1, FR5.2
- Not Implemented: FR5.3

As already mentioned in the implementation of FR3, the questions can be created using three methods:

- By the add button in the action bar, which creates an unordered question in the root level of a form.
- By the add button in the header of a specific question, which creates an unordered subquestion of that question.
- By adding a question to a specific place, by a blue rectangle, which can be seen in Figure 6.2 below the *Full name* question. Those rectangles are displayed on mouseover below and above each question. The newly added question by a blue rectangle is ordered and has *has-preceding-question* property set to a question above the blue rectangle, and a question below the rectangle has set a preceding question to the newly added question.

Every question can be moved by its header using Drag and Drop API across a form. If it is placed to the blue rectangle between questions, it becomes an ordered question, in the same way as newly added questions are. If it is placed on the section or wizard step question, it becomes an unordered subquestion. To create an unordered question on a root level, a drop area during a dragging process appears on the place of the form bar.

As mentioned earlier in the implementation of FR2, in a wizard form type only wizard step questions can be on a root level. If a section question is moved to a root level, its layout class is changed to a wizard step and vice versa.

This feature is implemented only partially in the prototype version. Preceding question property can be defined as an array of questions. However, in the current implementation, a question can have only one preceding question.

A form designer can remove preceding question property from a question using an option from a question menu of each ordered question.

FR6: Edit a source code of a form during a design process

- Implemented: FR6.1
- Not Implemented: FR6.2

A form designer can switch between edit mode in the user interface and edit mode of a source code of a form. To navigate to the second type of edit mode, a user can use a button in the action bar. The application provides a code editor using the *jsoneditor* library as in the implementation of FR1. The system transforms a form to the exporting format, more described in the implementation of FR13, and displays it in the code editor, where a form designer can modify it.

FR7: SForms form controls

- Partially implemented: FR7.1

All form controls from Table 4.1, except for dropdown, number field, SPARQL field and TURTLE field were implemented. The reason for not implementing excluded ones, is discussed in FR7.

A user can specify a form control type by an autocomplete form control labelled *Layout class* in the form bar. Together with the form control type, a user can select, for example, if a question is collapsed, hidden, or disabled. For specific form controls, new fields or buttons for modifying properties are displayed, such as for masked text the input mask field or for autocomplete the adding functionality of answer options is offered.

FR8: CRUD of answer options of closed questions

- Implemented: FR8.1, FR8.2, FR8.3, FR8.4
- Not implemented: FR8.5, FR8.6, FR8.7

The implementation of this requirement consisted of adding a button to the form bar to allow a user to specify answer options of autocomplete form controls. If a user clicks on the button, it opens a modal window. This modal window displays answer options of a question. A user can update them, add new ones or delete them. The answer options are created unordered. The ordering is not implemented, that means that the SForms library will sort the answer options according to its implementation. The current implementation sorts answer options alphabetically.

FR11: Multilingual forms

- Implemented: FR11.1, FR11.2, FR11.3
- Partially implemented: FR11.4

When a new form is created, and a user moves to edit mode, a form configuration modal window with settings of multilingualism and a form type is opened. Choosing of languages is implemented using autocomplete form control with an option to add the required languages. If a form with existing questions is imported, using the algorithm

of detection of all languages used in the label property is performed to determine which languages are used within a form and allow to work with them.

Only a label, a help and answer options can be specified in more languages using the user interface edit mode.

FR12: Preview a form in the SForms library

- Implemented: FR12.1, FR12.2, FR12.3, FR12.4, FR12.5
- Not implemented: FR12.6

Preview mode uses the SForms library to render the form created and modified in edit mode. To import the SForms library into the application, an update of the SForms library was required as discussed in Section 6.1.1.

If a form is multilingual, a dropdown with configured languages is provided. Also if the form is of a wizard form type, a switch between vertical and horizontal orientation of a wizard navigation bar is displayed to allow a form designer to test the form in all languages and layout orientations. In addition, it is possible to save answers filled into the preview of form which can serve as default answers. For adding the answers to the form structure, the *getFormQuestionsData* function from SForms is used, as discussed in Section 6.1.2. The obtained data from SForms are iterated, and found answers are added to the application form structure object.

FR13: Distribution of a form

- Implemented: FR13.1, FR13.2, FR13.3, FR13.4

Source code 6.4: Algorithm for compression of the form tree structure

```
static compressStructure = (rootNode) => {
    let object2IdMap = []; // mapping object -> id
    let idIncluded = new Set();
    object2IdMap = this._compressGraph(rootNode, object2IdMap, idIncluded);
    return object2IdMap;
};
static _compressGraph = (parentNode, object2IdMap, idIncluded) => {
  if (!idIncluded.has(parentNode['@id'])) {
    object2IdMap.push(parentNode);
    idIncluded.add(parentNode['@id']);
  }
  formShape.expandProperties.forEach((prop) => {
    if (parentNode.hasOwnProperty(prop)) {
      const childArray = parentNode[prop];
      for (let i = 0; i < childArray.length; i++) {
        const child = childArray[i];
        if (child !== undefined) {
          childArray[i] = child;
          object2IdMap = this._compressGraph(child, object2IdMap, idIncluded);
          parentNode[prop][i] = { '@id': child['@id'] };
        }
      }
    }
  });
  return object2IdMap;
};
```

When a form designer wants to export a form, the tree structure of a form, which was created using *expandStructure* algorithm from the SForms library is reversed. The tree structure reversion is done using a created algorithm *compressStructure*, which can

be seen in Source code 6.4. First of all, the properties, such as *has_related_question*, *has_answer*, *has_option*, which were expanded before, are now compressed back to the key-value format, where the value is represented by a list of *@id*s of compressed nodes. Afterwards, the structure which can be seen in Source code 4.1, is serialised into a compacted structure using *compact* algorithm from *jsonld* library. The parameters of *compact* function are set to a result of *compressStructure* algorithm and *@context* obtained during initialisation of a form.

There are three options of how a form designer can save the created or modified form. The form can be downloaded, copied to the clipboard, or it can be published to a *formUrl* specified in the pathname as a query parameter as shown in Source code 6.3. However, the form can be published using the POST HTTP request method only if it was previously loaded from an external source using the *formUrl* query parameter. Moreover, if the *draftUpdate* query parameter is set to a boolean value *true*, the modified source code of a form is sent to the specified *formUrl* on each change of a form structure using PUT HTTP request method.

