University of Turku
Faculty of Technology,
Department of Computing
Vasily Babenko
SN: 515515

Visualization and user interactions in RDF data representation

Turku 2021 University of Turku

Abstract

The spreading of linked data in digital technologies creates the need to develop new approaches to handle this kind of data. The modern trends in the information technology encourage usage of human-friendly interfaces and graphical tools, which helps users to understand the system and speeds up the work processes. In this study my goal is to develop a set of best practices for solving the problem of visualizations and interactions with linked data and to create a working prototype based on this practices. My work is a part of a project developed by Fail-Safe IT Solutions Oy. During the research process I study various existing products that try to solve the problem of human-friendly interactions with linked data, compare them and based on the comparison develop my own approach for solving the problem in the given environment, which satisfies the provided specifications. The key findings of the research can be grouped in two categories. The first category of findings is based on the existing solution examinations and is related to the features I find beneficial to the project. The second category is based on the experience I got during the project development and includes environment-specific and project-related findings.

Keywords:
RDF, linked data, user experience, graphical interfaces, interactions, web interface, JavaScript.

Special thanks:
I would like to thank the Fail-Safe team for amazing working experience, especially my work supervisor Mikko Kortelainen for providing the project and all assistance and my colleague Ronja Kyrölä for taking a huge part in the project development. I also want to thank the working collective of University of Turku, especially my study supervisor Erkki Sutinen and professor Jouni Smed for their influence and help during my research.
I warmly thank Iryna Nesterova and Ekaterina Danilova for all the help in proofreading and correction of the texts. Finally, I want to thank my family and friends for all the support and care I got during the studies.

Table of Contents

1. Introduction

1.1 Background

This thesis work was done by a master degree student in the program of interaction design, in cooperation with Fail-Safe IT Solutions OY, which provided the problem to solve and quality control of the final results. The main goal of the work is to find convenient and user-friendly ways of interacting with specific types and formats of data , creating a suitable software solution for it and exploring new methods of linked data interactions. The topic was chosen based on the previous tasks performed for the Fail-Safe and the overall obscurity of the topic in public media, making it a rather interesting topic for exploration.

1.2 Terms and important abbreviations

- API – Application programming interface
- Backend – part of the software which performs the computations and run on server
- B2B – Business-to-Business
- CLI – Command Line Interface (Terminal/Console)
- CSS – Cascading Style Sheets
- DB – database
- DOM - Document Object Model
- Fail-Safe – (in this format and as a short term) Fail-Safe IT Solutions OY. Could also be mentioned in other abbreviations as F-S or FS
- Frontend – part of the software for user interface and interactions
- GD – Graph Data
- GUI – graphical user interface
- HID – Human Interface Device
- HTML – Hypertext Markup Language
- IDE – Integrated Development Environment
- ID – Identification/Identity/Identifier
- IT – Information Technologies
- JS – JavaScript
- JSON – JavaScript Object Notation
- JSON-LD – JSON for Linked Data
- NPM – Node Package Manager, also Node.js
- OWL – Ontology Web Language
- PK, FK – Primary Key, Foreign Key
- PoC – proof of concept
- PO – project owner.
- RD – Research Design
- R(DB) – relational database
- RDF – Resource Description Framework
- RDI - research, development and innovation
- R&D – Resear ch and Design
- REST – REpresentational State Transfer
- SPARQL – recursive acronym for SPARQL Protocol and RDF Query Language
- SQL – Structured Query Language
- SVG – Scalable Vector Graphics
- UI – user interface
- UML – Unified Modeling Language
- URI – Uniform Resource Identifier
- URL – Uniform Resource Locator

- UX – user experience
- W3C – World Wide Web Consortium
- XML – Extensible Markup Language

1.3 Thesis questions

The work is intended to answer several theoretical questions:

1. What are convenient ways to interact with RDF data from the end-user perspective?
2. What are available tools for implementing a user-friendly interface for RDF data interactions in JavaScript?
3. Which ways of visualizing RDF data can assist user and provide satisfying user experience?

1.4 Problems to solve

Answering the questions will require solving several theoretical and practical problems, such as:

- Theoretical
  - Implementing Research Design in the project.
  - Gathering information about already known ways of interacting with graph data.
  - Comparing the techniques between each other from the perspective of UX; and researching the possible improvements in the current interaction techniques and creating new solutions.
- Practical
  - Testing and comparing various tools for project development.
  - Creating the dynamic visualization solution.
  - Implementing the interaction methods into the solutions; and testing and gathering feedback from the solution for improvements and analysis.

1.5 Outcome assumptions

The theoretical outcome of the project is assumed to be a set of conclusions, based on the researched material, which should combine and create the basic idea of good practices for RDF data visualization and interaction. The practical outcome of the project is expected to be a ready-to-use product for visualizing and interacting with RDF data, matching requirements and use-cases, specified by the project owner.

As a result, the thesis questions should be answered and the outcome should be compared to the initial assumptions, the clear vision of the solved and unsolved problems should be stated followed by the future development plans.

2. Theory

2.1 Research design and design science

This research uses Research Design and Design Science methodology for conducting the studies.

2.1.2 Design science

Design science is a term that describes science methodologies based on systematic design. It is usually opposed to natural science, in the form that natural science is problem oriented and is good for studying the natural world, while design science is solution oriented and was created to study the artificial world [35]. Design science became popular among Information Technologies researches in the form of design science research,

2.1.2 Definition of Research Design

Research design (next RD) is a methodology for research conduction aimed on increasing data collection and processing efficiency following to the answering the research questions and problems. It is very clearly defined in the “Research Design and methods. A process approach”(p.125): Some of the most important decisions that you will make about your research concern its basic design and the setting in which it will be conducted. Research Designs serve one or both of two major functions:

1. Exploratory data collection and analysis (to identify new phenomena and relationships)
2. Hypothesis testing (to check the adequacy of proposed explanations).

In the latter case, it is particularly important to distinguish causal from correlational relationships between variables. The relationship is causal if one variable directly or indirectly influences the other. The relationship is correlational if the two variables simply change values together (covary) and may or may not influence one another. [1]
The research itself can be classified in 4 types:

1. Basic research – the already created theory is evaluated via data collection and analysis.
2. Applied research – new data is generated in order to apply it for the raised problems and questions.
3. Overlap research – two previous types combined.
4. Design science research – research based on design science, solution-oriented investigations based on creating artifacts and applying them to the context.

According to this classifications, this work can be classified as applied research, as we are trying to generate new data in order to solve the thesis questions.

2.1.3 Design Science Research

Design science research is a type of research that implement design science methodology and focuses on development and performance analysis of artifacts, in order to improve these artifacts or adapt them to new solutions[36]. In terms of informational technologies, these artifacts are usually represented as algorithms, interfaces, data structures, protocols or languages.

2.1.4 Types of RD

The RD methodology can be classified into different types according to the methods used and results planned to achieve:

- Correlational – the research is based on collecting data on subject with observations or measurements and environment is set static or not manipulated. The correlational design is dedicated for establishing relations between data and forming conclusions out of it.
- Experimental – the research is based on experiment and the environment is strictly controlled. The environment manipulation is a key in this type of design and it can provide more detailed conclusions on data relations.

- Demonstration – an experimental design, where environment is not strictly controlled. The difference from correlational design the environment is artificially created and can be manipulated with limited factor, the difference from experimental design is the limitations on manipulation.
- Exploratory – an exploratory design is conducted about a research problem when there are few or no earlier studies to refer to. The focus is on gaining insights and familiarity for later investigation or undertaken when problems are in a preliminary stage of investigation.
- Review and Descriptive – design based on collection and recombination of data.

There are more sub-classes, such is a meta-analytics, however they are not connected to the current work and can be left aside.
Based on our view on the subject of research, the research environment and thesis questions, it can be assumed that our research design should be experimental or demonstrative or a combination of both.

2.1.5 The design methodology

Since this project is solution oriented and is based around a real world problem, it comes naturally that most beneficial to the research would be to adapt and implement design science approach.
In this approach we will use an "engineering cycle", consisting of 5 stages:

1. Problem and environment investigation – process of collecting information on the problem, how it is applied on the real-world realities, possible background research, analysis of similar solutions. In the case of this research, the problem would be presented by project owner and analyzed, including the provided environment specifications.
2. Solution design – based on the information gathered in the step 1, the solution would be prototyped and modeled. In this project that would take a form of UI prototyping via wireframes, diagrams and sketches.
3. Design validation. Usually is done via applying design(aka. artifact) to the environment (aka. Context) for evaluation. Can be conducted as simulation, experiments, prototype testing, but also can take form of expert or user evaluation. In our case, the project owner and potential clients are considered as expert and real context.
4. Design implementation – applying the design to create the actual product via development cycle. The development cycle can be any suitable process. In IT it is usually being either form of Agile or traditional waterfall development cycles.
5. Implementation evaluation – evaluating the designed product in real-world context. Usually is done with transferring the product to the test groups or straight to the client and collecting feedback.

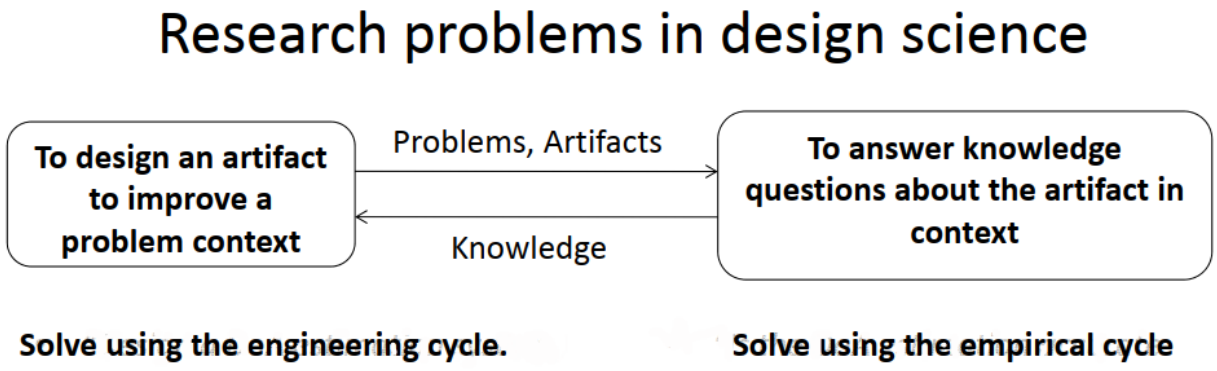


Figure 1: Design science research cycle. [37]

There is a standard research cycle than can be applied to the current research. While the artifact should be designed based on already acquired knowledge of the problem and context, it will raise other questions and problems that need to be solved. For that, design science suggests “empirical cycle”, which is based on explorations and observations and includes these 5 steps:

1. Knowledge acquiring – collecting theory, creating research questions, abstract problem analysis. In this research data collected during this step will be stored in this chapter as “theory”.
2. Research design – as described in previous section, this includes designing procedures and activities to perform correct research.
3. Design validation – usually consists of risk assessment and cost-efficiency balancing, analysis on how much the research will benefit to the project.
4. Research execution – self-explanatory, performing the designed activities and acquiring the data.
5. Result evaluation – answering the research questions, analyzing acquired data, performing corrections in the “engineering cycle”.

2.2 Modern Data models and formats

In the modern world data are becoming one of the most valuable resource. Information systems store and process more data every year exponentially. [2] The need of effective storing and structuring of the data becoming more and more obvious.
This work is focusing around the RDF data model and its representations and serializations. To understand it, the concept of regular databases should be explained and used as example.

2.2.1 Difference between data and data structured

To clarify the semantics and prevent misunderstanding, the term “data structure” should be clearly separated from the term “data”. Under the “data structure” in this work will be meant the actual structure, in which data is stored and handled. This term is close to the term “data model”, described in the next section. What may bring the confusion is that user can work with both data structure and data itself, and these are usually completely different processes. Under working with data we mean data collection/processing, and under working with data structure we mean creating/modifying the structure, in which data will be organized later. The closest example to the difference between these two actions would be setting up and filling a database tables. While defining the tables, naming columns and setting up the relations between them is considered a work on the data structure, since there is no actual data in the tables yet, filling the data into the tables and performing queries to it is considered working with the data itself.

2.2.2 Data model definition

The data model is an abstraction that defines the organization of the data elements and their relations between each other. It is a set of rules, both logical and syntactical, which is used to compose and store data, as usually serves a concrete purpose, such as: storing efficiency, usage efficiency or any other specific requirement. The requirements are specified via modeling, where two main abstract levels are taken: logical modeling and physical modeling [3]. While in Logical level, the structure of data is defined by its future usage, in Physical level it adjust the model to the real environment, such as storing on specific media or optimizing to meet special performance requirements. This can be summarized in a pipe-flow chart, where every operation on the data structure shapes and restricts the final data model [Fig 1].

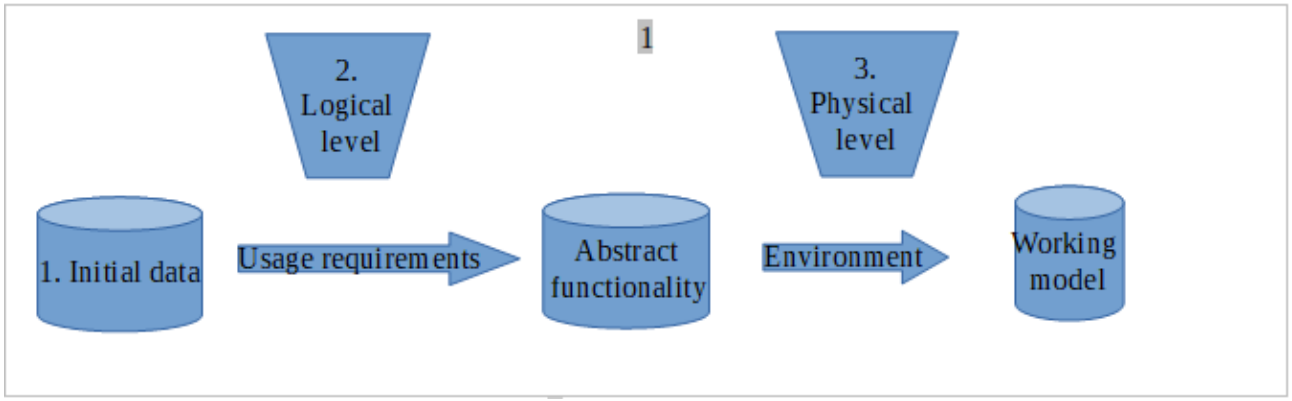
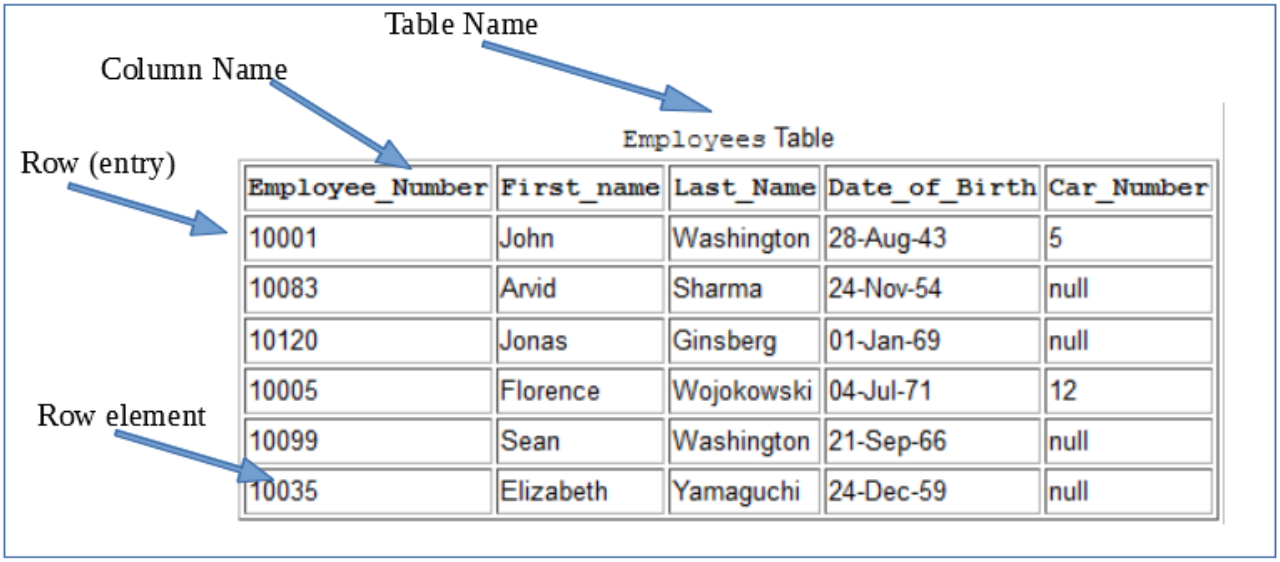


Figure 2: Data modeling flow

Each step, marked by number can be described by a set of questions or problems, solving which will define and shape the model:

1. What data will be used? Media, binary, linked data, text collection, etc.
2. How will the structure operate?
   a) What main operations will be performed on data? E.g. search, sorting, storing, updating, etc.
   b) What form of results is desired? E.g. Speed of access, availability, storage optimization, etc.
   c) Is there any relations between data pieces?
3. How it will be implemented in the system and hardware perspective?
   a) What storage type will be used?
   b) What applications will access data?
   c) How data will be transferred?
   d) For how long data should be stored?
   e) Are there any environment specifications? E.g. Encoding, compression, access rights.

2.2.3 Relational Databases



Employee_Number	First_name	Last_Name	Date_of_Birth	Car_Number
10001	John	Washington	28-Aug-43	5
10083	Arvid	Sharma	24-Nov-54	null
10120	Jonas	Ginsberg	01-Jan-69	null
10005	Florence	Wojokowski	04-Jul-71	12
10099	Sean	Washington	21-Sep-66	null
10035	Elizabeth	Yamaguchi	24-Dec-59	null

Figure 3: RDB table structure. [5]

In this work data models related to the data storing and aggregation, mostly databases (DB), will be described and examined. The main DB function is providing comfortable access and storage to the data that can be linked or sorted by any means.