FR19: Comment on problematic questions

- Implemented: FR19.2
- Not implemented: FR19.1

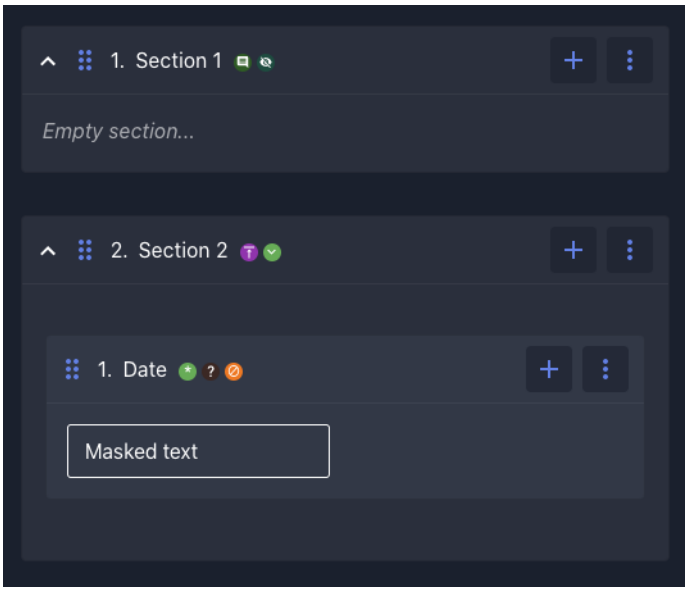
If a question has a comment property, an indicator of the comment is displayed in the header of a question. On mouseover, the indicator displays a preview of the content of the comment property.

FR20: Question properties overview without an interaction with a question

- Implemented: FR20.1, FR20.2, FR20.3, FR20.4, FR20.5, FR20.6, FR20.7, FR20.8

All indicators from FR20 were implemented and can be seen in Figure 6.4. The indicators are coloured circles with icons characterising their purpose. An exception is the form control type indicator, which is in the body of a question. For example, on mouseover the indicator of a help having a question mark icon, shows the content of a help property. On click the preceding question indicator, characterised by an arrow pointing to the rectangle above, shows the specific preceding question by highlighting.

Figure 6.4: Question properties overview in the header of questions



FR22: Collapsible and expandable sections

- Implemented: FR22.1, FR22.2

Questions of a type section and wizard step are implemented using the same React component which makes use of *Accordion* component from the Material-UI library. A user can use a chevron in the question header to collapse or expand an individual question. To expand or collapse all questions within a form at once, a user can use a button in the action bar.

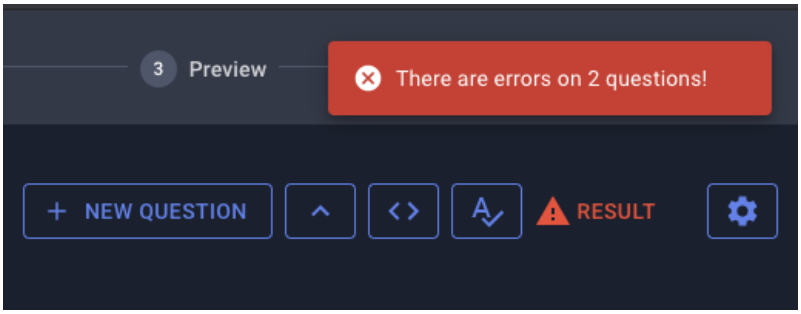
FR25: Visibility of system status

- Implemented: FR25.1, FR25.2

This requirement is solved by using the *notistack*[10] library, which provides a snackbar component, which can be used for showing brief messages. These messages are displayed only for a few seconds and inform a user about action states. The example of a brief message using *notistack* library can be seen in Figure 6.5. The error message is displayed after validating the form against errors for 5 seconds.

If a question is being updated or was moved, created or updated, the system highlights the question with a border

Figure 6.5: Notification about a form validation result



FR26: Validate a form

- Implemented: FR26.1, FR26.2, FR26.3, FR26.4
- Not implemented: FR26.5, FR26.6, FR26.7, FR26.8, FR26.9

A user can validate a form against ontology errors and best-practices by clicking on a button in the action bar. The form is validated using SHACL constraints specified in *best-practices-rules.shapes.ttl* and *ontology-rules.shapes.ttl* files in the application project.

For a validation, a JavaScript library *rdf-validate-shacl*[11] is used. As mentioned in Section 1.2.7, the SHACL constraints can be extended using SPARQL and JavaScript languages. However, the library supports validation only by using SHACL properties. To validate a form using this library, a validation rules and the form source code have to be loaded into a *Readable* object of a *stream*[12] library. Next, the parser imported from the *@rdfjs/parser-n3*[13] library is used for parsing the SHACL constraints and a parser from

[10] More information about Notistack available at `https://www.iamhosseindhv.com/notistack` to 25/11/2020

[11] rdf-validate-shacl library available at `https://github.com/zazuko/rdf-validate-shacl` to 27/11/2020

[12] stream library available at `https://github.com/juliangruber/stream` to 27/11/2020

[13] @rdfjs/parser-n3 library available at `https://github.com/rdfjs-base/parser-n3` to 27/11/2020

the *@rdfjs/parser-jsonld*[14] library is used for parsing the form source code. After that, parsed form source code and validation rules are loaded into a dataset using the *rdf-ext*[15] library. The dataset is a structure which provides functions for operations with the data loaded in a dataset, such as a function for matching triples or detecting duplicates.[11] Those datasets can be finally used with the *rdf-validate-shacl* library which validates the form source code against the SHACL constraints. When the validation is complete, a validation report is returned. This report is parsed by the application, and a result is displayed to a user.

A user is informed by the snackbar about the result. If there are issues in the form, the result can be opened in a modal window by clicking on a button in the action bar, as can be seen in Figure 6.5. Moreover, questions violating the validation constraints are indicated in a similar way as properties in the implementation of FR20.

The implemented best-practices validation constraints can be found in Section A.1. Only the constraints which can be easily implemented using the SHACL constraints without the use of SPARQL or JavaScript language were implemented. There are seven implemented validation constraints completely and one partially. One of them is illustrated in Source code 1.5. To demonstrate the ontology-based validation constraints two rules were specified, specifically the validation of the presence of a label and layout class.

FR29: Log application errors

- Implemented: FR29.1

The errors are logged using *trackjs*[16] library. This library sends JavaScript errors together with information about a user and activities done within an application to a TrackJS[17] management system, which is free of charge for open-source projects. In the management system, a developer can display the errors and based on them reproduce the problems and fix them.

6.2.5 Implementation of Non-Functional Requirements

All non-functional requirements were taken into account during the creation of the application. The ones which affected the implementation are described in this section.

NFR1: System will be working in the newest versions of browsers

Supporting of the application in the newest versions of browsers is realised automatically by Next.js framework itself. The application in terms of supported browsers will be tested in Section 7.1.1.

NFR2: System will be working on devices with display width >= 1000 pixels

This requirement was taken into account during the implementation. However, if the form designer makes a multilevel form with long labels, he should use a device with a screen wider than 1000 pixels as the longer labels can break the design.