The most common DB nowadays are relational DB, which, in simplified way of describing it, store data as a collection of tables.[4][32] The term “relational” was used as a description of data organization at the point of developing this data model, however, most of the data model strongly rely on various logical relations between data fragments, which will be shown on the “non-relational” databases further in the work. Not to be confused with this term, “relational” databases implement the technical level of data relations via keys, while “non-relational” DBs do not have this mechanics.
The Relational Database (next RDB) couldcan be described, as sets of data, organized in tables and connected (related) to each other via Primary-Foreign (P-F) Key relations. To clarify the terminology, all data in RDB is distributed and sorted by its meaning, the logical structure, amount of tables, their content and relations are defined on the DB creation by the designer. Each table has a strict logical structure – each row is an entry, a singular data piece, where all data is connected and it is split by columns – each defining the meaning of the element in the entry [Fig.2].
PK, must be unique for this column. That is why usually PK is defined by set of unique, usually incrementing numbers, known as Identification (ID). All other elements of the row in that table are connected to this ID and can be related further. To connect data from one table to another, connecting table should contain a Foreign Key (FK) column, which will contain ID for the PK of the table it is relating to. This allows a flexible, but very structured logical model. [6]

2.2.4 SQL language

Creation of such data model as database goes through full cycle of data model design. On the logical level of design, database is defined, as a collection of various, usually text-formatted data, where data elements are connected and should be easy to access, add, modify and perform various operations on them. Summarizing, efficiency and comfort of working with it is prioritized. To allow any interactions with database, a database engine is designed. In RDB the language family is called SQL – structured query language. It is clear from the name that language primary function is to perform queries to the database with correct structure and predictable results. SQL implementation is a part of Relational Database Management System (RDBMS), which is a complete data model, describing it on every level and defining environment and tools, required for this data model usage. SQL is a declarative language, consisting of statements, and it is used for both database management and data operations. Management function combines creation of the logical structure of database and manipulation of its content, adding, removing and modifying the data. Data operations allows user to access and process the data, fetching data with desired conditions, performing various logical operation and combining multiple sources into one result. Usually SQL offers a procedural functionality, allowing storing and using procedures.
The SQL makes a heavy use of keywords, which represent functions of the language, identifiers – reserved names of logical structure elements, such as tables and columns and logical expressions. From this elements in SQL it is possible to construct a query – a description of data to fetch from DB, or a statement – an instruction, which defines an operation on the database.
In some cases, the SQL expression can be very human-readable and easy to understand. On Fig.3 The Keywords are marked in bold, while identifiers are marked with italic.

```
SELECT Last_Name
FROM Employees
WHERE Car_Number = NULL
```

Figure 4: Example of SQL query based on Fig.2

The query displayed in Fig.3 is readable enough to understand that user wants last names of every person in the table “Employees” who don’t have car number (and as a result a car).

2.2.5 RDB: Pros and Cons

The technology of relational databases was proven to be versatile and reliable and had found its place in countless amount of software projects all over the world.[7] After years of successful implementations of RDBs specialists are able to define strong and weak point of this technology:
Strong sides:

- Logical level:
  - Data and queries are human-readable and easy to understand
  - Strict structure – data is easy to navigate
  - Relations are clearly defined
  - Standards are clear
- Physical level:
  -
  - Supported in most of the environments and has a very long history of practice
  - Good stability and performance
  - Plentiful amount of various features from access control and security to tuning for specific tasks
  - Easy maintenance and troubleshooting

Weak sides:

- Logical level:
  - Structure limits the size and types of data
  - Hard structure decreases flexibility
- Physical level:
  - High cost of setting up and maintenance
  - Interface complexity
  - Not well reacting on scaling

2.2.6 NoSQL and Non-relational databases

While RDB are vastly popular and widely used, from the start of new millennium other types of database started to spread, more specialized and aimed to solve regular RDB problems.
While RDB is a family of databases, which all utilize more or less the same idea and mechanics, under Non-relational DB comes everything that was designed aside RDB philosophy and does not follow its ideology. There are a variety of different database structures, which are not very similar to each other. Therefore, it is best to focus on the ones that are relevant to the topic.
NoSQL DBs are usually less structured and provide more flexibility for implementation, allowing to work with more data formats or are designed to fit a very specific use-cases. They are more suitable for storing media formats or when data input is not structured and has no strict types. [8] Another advantages of NoSQL can cost less, as they are usually open-source and cost-effective and scale way better than RDB.
The most common examples of NoSQL databases are: Document DB, Object DB, Key-Value stores and Graph DB. In this work the most attention will be given to the graph databases, their types and usage.

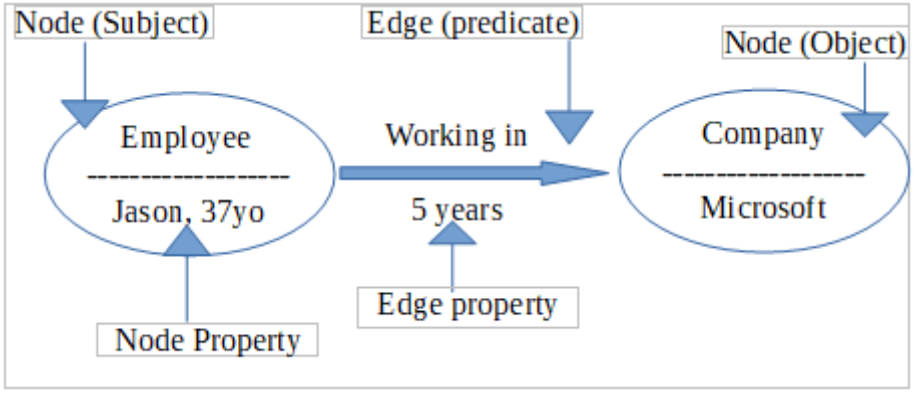


Figure 5: Simple example of a graph with additional correlations to RDF notation.

2.2.7 Graph Databases

Graph Databases implement a completely new logic of data organization: data in these databases can be represented as graphs, not as tables. Every piece of data in such representation takes form of node, edge or property [Fig 4.], so this structure allow a graph math and logic utilization. The key and atomic element of any graph DB is a graph – two data pieces and their relation between each other – an edge.

Note that edge cannot point to nowhere, but can point on the same node. Property – data, which was assigned to the graph node. However, in most notations, the properties of a node can be represented as additional graph connections. The graph by itself can be undirected (edges have no direction), directed (relation can be only in one way), and additionally it could be weighted – the edges can have numerical value. As seen on Fig.4, the “5 years” property can be used to make the graph weighted, but then all other edges should have some weight. This would work if the graph would represent some time relations between people and companies, but this might not be suitable for every situation, and solely depends on the use-case of the graph.

The graph notations allow representation of almost any information as a combination of graph relations, after which graph logic and math are applied, allowing complex processing of data.[32]

The graph representation of data makes it very flexible and scalable, the well-developed graph theory provides needed math and is supported by specialized hardware, allowing big data computing and analysis in much better way that can be achieved with relational databases.

However, graph databases usually cannot provide the compactness and user-friendliness of RDB, usually utilizing complex technologies on both implementation and operation, making the transition from RDB usage to graph DB an inefficient and nontrivial process.[9]

Many graph databases use unweighted directed graphs. The variety of graph database implementations includes technologies such as Neo4j, Oracle RDF and Apache Giraph, working in various environments and utilizing different query languages like Gremlin, SPARQL, and Cypher

In this work, the main focus is on graph databases utilizing RDF (Resource Description Framework) format – widespread specifications popular in web technologies and using SPARQL language for operations, which was designed to be SQL-like query language for RDF data manipulation. [33]

2.3 RDF in details

Initially developed for metadata data modeling and published by W3C, RDF specifications were later used for much broader data models and descriptions. The actual usage of RDF comes from its name – Resource Description Framework. It is commonly used to describe (web) resources, which in terms of Web can be anything. By describing relations between hyperlinks, data entities and

resource addresses it is possible to create a complete description of a service or a system. The current version of the specifications provided by W3C is 1.1, published in 2014, ten years after initial 1.0 release.
The RDF data model uses triplets to describe relations between resources. An example of such a triplet consisting of subject-predicate-object can be seen on Fig.4. However, the initial data entity RDF is describing is a web resource – anything, that can be identified with URI (Uniform Resource Identifier). Very common practice is to use URL (Uniform Resource Locator also known as web-address) as a URI because URL is a locator of the resource. They are commonly mixed together, however URI can be represented by any unique identifier in the system, such as name or serial number. In web resource description the simplest way to identify the resource by its location, which is unique and provide useful hierarchical information. The RDF storage is called RDF triplestore, as all data is stored in triplets. In graph perspective, a node can be represented in RDF by 3 data entities: a URI, which can be used for both subject and object in any combination, a literal – unicode string, representation of a piece of data, which can be only object (this implements a property-like functionality) and a blank node, which has no URI and can be either a subject or object, but not both at the same time. The functionality of blank (anonymous) nodes is used for simply identifying an existence of resource. RDF is utilized by various graph databases, a famous example of which can be Oracle database solutions.
The operation language of such databases is usually SPARQL (a recursive acronym of SPARQL Protocol and RDF Query Language) – standardized by W3C as a default RDF query language. It is also based on keywords, however despite SQL it is not that human readable. It uses some of the common SQL keywords, such as SELECT, WHERE and others, but because of completely different logic, the statement of the query has completely different structure. There is no need for SPARQL to explain from which tables to extract the data, but instead, it can specify the default graphs to work with. While in RDB the namesets of all columns and tables are registered in RDBMS, NoSQL require some other way around it. SPARQL need an ontology, which will provide naming and other data model descriptions. It is usually set via PREFIX keyword. SQL has a heavy use of NULL, a description of non-existent data, equivalent of empty cell in the table. RDF is not utilizing this principle, but can utilize NOT EXISTS for such case, as well as for non-equal logical statements.

2.3.1 RDF Advantages and disadvantages

- Advantages:
  - Pure graph representation
  - Stores data in compact triplets
  - SPARQL is a functionally full language
  - Data processing can be parallelized
  - Very scalable
  - Various serialization formats available for different purposes
- Disadvantages
  - Not as intuitive and human friendly as other DB
  - SPARQL is not easy to operate
  - The excessive amount of serialization formats, ontologies and identifiers can lead to an unneeded complexity
  - The web resources systems are usually dynamic environments. RDF does not provide good solution for reacting to changes

2.3.2 Serialization formats

Because RDF is a graph based data model, it has its own rules and syntax. However, there are several ways of writing it, which are called serialization formats. They drastically differ from each

other, allowing to represent the same information in completely different way, suiting for different use-cases. The most common formats are:

- Turtle
- N-Triples
- RDFa
- TriG
- N-Quads
- JSON-LD
- RDF/XML
- RDF/JSON

This work is mostly focused on JSON-LD, RDF/XML and Turtle, as they are most used in practical implementation. They all utilize different logic and are based on different technologies, partially inheriting their properties.
We will use a graph [Fig. 5] as an example for all serialization representations:

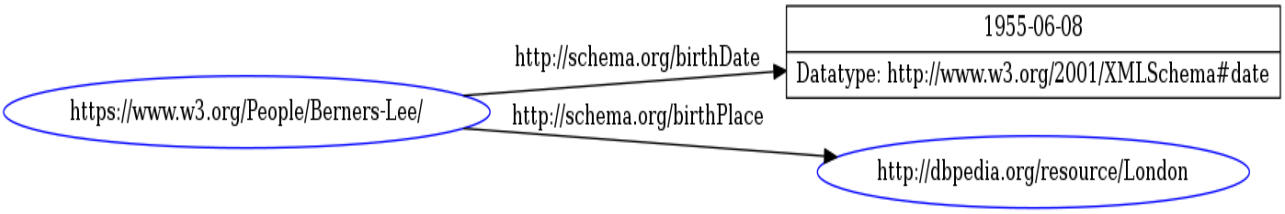


Figure 6: A simple RDF data in a graph format. [10]

2.3.2.1 N-Triples

The most simple for the perspective of RDF ideology is N-Triples, which is almost a plaintext representation of the actual data, where every triplet is written as it is, in a format of “subject, predicate, object“. While being very straight-forwarded, this approach is neither compact, nor user-friendly, and is difficult to use in the work.

```
Triple 1:
<https://www.w3.org/People/Berners-Lee/> <http://schema.org/birthDate> "1955-06-
08"^^<http://www.w3.org/2001/XMLSchema#date>.
Triple 2:
<https://www.w3.org/People/Berners-Lee/> <http://schema.org/birthPlace>
<http://dbpedia.org/resource/London>.
```

Figure 7: Data of Fig.5 represented with N-Triple serialization. [11]

2.3.2.2 Turtle

One of the most popular formats, Turtle, is a data expression method, which is very close to SPARQL language itself and data in this format is resembling SPARQL syntax and query patterns. It takes over the N-Triples by providing methods of grouping triples and compacting them via reducing common parts of URI into prefixes. It makes a group of several triples readable, reduce the repetitiveness and adds logical hierarchy to the data representation. Turtle is considered a human-readable and more convenient to modify by person than RDF/XML [Fig.7].

```
@prefix tim: <https://www.w3.org/People/Berners-Lee/>.
@prefix schema: <http://schema.org/>.
@prefix dbpedia: <http://dbpedia.org/resource/>.

<tim> schema:birthDate "1955-06-08"^^<http://www.w3.org/2001/XMLSchema#date>.
<tim> schema:birthPlace <dbpedia:London>.
```

Figure 8: Data of Fig.5 represented with Turtle serialization.

2.3.2.3 RDF/XML

The Extensible Markup Language (XML) is a language that was created to markup documents, so they would be readable by both humans and machines with comparable ease. The specifications are defined by W3C. It was meant to be universal and relatively simple, so it utilizes Unicode encoding and markup structure, which can remind viewer of HTML. The universality of the language went even further than it was expected, as now it is used for structuring not only documents, but other data structures as well, and is widely used in web services. However, even though the language is supposed to be simple and human-readable, it is not close to its ideals and is often described as redundant and complex.

Based on XML syntax, RDF/XML was created. It utilizes XML structure and several standardized notations to describe RDF graphs in XML format. It could be particularly useful, if initial system was designed to use XML data format or other sources of XML data are present in the system, so the parsing and processing can reuse some parts of project's code.

It is rather easy to see the differences and drawbacks of this serialization type on following example:

```
<?xml version="1.0"?>

<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:schema="http://schema.org/">

<rdf:Description rdf:about="https://www.w3.org/People/Berners-Lee/">
        <schema:birthDate>1966-06-08</schema:birthDate>
        <schema:birthPlace rdf:resource="http://dbpedia.org/resource/London"/>
</rdf:Description>

</rdf:RDF>
```

Figure 9: Data of Fig.5 represented with RDF/XML serialization.

As it can be seen on Fig.8, the language makes serialization excessively verbose, which is the result of mixing language that was created for structuring tree-like hierarchical documents with triple-based graph data, which loses all initial structure.

2.3.2.4 JSON-LD

One of the latest standards adopted for the RDF is based on JavaScript Object Notation (JSON). It is clear from the name that JSON was developed for supporting JavaScript, which makes it particularly valuable in the case of current work.

JSON is a language-independent data format, which is using human-readable structure to store data in form of objects, which contain attribute-value pairs and arrays. It was created to be universal and light-weight, serving multiple different purposes and becoming especially popular in the web

services. JavaScript supports this format natively, making processing JSON data an easy and efficient task.
Basically, the JSON document is a plaintext document, which is structured in a form of object – an non-ordered collection of key-value pairs, where each key (also known as “name”) is a unique string, and value can be anything serializable, the most common are: strings, numbers, booleans and empty values, represented by “null”. It also allows nesting, so any object can include any number of other objects or arrays with any serializable content, including objects. In fact, arrays are also represented as a type of object with hidden naming for which integer increment is used. This allows using JSON for storing almost any data in short and ready-to-use format.

```
{
 "@context": {
  "dbpedia": "http://dbpedia.org/resource/",
  "schema": "http://schema.org/"
 },
 "@id": "https://www.w3.org/People/Berners-Lee/",
 "schema:birthDate": "1955-06-08",
 "schema:birthPlace": {
        "@id": "dbpedia:London"
 }
}
```

Figure 10: Data of Fig.5 represented with JSON-LD serialization.

The simplicity of the format made storing of some data types a bit more challenging, so it was expanded a bit into the JSON-LD (for Linked Data). The goal was to achieve the simplest way of converting both linked data and JSON data in one format, which was done via “context” links to ontologies and reserving few names for special linked data fields. All reserved names are starting with “@” symbol, and are easy to notice. The JSON-LD format was initially developed by separate community, before it was transferred to major RDF group and approved by W3C.
As can be seen on Fig.9, JSON-LD combines hierarchical structure, similar to XML/RDF while keeping the simplicity close to Turtle. These properties along with its native usage in JS makes it the most preferable format in this work’s project.

2.3.3 Ontologies

The W3C gives another name for ontology – vocabulary [12]. In simple terms it is a collection of information, which describes concepts, patterns and relations inside specific area. W3C also implies that currently it is more common to use word “ontology” for it and that elements in such collection can be referred as “terms”.
While the idea and first implementations of ontologies were done far before digital era, as a helpful tools to organize information ontology studies and implementations in Computer Science gave it a completely new wide area of implementation, expanding the idea to a new level.
Ontology is necessary for data unification, simplification and integrity. It helps to organize data and provides means of integration of different datasets into each other. Usually, when two different linked-datasets are combined, it might cause troubles due to various naming rules, syntax, etc. Ontology allows to define common relations in both datasets and refer them to various names commonly used around. On a simple abstract level, this means that if two things in both datasets means the same, but named differently, the mismatch will be resolved, as ontology will tell that these are the same.
Ontology also can expand the data providing unified and common additions to matching patterns.

The ontology technology is widely used among Linked Data technologies, it is operated by RDF group, OWL (Web Ontology Language) and others.

2.4 RDF visualizations and interactions

The graph data itself is stored in a very human-unfriendly manner. However, graphs are well known to humanity for long time and graphical representation of graph data should be comprehensive. The interactions with graph databases is usually performed via coding or SPARQL, which is not user-friendly as well. This work will try to solve the problem of graph data interactions simplification based on concrete practical example.

2.4.1 The importance of user interface and user experience

In past 30 years IT environment has evolved drastically, from machines operating only via electric bulbs to a super-ergonomic devices with sensors, allowing us to interact with them in huge variety of ways, including touch, voice, motions and other different HID. The main reason for it was that the more intuitive and user-friendly the system is, the more people can learn and use it, the faster and more productive the work becomes.

This brings us to the point that any new tools and data models coming to the broad use should have well-designed interfaces and interaction models. Well-designed interface can speed up and improve worker's performance significantly while intuitive mechanics of interactions can decrease the time of learning the new system. Overall, a well-designed UI/UX can boost any business process, which is why many companies nowadays invest in in interaction design in their RDI.

2.4.2 Graphs in daily use

Graph charts are now widely used in mind maps and other representational methods, to present data more clearly for the users [Fig. 10].