[14] @rdfjs/parser-jsonld library available at `https://github.com/rdfjs-base/parser-jsonld` to 27/11/2020

[15] rdf-ext library available at `https://github.com/rdf-ext/rdf-ext` to 27/11/2020

[16] trackjs library available at `https://www.npmjs.com/package/trackjs` to 26/11/2020

[17] More information about TrackJS available at `https://trackjs.com/` to 26/11/2020

NFR3: System will be in English

All texts in the application are provided in English. Localisation to other languages was not taken into account during the implementation.

NFR4: System allows creating of questions using keyboard

A form designer can navigate through the application only by a tab key except for new / import mode and edit mode of the source code of a form. The limitation is due to a problem with the *jsoneditor* library, which uses a tab key for indentation.

In addition to the tab key navigation, a modified question in the form bar can be saved using the *Enter* key.

NFR8: System will be created in the newest versions of mainstream web technologies

The application was implemented in the newest versions of technologies mentioned in Section 6.2.1. These technologies, specifically React, Next.js and TypeScript, are trending in web development for the year 2020.[8]

Chapter 7

Testing and Evaluation

This chapter discusses the testing methods used in the prototype. In addition, it evaluates the Semantic Form Editor from the perspective of provided functionalities, as well as the usability of the application in real environment on real data.

7.1 Testing

Unit testing, end-to-end testing and user testing were performed on the Semantic Form Editor. Four users participated in the user testing of the application. Next, the application was tested in the newest versions of Google Chrome, Safari and Firefox browsers.

7.1.1 Browser Testing

Due to NFR1, the system should be working in the last three versions of Google Chrome, Safari and Firefox. Therefore, the application was tested in the last versions of these browsers, as well as in the older versions of them. Due to NFR2, the width of a browser during the testing was set to 1000 pixels. For the testing, the scenarios prepared for user testing from Section B.1.3, were used. The result can be seen in Table 7.1.

Table 7.1: Browser testing

Operating system	Browser	Supported
macOS 11.0.1	Google Chrome 85.0	Yes
macOS 11.0.1	Google Chrome 87.0	Yes
macOS 10.15.5	Safari 13.0.5	Yes
macOS 11.0.1	Safari 14.0	Yes
macOS 11.0.1	Firefox 81.0.2	Yes
macOS 11.0.1	Firefox 83.0	Yes

7.1.2 Unit Tests

Due to NFR5, the core functionalities, such as building a form structure, moving questions or exporting a form, were covered by 37 unit tests using Jest testing library. The coverage of logic functionalities is 81.67%. Other parts of the application are not tested by unit tests because of the time constraints.

7.1.3 End-to-End Testing

Due to NFR6, the Semantic Form Editor should by tested by end-to-end tests. The application as a whole was tested using Cypress testing framework. The majority of im-

plemented requirements, which were able to be tested in a browser, were tested by 42 Cypress tests. The requirements which were not tested are listed below:

- FR13, the distribution of a form was not tested, because of unsuccessful testing of downloaded files and content of the clipboard.
- FR25, the system status was tested only partially.
- FR29, due to the fact that it is not possible to test the tracking to TrackJS using the test environment.

All tests were run in Chrome 87.0, Edge 87.0, and Firefox 81.0.2. The code coverage reaches values higher than 75% in all four coverage criteria. Those are statements, branches, functions and lines. Part of the report can be seen in Figure 7.1. The percentages under the title demonstrate the coverage in the specific criteria already mentioned. The table shows the coverage of individual React components and JavaScript files.

Figure 7.1: The code coverage using end-to-end tests

All files

87.38% Statements 1392/1593 **77.02%** Branches 496/644 **90.41%** Functions 396/438 **87.45%** Lines 1359/1554

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements		Branches		Functions		Lines	
cypress/helpers	100%	50/50	100%	0/0	100%	22/22	100%	50/50
src/components/editors/Editor	95.83%	23/24	80%	8/10	100%	6/6	95.83%	23/24
src/components/editors/EditorCustomise	85.71%	18/21	95%	19/20	80%	4/5	85.71%	18/21
src/components/editors/EditorCustomiseCode	82.86%	29/35	50%	1/2	66.67%	6/9	82.86%	29/35
src/components/editors/EditorExport	57.14%	16/28	83.33%	5/6	66.67%	4/6	57.14%	16/28
src/components/editors/EditorNew	85%	51/60	71.43%	10/14	83.33%	10/12	85%	51/60
src/components/editors/EditorPreview	86.84%	33/38	73.33%	11/15	66.67%	6/9	94.12%	32/34
src/components/items/Item	85.71%	18/21	66.67%	4/6	85.71%	6/7	85%	17/20
src/components/items/ItemAdd	72.73%	64/88	53.45%	31/58	87.5%	14/16	72.41%	63/87

7.1.4 User Testing

User testing evaluates on a set of users if the design decisions made during the implementation of the application were correct.[35] The goals were to test if the application is easy to use and to identify bugs and issues with the application. Due to the fact, that the user roles from FR27 were not implemented, the target group of participants were users who have at least basic level of ontology knowledge or coding skill.

The user testing was conducted remotely involving four participants using their computers, video communication software with voice and video enabled and with the feature of screen sharing. The participants' profiles containing their education, their level of coding skill and ontology knowledge, as well as the operating system of their computer, can be seen in Section B.1.1. Before starting with test scenarios, the participants were introduced into the problematics of the form editor by an introduction in Section B.1.2. Eventual questions concerning the introduction were answered. The application was tested by two prepared scenarios with together 36 tasks, as can be seen in Section B.1.3. After the testing was performed, the participants were asked a few post-test questions, shown in Section B.1.4. The duration of testing was measured and took on average nearly an hour per each participant as can be seen in Section B.3.

In total, 17 issues of different severity were found. The majority of them were only minor problems with user experience, except for one, which prevented importing a form source code by pasting it to the editor as required by FR1.4. The findings can be seen in Section B.4. Some of the findings, which were relevant, including the severe one, were fixed in the prototype version of the application. The rest of them will be fixed in the next versions of the form editor. The overall user testing met with success, as the participants were able to complete the scenarios without the need of interventions from the moderator and significant struggling during the testing process. The answers of post-test questions can be seen in Section B.2.

7.2 Evaluation

This section discusses the evaluation of the Semantic Form Editor. It is divided into two sections. The first one verifies, that the implemented form editor sustains complex data in terms of forms from real applications. The second one compares the implemented form editor to other already existing form editor solutions.

7.2.1 Verification of Functioning with Complex Forms

The Semantic Form Editor was tested using three complex forms, which are being used in three clinical trials, namely in FERTISS, ABRAX, and CITECO clinical trial[1]. These clinical trials are used in the StudyManager application.

- FERTISS form consists of 358 questions. The application loads the form without problems. The loading of the form takes about three seconds. The navigation in edit mode is smooth and the response times of modifying the form are in the order of hundreds of milliseconds. Overall the form editor handles the FERTISS form without problems.