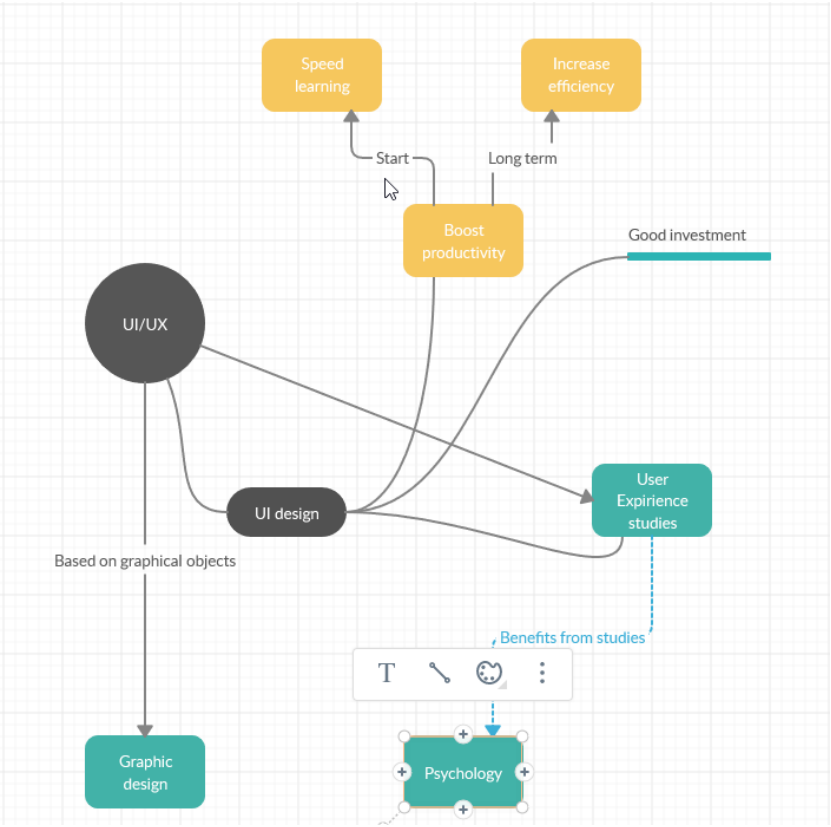


Figure 11: Simplified example of graph mind map [13].

Data in simplified graph format is proven to be more human-readable and provides better picture overview. Another use of graphs is various modeling notations in planning. Widely-used UML notations utilize various graph structures [Fig. 11] to represent relations and hierarchy inside the described model [14].

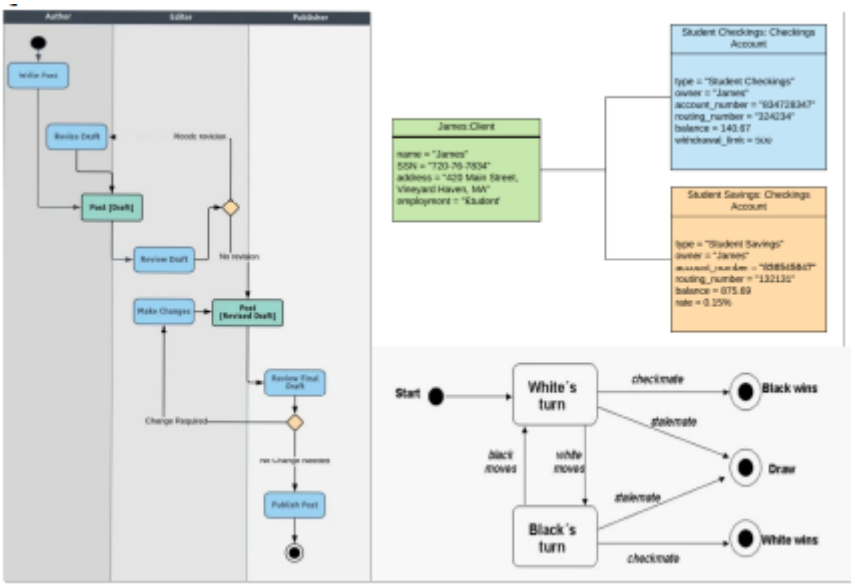


Figure 12: Various examples of graphs implemented in UML notations, including simple directed graphs (Activity and State machine diagrams) and object graph (Object diagram) [15].

2.4.3 RDF visualization

RDF data, being a graph data, is relatively easy to represent in the graph form. There are several services in the Web, which allow a user to convert data in most popular RDF formats into a visual graph [10]. However, these visualizations are mostly static and cannot provide any interactivity. It is still possible to use such services for fast graphic representations of data, creating overviews, even dynamic models, but this would require a lot of tinkering and the whole process is less likely to be optimized.

However, there are ways to recreate a dynamic and interactive environment for provided graph data, which would be based on graphical engines or simple versions of them.

A graphical engine (similarly to game engines) or a graphical library isa set of functions and procedures which provide access to creation of computer graphics. Usually it allows rendering visual data for selected environment using various programming languages. A good graphical engine uses optimized code for efficient software or hardware graphics processing, which allows end user to create graphics without going into the complex math or hardware optimization, speeding up the graphics development and rendering process. Depending on the library, it can be very purpose-specific or implement a wide range of functions allowing complex and universal usage. Graphic libraries are used in GUIs, starting from OS GUI and including any GUI-rich software, videogames, media-editors and model combiners. The most known graphical libraries are DirecX (Used in MicroSoft™ operating systems), Mesa 3D (open-source for OpenGL), Qt (lightweight software framework).

A graphical library can be built on top of another library to provide broader or specific functionality, including support for different operation languages, data formats or various environment-specific optimizations.

The term Graphical Engine (later GE) in this work will be mostly referred for graphical libraries and frameworks, which provide a specific functionality to the user. It should have a clear interface to the most functions, a broad functionality for creating graphical objects in a simplified and more

user-friendly way, provide dynamic rendering of the objects and handle user interactions. So, in short: GE is a graphical library with high automation and user-friendliness.

2.4.4 Interactions with visualized data

The wide use of visualized graph data can indicate that there are common practices of interactions with such data. In fact, the most intuitive ways for interacting with graph data would be physical manipulations with the visual representations via dragging, scaling, rotating and editing the graphs. The other ways of interactions include creation of interactive representations of graph data, such as interactive tables or scripting interactions in some “wizard”-like interactions.

2.5 Testing

It is difficult to overestimate the role of testing in the Software Development, as it defines the success of the final product. There are multiple different types of testing, mostly used for finding and location errors in the product for further mitigation of them. The testing can be very resource-intensive process and it is a crucial task for management to balance the costs of testing with risks they allow to mitigate and the amount of testing itself. The ideal situation of 100% testing (or even 100% code coverage) is usually unreachable and is not a goal worth reaching. Every testing practice should balance the cost-efficiency based on the importance of the tests. This means that effective testing should be precise and should work with things that are the most important in the project.
When the software is tested by the developer, it is usually expected to be a “white box” testing, when the program code is known and the testing is performed using this knowledge and access to code itself. It usually takes the form of unit testing – small sub-programs that run parts of the program’s code with predictable inputs and expected results. If the code returns results matching with the expected ones, it passes tests and usually is allowed further in the production pipeline. The amount of code tested by this is called “code coverage” and its value depends on the type of the project, resources dedicated to such testing, including time limits, and overall management. An important factor is that unit testing tests well algorithms and logic, but is rather ineffective in UI testing. There are technologies created in order to simplify and automate the process of UI unit testing, but they are usually platform specific and still provide rather low outcome.
Another way of testing is a “black box” testing, when the code is not manipulated directly and is presumed to be obscure. This is usually a classic UI testing, when tester takes a role of an ordinary user and tries to find weak or broken spots in the program, which may cause serious troubles if encountered by end-user. This type of testing also can be automated, but it takes more complex approach and complicated software. Usually such testing is done by specialists, both because of skill-intensive reasons and the fact that developer is too used to the program, so some “blind spots” are nearly guaranteed to happen.
Another area of testing is a “Usability testing”. It is an end-of-the-line type of testing, when most of the errors are solved and the main point of interest is to make solution more pleasant, simple, user-friendly, optimizing the user experience. This testing is performed both by professionals and by mass user testing, with collection of feedback and accumulating statistics. Usually, it is done with first releases, or as they are called “open/closed beta-testing” and based on the results gathered the next iteration of the development is planned. Because this is an attempt to measure empirical values, a lot of inaccuracy should be managed, especially when trying to create a unified statistics and summarize responses mathematically. There are several common practices in the usability testing, a vast amount of predefined survey templates of different complexity, each implementing its own way of calculating the results. Many of them can be found on-line, even providing service for data collection [30]. However, blindly taking the questionnaires and throwing them into the use is not always a wise practice, usually they could be adapted to the environment, if not specifically stated.

When considering the ways of usability testing with unprepared subjects, there are several ways of approaching the testing. The most obvious one is to give the user access to the testing environment, allow to roam aimlessly around it (sometimes recording the process with various equipment) and then provide a survey to collect any feedback generated. Another way is to instruct user to perform a sequence of actions – scenario, and after completing it fill out the survey. The second method allows to scope testing to the most important points, with possibility to survey more precise questions, but it does not give a broad view on the situation itself.

2.6 Theoretical assumptions of the practical outcome

Based on theoretical data gathered, we can assume that solving the interaction design problem for RDF data should take a different approach compared to the usual data models. The toolkit and methodology of practical problem solving always depends on three factors: the environment, the requirements and personal experience and expertise of the developer.[34]
In practical part we will define these factors and explore the variety of instruments available, compare them, perform analysis and choose accordingly to the situation.
As a result of the practical part we assume to have a sufficient toolkit for problem solving followed by practical implementation of the solution.

3. Practice

I chose one of Fail-Safe R&D projects as a practical example and source of the research material.The project is intended to solve the problem of providing a visual user-friendly interface to the RDF data stored and aggregated with other company products.

Fail-Safe (Short to the “Fail-Safe IT Solutions Oy”) is a Finnish company working in the IT market in Business-to-Business (B2B) relations and providing Record Management, Compliance, Data Integrity, Information Security, High Availability and Service Continuity solutions to the medium and large businesses. Fail-Safe is specializing on Big Data manipulations, developing solutions for record data management and providing integration, analytic and consultancy services.