- ABRAX form consists of 537 questions. The application loads the form without problems. The loading of the form takes about five seconds. The navigation in edit mode is smooth and the response times of modifying the form are in the order of hundreds of milliseconds. Overall the form editor handles the ABRAX form without problems.

- CITECO form consists of 1,546 questions. The application loads the form without problems. The loading of the form takes about ten seconds. The navigation in edit mode is not entirely smooth. The response times of modifying the form are also worse compared to FERTISS and ABRAX forms. It takes about two to three seconds to update the questions. Overall it is possible to use the form editor for small changes of the such large form, but making complex changes, such as displacing the questions, would be strugglesome.

The verification of functionality on complex forms was conducted on the MacBook Pro (16-inch, 2019) with 2,6 GHz 6-Core Intel Core i7 processor and 16 GB RAM, in Google Chrome 87 browser.

Based on the testing, the application is able to handle complex forms consisting of at least 537 questions without problems. However, if the application is used for more complex form such as the CITECO form, containing 1,546 questions, the application starts slowing down.

[1] More information about those clinical trials can be found at `https://trial4you.eu/` to 30/11/2020

One of the possible solutions for the application slowing down with complex forms, is integrating one of the libraries *react-window*[2] or *react-virtualized*[3] to the next versions of the form editor. Those libraries would optimise rendering in a way that only the part of the form, which is visible on the user's screen, is rendered. This optimisation would speed up the process of re-rendering the form on each update. Another possible solution applicable for complex wizard forms is modifying the user interface, so it always only renders the questions from one wizard step.

7.2.2 Comparison with Existing Solutions

The selected properties of form editors, described in Section 3.4, are used for the comparison of the prototype version of the Semantic Form Editor with other existing solutions in Table 7.2. SurveyMonkey, Google Forms and formBuilder from Chapter 3 are chosen for the comparison. The reason these tools were chosen, is that they are more powerful compared to other form editors mentioned in that section.

Table 7.2: Comparison of selected properties of the Semantic Form Editor with existing solutions

	The Semantic Form Editor	SurveyMonkey	Google Forms	formBuilder
Form types	**Classic; Wizard**	Classic; Card; Multi-step; Conversation	Classic; Multi-step	Classic; Wizard
Reusability of questions	**Partially*; By duplication**	Question bank; By duplication	Import from other form; By duplication	By duplication
Reusability of answer options	**No***	Predefined options	By add-on	No
Validation of form design	**Partially***	Yes	No	No
Forms satisfying form design guidelines	**No#**	No	No	No
Customisable design of questions and answer options	**No***	Yes	No	Yes
Other answer option	**No***	Checkbox with a text field or only a text field	Checkbox with a text field	No
Conditional logic and branching	**No***	Yes	Yes	No
The tree structure of forms	**Yes**	No	No	No
Ordered and unordered questions	**Partially**	No	No	No
Ordered and unordered answer options	**No***	No	No	No
Multilingual forms	**Yes**	Yes	No	No
The Semantic Web forms	**Yes**	No	No	No

* - Designed but not (fully) implemented in the prototype version.
\# - Requires modifications of the SForms library.

[2]More information about react-window library available at `hhttps://react-window.now.sh/` to 30/11/2020

[3]More information about react-virtualized library available at `https://bvaughn.github.io/react-virtualized/` to 30/11/2020

Conclusion

The goal of this master thesis was to design form editor for efficient and high-quality data collection and implement prototype demonstrating its core features. The Semantic Web is a perspective technology which, as an extension of the World Wide Web, creates a network of interconnected data. The data is in a format that is readable not only by humans but also by computers, which allows the computers to access the data more intelligently and efficiently. However, the Semantic Web is a technology yet to be understood and taken advantage of. There is a sparse amount of technologies utilising the Semantic Web, including the one for data collection, a form. SForms is one of few technologies capable of representing forms in the format of the Semantic Web.

The guidance of the user to create valid forms according to the form design guidelines represents an important trait of the implemented application. Valid forms are overall more understandable and less strugglesome for their respondents. In result, these forms are more efficient and are prone to produce higher-quality data. To fulfil this goal, the implemented form editor validates the created forms against form best-practices.

The practical part of this thesis is further supported by a theoretical analysis of related topics. The thesis studies the problematics of the Semantic Web technologies, as well as outlines current form design guidelines. The implementation phase of the editor was preceded by a collection of requirements and an analysis of already existing form editor solutions. In addition, the SForms library was analysed to discover that some parts of the library had to be modified in order to fulfil all implemented requirements. Also, the visual representation of SForms forms was found insufficient for the proposed form designed guidelines.

Testing of the implemented form editor was performed gradually during development, and the final version of the form editor prototype was tested with four users. The user testing altogether met with success, as the participants appreciated the overall experience with an editor. The user testing discovered a number of shortcomings, but they generally did not harm the functioning of the prototype. One severe error was found, and it was fixed along with a few minor ones.

The Semantic Form Editor is now a functioning prototype that can be used by form designers creating semantic forms. The created forms can be validated against proposed form design guidelines. However, the validation supports only a part of the applicable best-practices, as the used library for validation using SHACL lacks the possibility of extension of SHACL validation constraints by SPARQL and JavaScript. The future work could solve this problem by using a server for the purpose of validation, as there are existing solutions that have the functionalities that the frontend validation library lacks.

The other parts of the application still leave plenty of room for improvement. There are still requirements implemented partially or not at all. For example, the ordering of answer options or conditional logic and branching are both functionalities that can be the subject of future work for this form editor. Moreover, the user testing indicated some minor issues with the user interface. Also, the evaluation pointed at the slowdown of application with large complex forms. Finally, the SForms library can also be further extended. Firstly, to support the remaining requirements of the Semantic Form Editor.

Secondly, SForms needs to be modified to be in compliance with the proposed current best-practices of form design.

All requirements of the form editor prototype in the scope of this master thesis were implemented and tested. Therefore, I think the goals set were satisfied. I am glad I could participate in a project that I could design and implement from scratch, which provided me with useful experience. On top of that, I was given the opportunity to work with powerful web technologies, such as JSON-LD, SHACL, Next.js, and Cypress.

References

[1] N. Babich. Designing efficient web forms: On structure, inputs, labels and actions, 2017. URL https://www.smashingmagazine.com/2017/06/designing-efficient-web-forms/. [Accessed: 28 October 2020].

[2] J. A. Bargas-Avila, O. Brenzikofer, S. Roth, A. Tuch, S. Orsini, and K. Opwis. Simple but crucial user interfaces in the world wide web: introducing 20 guidelines for usable web form design, user interfaces. 2010.

[3] J. Carroll and G. Klyne. Resource description framework (RDF): Concepts and abstract syntax. W3C recommendation, W3C, 2004. URL https://www.w3.org/TR/2004/REC-rdf-concepts-20040210/. [Accessed: 2 November 2020].

[4] *Contrast and Color Accessibility*. Center for Persons with Disabilities, 2018. URL https://webaim.org/articles/contrast/#sc146. [Accessed: 28 October 2020].

[5] Colblindor. Colorblind population, 2006. URL https://www.color-blindness.com/2006/04/28/colorblind-population/. [Accessed: 28 October 2020].

[6] W. W. W. Consortium et al. W3c semantic web. URL https://www.w3.org/standards/semanticweb/. [Accessed: 7 November 2020].

[7] Cypress.io. Cypress. URL https://www.cypress.io/. [Accessed: 19 November 2020].

[8] B. C. Dylan Schiemann. Javascript and web development infoq trends report 2020, 2020. URL https://www.infoq.com/articles/javascript-web-development-trends-2020/. [Accessed: 25 November 2020].

[9] J. Enders. *Designing UX: Forms: Create Forms That Don't Drive Your Users Crazy*. SitePoint, 2016. ISBN 9781492017547. URL https://books.google.com/books/about/Designing_UX.html?id=ISAWnQAACAAJ.

[10] forms.app. forms.app. URL http://forms.app/. [Accessed: 28 October 2020].

[11] R. J. L. C. Group. rdf-ext, 2018. URL https://github.com/rdf-ext. [Accessed: 25 November 2020].

[12] S. Harris and A. Seaborne. SPARQL 1.1 query language. W3C recommendation, W3C, 2013. URL https://www.w3.org/TR/2013/REC-sparql11-query-20130321/. [Accessed: 4 November 2020].

[13] P. Hayes. RDF semantics. W3C recommendation, W3C, 2004. URL https://www.w3.org/TR/2004/REC-rdf-mt-20040210/. [Accessed: 2 November 2020].

[14] M. H. Hoeflich. Legal forms and the practice of law in nineteenth-century kansas. *U. Kan. L. Rev.*, 64:1055, 2015.

[15] F. Inc. React, . URL https://reactjs.org/. [Accessed: 19 November 2020].

[16] G. Inc. Google forms, . URL https://docs.google.com/forms/. [Accessed: 28 October 2020].

[17] J. Inc. Jotform, . URL https://www.jotform.com/. [Accessed: 28 October 2020].

[18] S. Inc. Surveymonkey, . URL https://www.surveymonkey.com/. [Accessed: 28 October 2020].

[19] I. Jacobs, D. Raggett, and A. L. Hors. HTML 4.01 specification. WD not longer in development, W3C, Mar. 2018. URL https://www.w3.org/TR/2018/SPSD-html401-20180327/. [Accessed: 1 November 2020].

[20] B. Jue. Survey design tips: Is it better to have one question per page or multiple questions on a page?, 2019. URL https://www.focusvision.com/blog/survey-design-tips-is-it-better-to-have-one-question-per-page-or-multiple-questions-on-a-page/. [Accessed: 28 October 2020].

[21] T. Klíma. Semantic manager for prospective clinical trials. B.S. thesis, CTU in Prague, 2018. URL http://hdl.handle.net/10467/76531.

[22] H. Knublauch. Dash data shapes vocabulary, 2020. URL http://datashapes.org/dash. [Accessed: 4 November 2020].

[23] H. Knublauch. Form generation using shacl and dash, 2020. URL http://datashapes.org/forms.html. [Accessed: 4 November 2020].

[24] H. Knublauch and D. Kontokostas. Shapes constraint language (shacl). W3C recommendation, W3C, 2017. URL https://www.w3.org/TR/2017/REC-shacl-20170720/. [Accessed: 4 November 2020].

[25] G. Kuck. Tim berners-lee's semantic web. *SA Journal of Information Management*, 6 (1), 2004.

[26] D. Longley, P.-A. Champin, and G. Kellogg. JSON-ld 1.1. W3C recommendation, W3C, 2020. URL https://www.w3.org/TR/2020/REC-json-ld11-20200716/. [Accessed: 4 November 2020].

[27] Material-UI. Material-ui. URL https://material-ui.com/. [Accessed: 19 November 2020].

[28] J. Matthews. Should i use a drop-down? four steps for choosing form elements on the web, 2001. URL https://www.effortmark.co.uk/should-i-use-a-drop-down/. [Accessed: 28 October 2020].

[29] D. McGuinness and F. van Harmelen. OWL web ontology language overview. W3C recommendation, W3C, 2004. URL https://www.w3.org/TR/2004/REC-owl-features-20040210/. [Accessed: 3 November 2020].

[30] I. Merriam-Webster. Merriam-webster online dictionary. URL https://www.merriam-webster.com/. [Accessed: 28 October 2020].

[31] S. Messenger. Moscow prioritisation, 2014. URL https://www.agilebusiness.org/page/ProjectFramework_10_MoSCoWPrioritisation. [Accessed: 19 November 2020].

[32] S. Mestel. How bad ballot design can sway the result of an election, 2019. URL https://www.theguardian.com/us-news/2019/nov/19/bad-ballot-design-2020-democracy-america. [Accessed: 28 October 2020].

[33] Microsoft. Typescript. URL https://www.typescriptlang.org/. [Accessed: 19 November 2020].

[34] *The Input (Form Input) element*. Mozilla Contributors, 2020. URL https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input. [Accessed: 28 October 2020].

[35] C. Murphy. A comprehensive guide to user testing, 2018. URL https://www.smashingmagazine.com/2018/03/guide-user-testing/. [Accessed: 28 October 2020].

[36] J. Nielsen. Ten usability heuristics, 2005. URL http://www.nngroup.com/articles/ten-usability-heuristics/. [Accessed: 20 November 2020].

[37] O. U. Press. Oxfordreference.com. URL https://www.oxfordreference.com/. [Accessed: 7 November 2020].

[38] M. Seckler, S. Heinz, J. Bargas-Avila, K. Opwis, and A. Tuch. Designing usable web forms – empirical evaluation of web form improvement guidelines. *Proceedings of the 2014 annual conference on Human factors in computing systems*, pages 1275–1284, 2014.

[39] M. Sporny, D. longley, G. Kellogg, M. Lanthaler, and N. Lindstöm. Json-ld 1.1, 2019. URL https://json-ld.org/spec/latest/json-ld/. [Accessed: 15 November 2020].

[40] S. s.r.o. Question types 4: Semi-closed / semi-open questions, 2020. URL https://www.survio.com/en/blog/question-types-4-semi-closed-semi-open-questions/. [Accessed: 6 December 2020].

[41] O. Stoica. What is a form? well, let's tell you!, 2018. URL https://www.123formbuilder.com/blog/what-is-a-form/. [Accessed: 7 November 2020].