3.1 Existing solutions

In order to obtain a better insight into the possible problem solutions and outcomes, it is useful to first research already implemented solutions, which may provide useful data on effective and ineffective practices. During this research, I was performing a search in the Internet and articles with related topics for projects that were developed with purposes, similar to the project described in this work. Numerous instances of software were found and several projects were selected for examination. Selection was based on availability (should be open-source or free to use/trial), similar functionality (the project should focus on visualizing linked data in form of graphs or networks), stage of development and popularity of the project.

3.1.1 WebVOWL

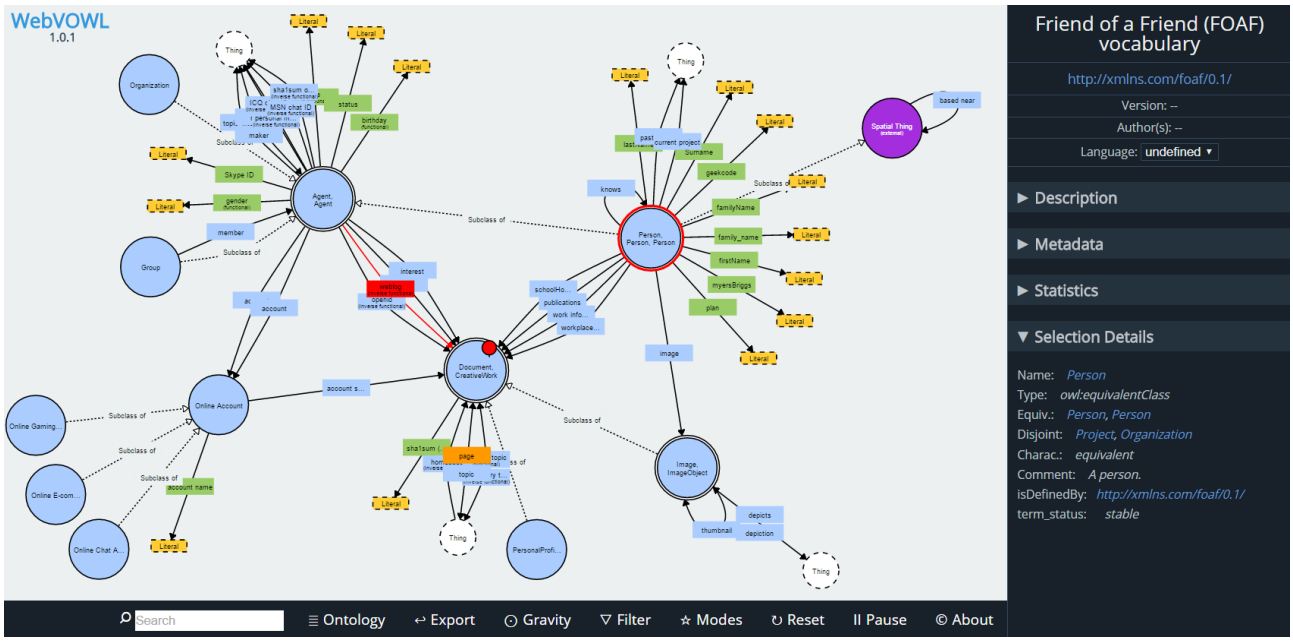


Figure 13: WebVOWL demonstration based on FOAF vocabulary

WebVOWL is a web application for the interactive visualization of ontologies [16], based on the Web Ontology Language (OWL). This already puts it a bit aside from the main scope, as ontology is quite specific type of data, yet it is still linked data used for describing various things. OWL is part of the W3C’s Semantic Web technology stack, which includes RDF, RDFS, SPARQL, etc. [17]Very well implemented UI/UX part is one of the main noticeable features of this product.

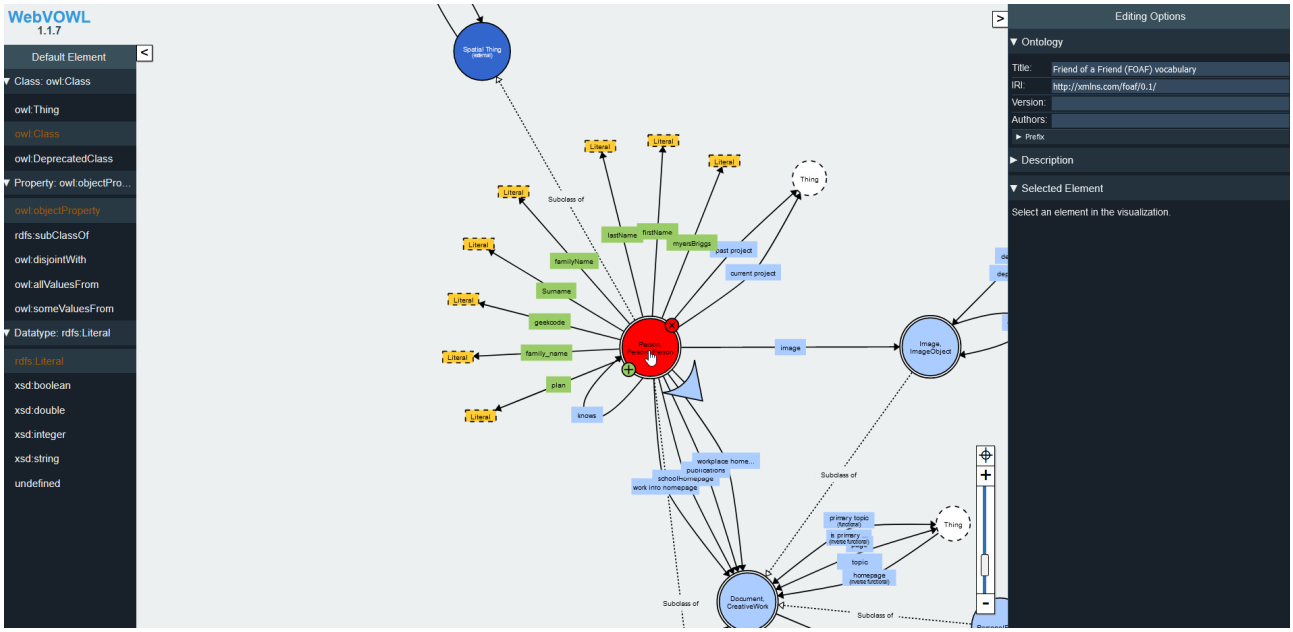


Figure 14: Advanced WebVOWL editing interface (currently in BETA)

I inspected several aspects of the interface and took them into account for future development of own solution:

- The representation of the linked data as a clustered and hierarchical graphs, with possibilities of the graphical elements differentiation, looks readable in both overall and particular examination.
- The graph representation applies live “physics” simulation, allowing graph to expand and arrange itself automatically.
- Every piece of information related to an element is visually connected to it and affects the size and shape of the element.
- Several notation techniques are implemented to distinguish various elements of the graph from one another
- The working area is responsive to the gestures, including zoom, drag&drop (both as a navigation and as a graph reorganization). The user is also able to control the view with additional controls, like zoom bar and control buttons.
- The graph editing is available as a “beta” feature, implying that it is still in development and may contain errors. However, even in current state, most of the basic data manipulations performed were intuitive and functional.
- Possibility to add and remove nodes to the graph, add nodes connected to selected node, add and remove connections between nodes, basically providing a full set of tools needed for editing the graph.
- Additional information and fields for editing are presented on both sides of the working area, which is comfortable, but limits the size of the working area.
- Additional features are provided at the bottom toolbar, with bottom-up directed dropdown panels, which is less intuitive and may work better in reversed way.
- The interactions implement additional color differentiation of the selected element and some temporary drawn additional UI elements, which are hidden, when not in use.
- The process is reversible, and sometimes allows to see a preview of several actions before they are performed.

3.1.2 LodLive

The LodLive project looked promising, as several other projects listed it as a source for development, but despite that, the amount of information available on the project is frustratingly low. There is no comprehensive documentation on the code available. Moreover, the project is almost not at all translated to English, the source code being commented in Italian, which obstructs the exploration of the project further.