[42] M. M. Taye. Understanding semantic web and ontologies: Theory and applications. *arXiv preprint arXiv:1006.4567*, 2010.

[43] M. Taylor. S58 form design best practices & form ux examples, 2020. URL https://www.ventureharbour.com/form-design-best-practices/. [Accessed: 28 October 2020].

[44] R. Tourangeau and K. A. Rasinski. Cognitive processes underlying context effects in attitude measurement. *Psychological bulletin*, 103(3):299, 1988.

[45] I. Vercel. Next.js by vercel. URL https://nextjs.org/. [Accessed: 19 November 2020].

[46] K. Whitenton. Website forms usability: Top 10 recommendations. *Nielsen Norman Group*, 2016. URL https://www.nngroup.com/articles/web-form-design/. [Accessed: 28 October 2020].

[47] D. Wood, R. Cyganiak, and M. Lanthaler. RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C, Feb. 2014. URL https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/. [Accessed: 1 November 2020].

[48] L. Wroblewski. *Web Form Design: Filling in the Blanks*. Rosenfeld Media, 2008. ISBN 9781933820248. URL https://books.google.com/books?id=AkBB3XKifiEC.

List of Figures

List of Tables

List of Source Codes

Appendix A

List of Form Best-Practices

This appendix summaries best-practices from Chapter 2, which are marked by either * or †. Section A.1 is a summary of best-practices, which can be used for validation of a form structure using SHACL language. Section A.2 represents the summary of best-practices which are applicable to a form rendering application, concretely the SForms library.

A.1 Best-Practices Applicable for Validation Using SHACL

The left column in Table A.1 shows a best-practice constraint, and the right column indicates if the specific constraint is implemented in the prototype version of the application.

Table A.1: Overview of best-practices applicable for validation using SHACL

Best-practice constraint	**Implemented**
Full-sentence labels should not be mixed with brief prompts labels, unless the situation requires it.	No*
Punctuation at the end of labels should be consistent.	No*
Personalised wording like *My* or *Your* should be avoided in labels, unless the situation requires it.	Yes
Form title should be present.	Yes
Form title should not be longer than five words.	Yes
Words like *form*, *online*, *web*, *for* and articles *a*, *an*, *the* should be omitted in a form title.	Yes
In terms of a multi-step or a wizard form type, numbered steps should be avoided, when there is a large number of steps.	No*
As few form control types as possible should be used to avoid confusion of a user.	No*
Questions with one checkbox can be designed using two radio buttons.	#
Autocomplete and dropdown form controls should have at least four answer options.	Yes
Time pickers should be omitted, masked text or text field with question-level help should be used instead.	Yes
Labels should be written in sentence case.	Partially
A multi-step and a wizard form type should have less than eight steps.	No*
Default answers should be avoided, unless statistics show that an answer has a high probability to be chosen by a user.	Yes
Conditional logic and branching should not cause dead end.	No*

* - Can be implemented using SPARQL or JavaScript in SHACL. However, this functionality is not supported by *rdf-validate-shacl* library, and therefore, not implemented in the prototype version.

- Cannot be applied for the current version of the SForms.

A.2 Best-Practices Applicable to SForms

The left column in Table A.2 shows a best-practice constraint, and the right column indicates if the constraint is already satisfied in the SForms library.

Table A.2: Overview of best-practices applicable to SForms

Best-practice rule	**Available**
Every form should have a title.	No
Every section should have a title.	Yes
Every step should have a title.	Yes
The minority of either required or optional questions should be marked with an indicator.	No
If there is less required questions than optional ones, then it should be stated in the top of the form by *All questions are optional unless marked (required)*.	No
Designing a form by using text fields, single-choice and multiple-choice form control types should be possible.	No
Form elements and buttons should have at least 48 pixels height.	No
The gap between form controls should be between 50% to 75%.	Yes
Radio buttons and checkboxes should allow interaction by clicking on their label.	Yes
Input form element should have a type corresponding to the expected answer.	Yes
Answer field size should correspond to the expected length of an answer.	Yes
Question help should be visible without an interaction.	No
Reset and clear buttons should be avoided. If it is required, the undo action should be provided.	*
Regular font typeface should be used, italics and oblique typefaces should be avoided.	Yes
Bold font typeface should be used only for titles and emphasis.	No
Font size used in form controls should be of size at least 16 pixels.	Yes
Errors should be provided both with an icon and a message.	No
Non-text elements should have at least 3:1 contrast.	Yes
Text elements should have at least 4.5:1 contrast.	Yes
Navigation should be possible using keyboard.	Yes
There should be an option to provide a welcome page.	No
There should be an option to provide a thanks page.	No
Review page of answers should be provided in case of a long form.	No
Question-level error messages should be visually attached to the error causing question.	Yes
Longer forms should have errors summary in the top of the form.	No
User should be unable to hit the submit button twice.	*
Longer processes such as submitting of a form should be indicated.	*
Name of a button should accurately indicate the purpose of the button.	*

* - Functionality is out of scope SForms.

Appendix B

User Testing

B.1 Testing Setup

This section represents the materials the participants obtained during the user testing.

B.1.1 Participant Profiles

Basic information about the participants of the user testing can be seen in Table B.1.

- The basic level of coding or ontology-based knowledge represents that a participant understands the concept of the domain, but has slight or no experience.
- The intermediate level of coding or ontology-based knowledge represents that a participant understands the concept and has junior experience in the domain.
- The advanced level of coding or ontology-based knowledge represents that a participant has rich experience in the domain.

Table B.1: Participant profiles

	Education	**Coding skill**	**Ontology knowledge**	**Operating system**
Participant 1	High school	Intermediate	None	MacOS 10.15
Participant 2	Higher	Basic	None	Windows 10
Participant 3	Higher	Advanced	None	Ubuntu 20.04
Participant 4	Higher	Advanced	Intermediate	Windows 10

B.1.2 Introduction

The application is a form editor which allows a user to create smart ontology-based forms. The form created is represented in JSON-LD format, which is afterwards used in different applications for visualisation of the form. The form has a tree structure, that means that there are questions which can contain subquestions. The editor allows a user to create two types of forms: the classic form, where all questions are displayed on one screen and the wizard form, where questions are divided into more pages (wizard steps). Some of the basic types of questions are, for example, text fields and text areas. The form structure also contains sections and wizard steps, which are represented as questions in the form editor. Both can contain subquestions, as they are used for grouping of questions. The questions can be ordered or unordered within the form. That means that if the question is ordered, it has set the preceding question, if it is unordered it is, in this

case, sorted alphabetically. The application allows creating a new form or importing an existing one, editing a form in a user interface or in a code. In addition a user can preview the created form and export it.