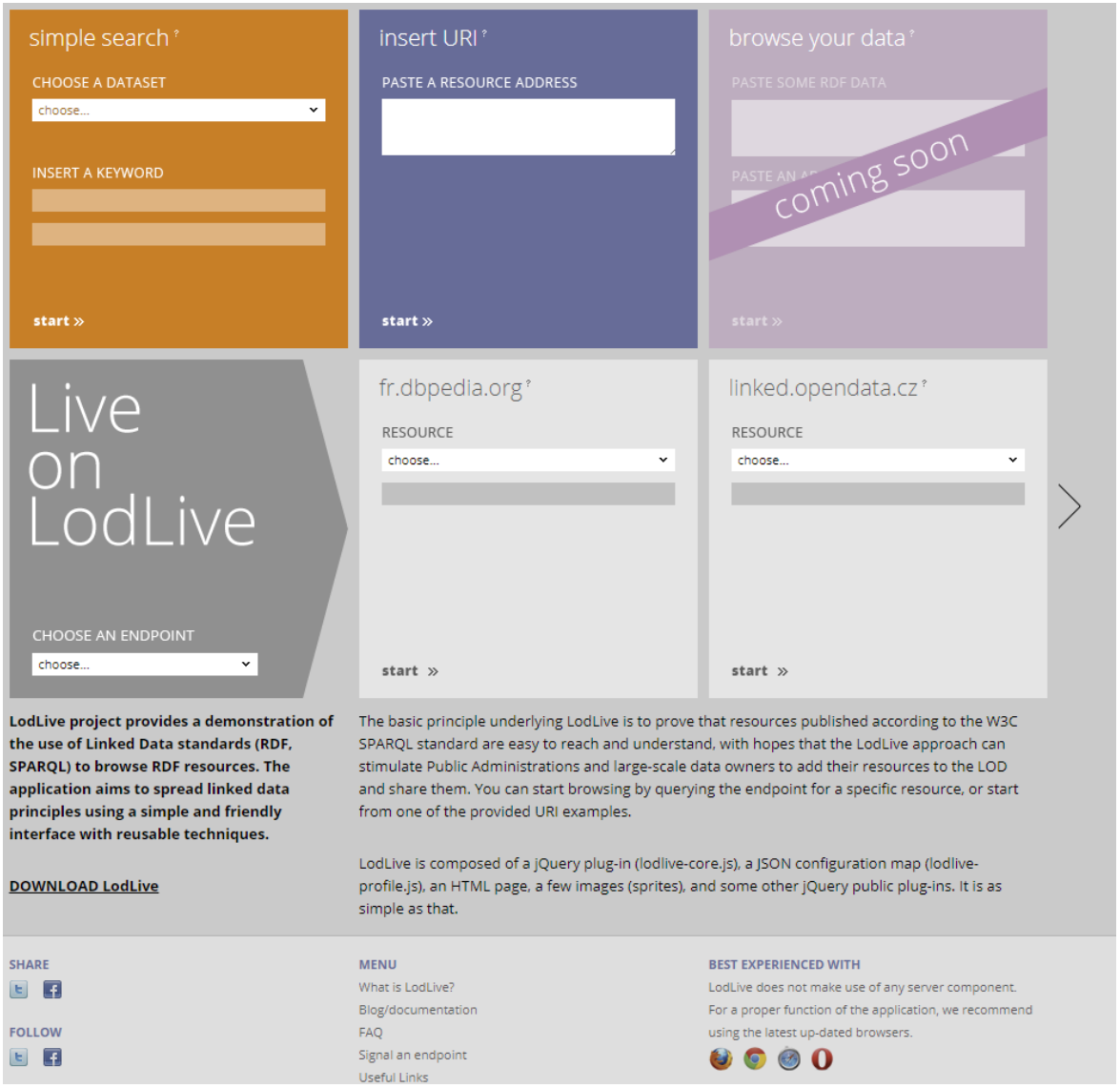


Figure 15: LodLive webapp starting screen

In addition to that, the project is being developed at a very slow pace, with most of the branches abandoned for 6-7 years and the main branch receiving minor updates every half-year, which brings us to the conclusion that the project is close to be considered as abandoned. [18]
However, I used the demo version of the project in local environment, explored it and was able to compare it to other existing solutions. The demo project uses examples mostly based on European Central Bank Exchange Rates Common Procurement Vocabulary [19], which made the testing difficult. The possibility to browse user-provided data is promised to be available in the future, however the project is not updating and chances to see this feature released are neglectable.
I was able to test the prototype by exploring the examples related to French DBPedia. However, that provided me with enough information to compare it to WebOAL and make conclusions for the development of own project.

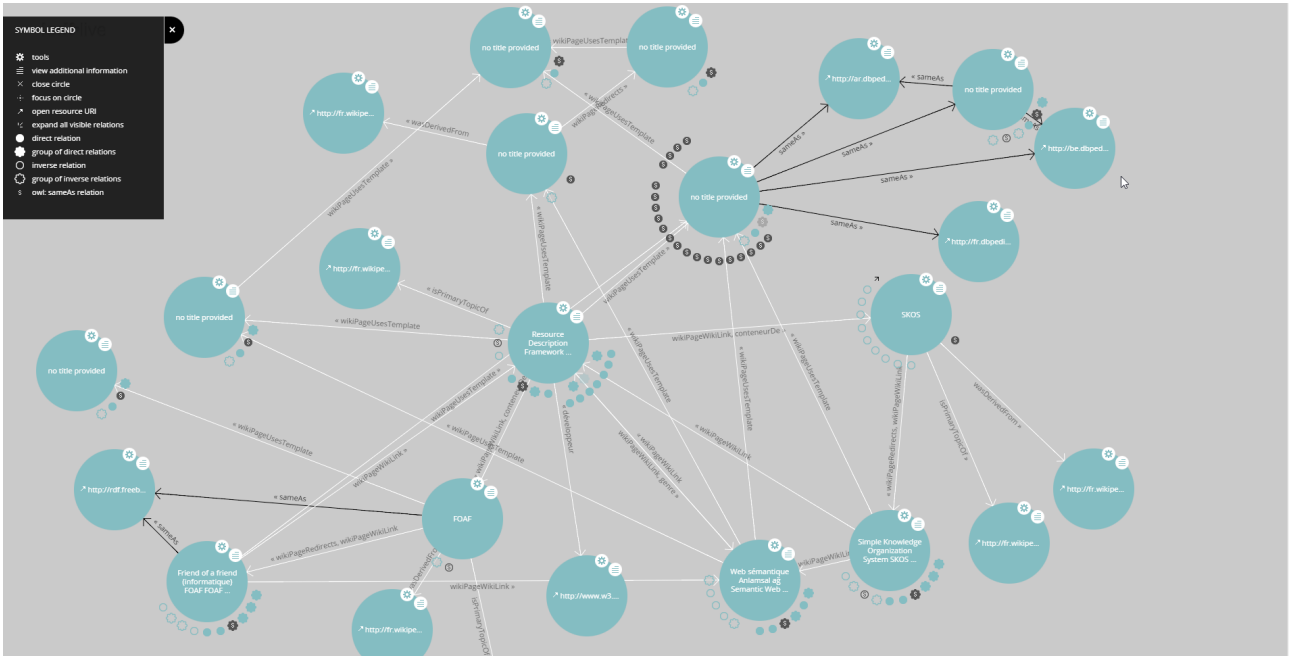


Figure 16: LodLive basic working example with symbol description

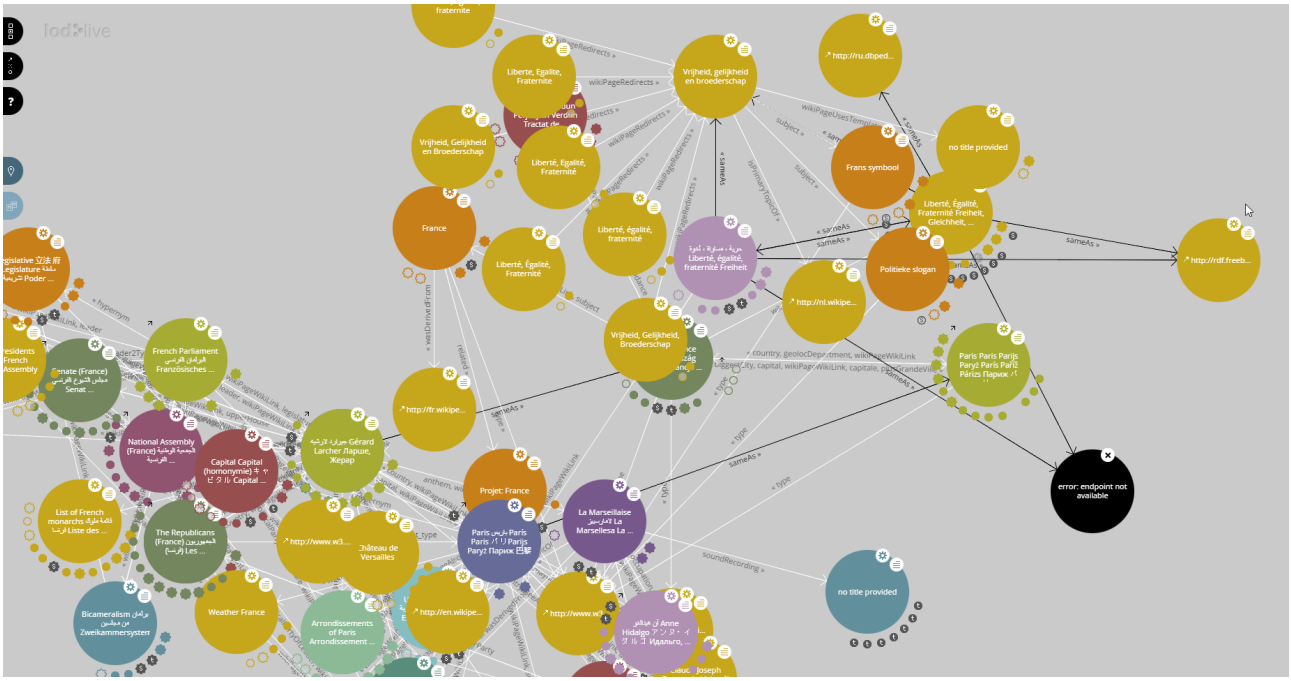


Figure 17: Example of workspace overloading

Results of the investigations were collected in the form of a table [Table 1]:

Points of interest	**WebVOWL**	**LodLive**	**Conclusion**
Initial view, starting layout	Fully expanded. All nodes are present on the screen and can be collapsed by user.	Compacted to starting nodes. Nodes can be expanded by user.	While a compacted view allows the user to choose what to examine and thus more control, an expanded view allows immediate impression on the situation. For controllable-

			sized graphs, the expanded view seems to be more preferable.
Node appearance	Different colors, shapes, sizes.	Different colors of thesame size and shape.	More differentiation may require more intellectual data parsing, but offers a higher quality of UX.
View control	Drag-n-drop, zoom, slider controls.	Scroll.	LodLive renders the graph only as a scrollable picture. The UX doesn’t feel good with such a solution and it should be avoided.
Interactions	Choosing the action from side toolbars, hover and selection affect colors. Some additional UI elements (arrows, new element creation buttons) appear if matching the context of user actions.	All interactions are drawn near node with symbolic notation. No additional UI on the sides.	While keeping borders of the screen clean extends the working area, this may create an overpopulated central area.
Help and tips	While easy to learn, it provides very little information on how to use.	Has legend, help panel and provides useful tips on how to use.	Any system of such type should contain at least some degree of tutorial and help for new users.
Additional information	Displayed in large collapsible sidebars that take considerable amount of view area.	Displayed as appearing and dismissable pop-ups.	Pop-ups feel more natural and helps to keep the working area clean and wide.
Elements arrangement	All nodes strictly follow the physics simulation rules, arranging themselves according to each other, trying to self-organize in logically correct and readable formation. A user can rearrange elements, but they still will be rearranged after the user is done.	No arrangement is applied, nodes are created near the source, overlap each other. Edges cross, overlaps and undergoes nodes. A user can arrange nodes freely, they stay on place.	While strict auto-orientation clearly limits user-based scenarios and usability, physics simulation is definitely a useful tool. Nodes should group logically and must not overlap with each other. At least a degree of user-based placement should be allowed, by making simulation less strict.