B.1.3 Scenarios and Tasks

B.1.3.1 Scenario 1

1. Open `https://semantic-form-editor.now.sh/` in your browser.
2. Start creating a new form.
3. Save the created form, set this form to be of a classic form type and make it multilingual by allowing support for English and Czech language (in this order).
4. Create a question of type *Section* with English label *Person* and Czech label *Osoba*, do not specify any additional properties.
5. Add a question of type *Text field* to the *Person* section with English label *Full name* and Czech label *Celé jméno*. Make the answer of this question required.
6. Add a question of type *Autocomplete* question to the *Person* section with English label *Title* and Czech label *Titul*. Add three answer options, in English: *Mr.*, *Mrs.* and *Ms.*, and in Czech: *Pan*, *Paní* and *Slečna*.
7. Move the question with label *Title* before the question with label *Full name*.
8. Create a question of type *Section* that is placed after the created *Person* section. The section should be on the same level as the *Person* section. You should not use the Drag and Drop to place the question. The section should have English label *Job* and Czech label *Práce*.
9. Add a *Single checkbox* question to the *Job* section with English label *Are you employed?* and Czech label *Jste zaměstnaný?*.
10. Navigate to preview mode.
11. Check the form in the Czech language.
12. Navigate back to edit mode and make the *Job* section to be collapsed.
13. Preview the form again.
14. Export the form by downloading.
15. Start the process of form editing again.
16. Import the downloaded form and preview the form again.

B.1.3.2 Scenario 2

1. Open `https://semantic-form-editor.now.sh/` in your browser.
2. Start creating a new form.
3. Save this created form and set this form to be of a wizard form type.
4. Create a question of a wizard step type with label *MUSIC*.
5. Add a text area question to *MUSIC* section with label *What do you think about this song?*.
6. Navigate to edit mode of a source code of a form.
7. Add a property code below to the *@context* part of the source code of a form.

```
"has-media-content": {
   "@id":"http://onto.fel.cvut.cz/ontologies/form/has-media-content"
}
```

8. Find a question with label *What do you think about this song?* and delete its *has-layout-class* property.
9. Save changes and get back to the user interface edit mode.
10. Hit validate a form.
11. Find questions with issues and fix them.
12. Add *has-media-content* property to the *Music* question with URL to your favourite song on YouTube[1] (The URL has to be in the embed format. The embed format can be found by clicking on the share button under the YouTube video. Choose *Embed* and copy the URL next to the *src* property).
13. Duplicate the *Music* section.
14. Delete the *What do you think about this song?* question in the duplicated section.
15. Rename the duplicated *Music* section to *Empty tab* and remove its *has-media-content* property.
16. Preview the form.
17. Answer the question *What do you think about this song?* and save the answer to the form.
18. Get back to edit mode and change the form type to the classic form type.
19. Preview the form again.
20. Export the form by copying the source code of the form to the clipboard.

[1] Online video sharing platform available at `https://www.youtube.com/` to 23/11/2020

B.1.4 Post-Test Questions

- What was your favourite part of the application?
- What was the most confusing part of the application?
- How is your overall experience with this application?

B.2 Post-Test Questions Answers

In this section, questions from Section B.1.4 are answered by the participants.

- What was your favourite part of the application?
  1. The participant admired the styling of the application. The application did what was expected and the processes were understandable.
  2. The participant liked the possibility to download a source code of a form and its content.
  3. The participant liked that he can add properties to the *@context* part of the form source code and use it in the form editor.
  4. The participant liked that he can edit the form either using the user interface or the editor of the source code of a form.
- What was the most confusing part of the application?
  1. Configuration of question type using form control with label *Layout class*.
  2. Configuration of question type using form control with label *Layout class*. (The same as with participant 1)
  3. Questions are sorted alphabetically by default.
  4. The presence of multiple add question buttons with different purposes. The naming of form controls, such as *Layout class*.
- How is your overall experience with this application?
  1. The editor always did what the participant expected, so it seemed intuitive.
  2. The participant rated the experience with the editor by 8 points out of 10.
  3. From a technical perspective, the participant experienced only one bug, which happened after loading a previously saved form. Overall, he would say that the application is in a great shape (short response times etc.).
  4. The participant rated the experience with the editor by 7.2 points out of 10.

B.3 Duration of User Testing

The duration of user testing was measured starting with introduction and ending with answering of the post-test questions. The average duration of one user testing session was almost 58 minutes. The final duration per participant can be seen in Table B.2.

Table B.2: Duration of user testing per participant

Participant	Duration
Participant 1	46 minutes
Participant 2	53 minutes
Participant 3	58 minutes
Participant 4	74 minutes

B.4 Problems Found

The findings are divided into three categories according to its severity. Together with the severity, every finding contains a participant, a scenario and a task, where it was found, as well as a description of the issue. Some of the findings also contain a reason why the specific severity was chosen and a piece of information whether the problem was fixed in the prototype version. The categories are listed below:

- High severity
  - A finding that prevents using the application or has a big impact on the usage of the application. It should be fixed immediately.
- Medium severity
  - A finding that has only a little impact on the usage of the application, but it should be fixed as soon as possible.
- Low severity
  - A finding that has only a little impact on the usage of the application. It can be fixed later.

Finding 1: Form element for selecting form controls labeled by *Layout class*

- **Severity:** Medium
- **Found by participants:** 1, 2, 3, 4
- **Found in scenario:** 1
- **Found in task:** 4
- **Description:** The participants could not find how to set a question type, because the form control for setting the question type was named unaccordingly.
- **Reason of severity:** The *Layout class* form control should be renamed, because without knowing a structure of a form source code, a form designer cannot know, that the *Layout class* form element serves for choosing form control type.
- **Fixed:** The wording of the label was changed to *Question type and properties*.

Finding 2: The form bar closes on click outside

- **Severity:** Medium

- **Found by participants:** 3, 4
- **Found in scenario:** 1
- **Found in task:** 4
- **Description:** The participants clicked outside of the form bar component during the configuring process of a question, and the form bar closed. They did not expect the closing to happen. Also, they expected that the configuration data of a question would save automatically.
- **Reason of severity:** Participants did not expect that they would lose the filled data when clicking outside the form bar component.

Finding 3: Order of answer options in *Layout class* form control

- **Severity:** Low
- **Found by participants:** 4
- **Found in scenario:** 1
- **Found in task:** 4
- **Description:** The participant expected the answer options of the *Layout class* form control to be sorted alphabetically when there is no natural ordering of the answer options.
- **Reason of severity:** It does not have a big impact on the usage of the application.
- **Fixed:** The *Layout class* form control answer options are now sorted alphabetically.

Finding 4: Id of a question

- **Severity:** Low
- **Found by participants:** 1, 4
- **Found in scenario:** 1
- **Found in task:** 4
- **Description:** The participants were confused by the fact that the *@id* form control is displayed in the top of the form bar, but it cannot be modified. The participants suggested hiding the *@id* form control and displaying it only when a user desires.
- **Reason of severity:** This finding does not have a big impact on the usage of the application.

Finding 5: Adding a question to a section question

- **Severity:** Low
- **Found by participants:** 2, 4
- **Found in scenario:** 1

- **Found in task:** 5
- **Description:** The participants looked for an option to add a subquestion of a section in its menu and form bar, but it was not present there.
- **Reason of severity:** The subquestion to a question can be added by a plus button on the left of the question menu.
- **Fixed:** The option to add a subquestion was also added to the question menu.

Finding 6: Adding answer options

- **Severity:** Low
- **Found by participants:** 3
- **Found in scenario:** 1
- **Found in task:** 6
- **Description:** The participant overlooked the *Add answer options* in the form bar. He suggested making the button larger. Moreover, he was not sure if he added answer options correctly. Therefore he suggested indicating it more clearly.
- **Reason of severity:** The add button is located under in the form bar under *Remote answer options URL* and displays how many answer options there are.

Finding 7: Adding a question to a specific place

- **Severity:** Low
- **Found by participants:** 2, 3
- **Found in scenario:** 1
- **Found in task:** 8
- **Description:** The participants struggled with finding how to add a question directly to a specific place. They overlooked areas indicating the possibility to add a question to that area, which are only visible on mouseover. They suggested showing those areas always, without any need of an interaction and highlighting those furthermore on mouseover.
- **Reason of severity:** When a user knows about the behaviour of this functionality, he is not likely to get confused.

Finding 8: Navigating between modes

- **Severity:** Medium
- **Found by participants:** 3, 4
- **Found in scenario:** 1
- **Found in task:** 10

- **Description:** The participants expected to use a button to navigate between edit mode and preview mode instead of using the navigation bar.
- **Reason of severity:** Not knowing how to navigate in an application can slow down a user's workflow. On the other hand, when a user knows about the behaviour of this functionality, he is not likely to get confused.

Finding 9: The invisible font in a language dropdown in preview mode

- **Severity:** Medium
- **Found by participants:** 2, 3
- **Found in scenario:** 1
- **Found in task:** 11
- **Description:** In preview mode, the participants' browser displayed a language dropdown with a white font in front of a white background.
- **Reason of severity:** A user saw an empty dropdown. It is caused by dark mode enabled in the system.
- **Fixed:** The error was fixed, and the font is visible in the dark mode as well as in normal mode.

Finding 10: Collapsing a section

- **Severity:** Medium
- **Found by participants:** 1, 2, 3, 4
- **Found in scenario:** 1
- **Found in task:** 12
- **Description:** The participants were unsure of how to set a question to be collapsed because there is no form control corresponding to it. The confusion was due to the collapsed property being set in the *Layout class* form control.
- **Reason of severity:** This problem relates to the Finding 1. The *Layout class* form control should be separated into form controls, in order to have form control types and its properties separated and named correctly.
- **Fixed:** The wording of the label was changed to *Question type and properties* as a quick workaround.

Finding 11: Exported form source code is not formatted

- **Severity:** Low
- **Found by participants:** 1
- **Found in scenario:** 1
- **Found in task:** 16

- **Description:** After exporting a form, the participant found the downloaded form in JSON-LD format confusing, as the form source code was not formatted.
- **Reason of severity:** This finding does not influence the functioning of the application.
- **Fixed:** This shortcoming was fixed, and the exported form source code is now formatted.

Finding 12: File was exported without a file extension

- **Severity:** Medium
- **Found by participants:** 3
- **Found in scenario:** 1
- **Found in task:** 16
- **Description:** When a participant exported a form by downloading, it did not contain a file extension, and therefore he was not able to import it to the form editor again unless he renamed the file in the system to have a *.json* extension.
- **Reason of severity:** A user was unable to continue without a modification of the downloaded file in the system.
- **Fixed:** An exported file now has a *.json* extension explicitly coded in the source code of the editor.

Finding 13: Change a form type

- **Severity:** Medium
- **Found by participants:** 1, 4
- **Found in scenario:** 2
- **Found in task:** 3
- **Description:** The participants were confused about the fact that a switch form control is used for changing a form type. Moreover, it is not clearly visible, which form type is being chosen. Instead, the participants suggested using a single-choice question or an image choice question for the purpose of choosing a form type. The fact that the form configuration modal closes right after the form type is changed also caused participants confusion.
- **Reason of severity:** The switch form control is not suitable for this type of action. Also, since the editor will be further extended by the possibility to create forms of more different types, the switch form control should be changed to another one.

Finding 14: Navigation between edit mode in user interface and edit mode in source code

- **Severity:** Low
- **Found by participants:** 4

- **Found in scenario:** 2
- **Found in task:** 6
- **Description:** The participant expected that he can switch between edit mode in source code and edit mode in the user interface by clicking one button.

Finding 15: Indication of different edit mode types in the navigation bar

- **Severity:** Low
- **Found by participants:** 4
- **Found in scenario:** 2
- **Found in task:** 6
- **Description:** The participant expected to see an indication of whether he is navigated in source code edit mode or user interface edit mode in the navigation bar. He would appreciate a possibility to navigate between those two edit mode types using the navigation bar.

Finding 16: Text field visualisation with no layout class set

- **Severity:** Low
- **Found by participants:** 4
- **Found in scenario:** 2
- **Found in task:** 10
- **Description:** The participant was confused about the fact that the form editor previews a question with no layout class set as a text field.
- **Reason of severity:** This issue is caused by the fact that if no layout class is set, then the question is treated as a text field in SForms.

Finding 17: Load a form from clipboard

- **Severity:** Severe
- **Found by participants:** 4
- **Found in scenario:** 2
- **Found in task:** Out of scope of the scenario after task 20
- **Description:** The participant tried to load a newly created form by pasting the form source code from a clipboard to the JSON editor, and the application got stuck. After reloading, the application started working.
- **Reason of severity:** This bug causes impossibility of using the application.
- **Fixed:** Loading of a form using the JSON editor error was fixed.

Appendix C

Installation Manual

C.1 Required Software

- **Semantic Form Editor** - see the enclosed CD
- **Node.js version 14** - available for download at `https://nodejs.org/dist/`

C.2 Installation

1. Download and install Node.js
2. Copy the *semantic_form_editor.zip* file from the enclosed CD to your computer
3. Unzip the *semantic_form_editor.zip* file
4. Open the command line and navigate to *semantic_form_editor* folder
5. Type command *npm install* and wait until the process finishes
6. Type command *npm run build* and wait until the process finishes
7. Type command *npm run start*
8. Open the browser at `http://localhost:3000/`
9. Now you can create a new form or import an existing one

Appendix D

The Contents of the Enclosed CD

- **semantic_form_editor.zip** - source code of the application
- **klimato2_2021_mt.pdf** - master thesis report
- **master_thesis_source.zip** - source code of the master thesis report
- **README.txt** - installation manual