Table 1: Comparing WebOWL and LodLive

3.1.3 Cytoscape

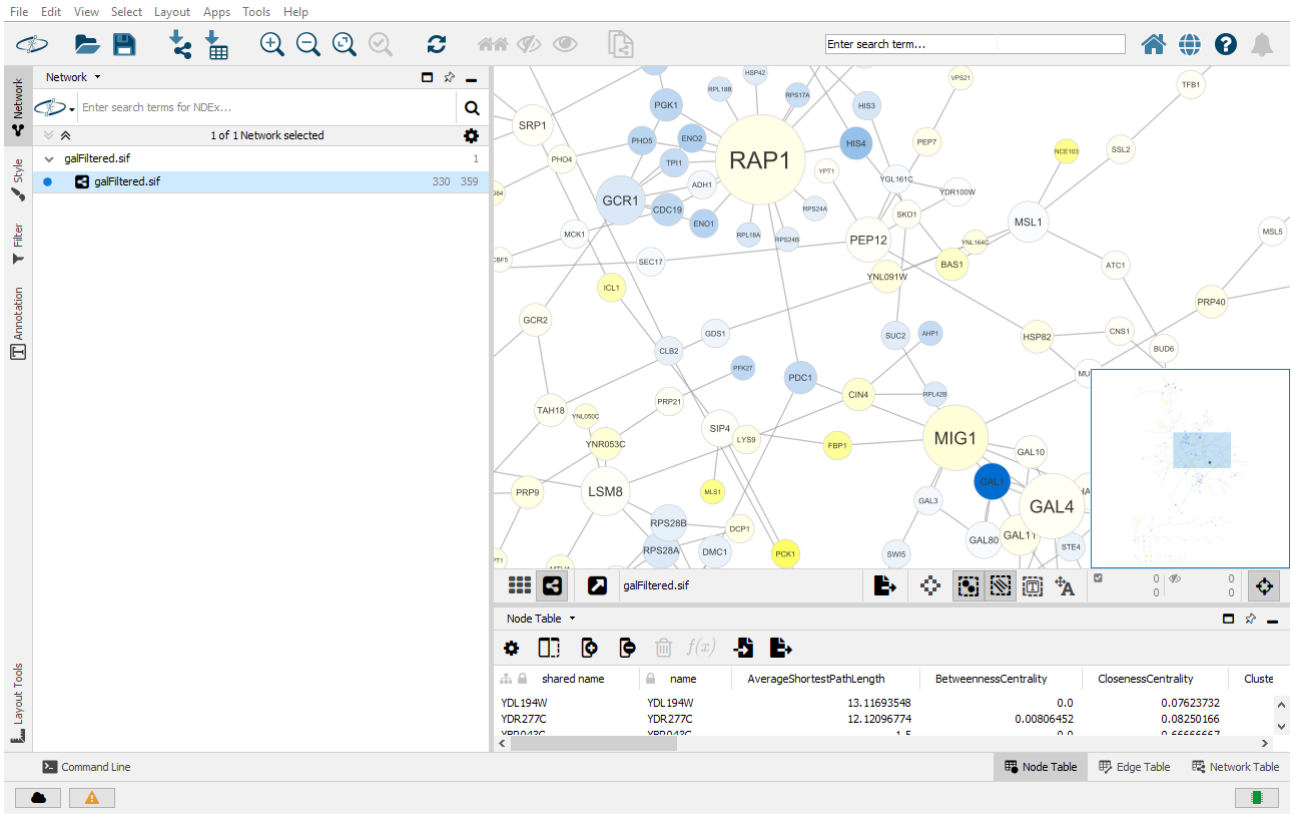


Figure 18: Example project "Yeast Perturbation" opened in the Cytoscape 3.8.2

Cytoscape is a solution created for visualizing complex data used in natural sciences (chemistry and biology), and intended to be free and open source. The code of this project is used in various other projects as well. [20] The project is rapidly developing, the current major version at the moment of this document creation is 3. It has a well-maintained documentation and knowledge base, with a large supporting community.

The software is a well-developed solution for working with science-specific types of data. It is professional-oriented and requires skill and training to use, because for a new user it may look over-complicated. As a result of the exploration research, examining the software several points were taken:

- The program has a long learning curve, during which UX improves a lot.
- To aid the learning there is a very detailed manual and several tutorials, which are opened in the embedded browser window.
- Additional to the tutorials, there are animated hints in the form of short tutorial animations, which pop-up when user hover the mouse above the button. Based on reviews and comments available in project-related forums and community places, this method has a potential to be very effective.
- The working area is split into several sections, where the graph takes the biggest one, but still, even with other sections minimized in size, it takes 70%-75% with all panels present.
- Tool panels can be closed, resized and reordered to improve UX.
- The graph is force-simulation arranged, but the arrangement is not strict and allows customization.
- The UI feels overflown with information and can be hard to navigate.
- Near the corner of visualization there is a mini-map, which shows a whole view on the graph and pictures the current view area on it, which also empirically felt useful in navigation.

- Rich graph collections with massive amount of nodes and connections between them is very hard to read. Even utilizing various notations for classifying and grouping data is not helping a lot, the overview it still difficult and precise graph examination becomes gradually harder.

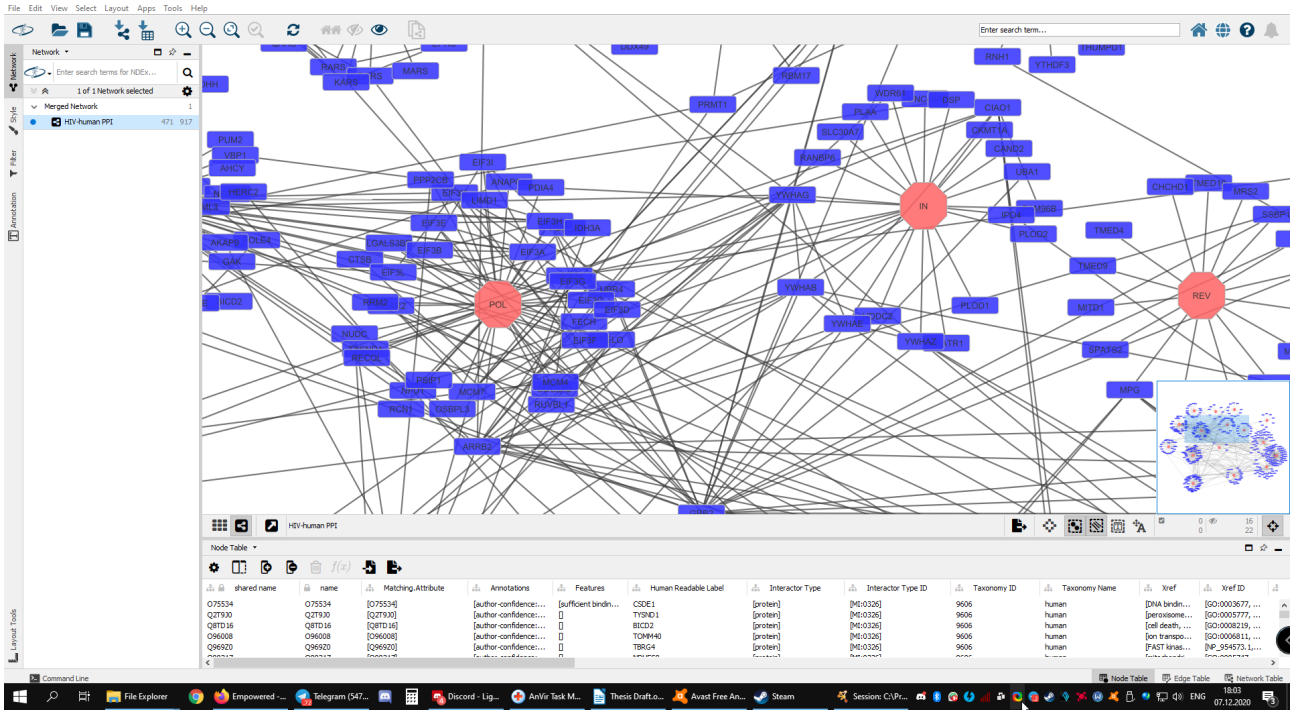


Figure 19: Example of massive graphs collection with maximized graph view area.

In summary, the Cytoscape application was proven to be the most complex and most developed (especially UX-regarding) solution, with a lot of design decisions being close to PO view on SPQ-02 project design.
However, the major interest for the research was in the core of the side-projects, initially named as Cytoscape Web. While the main project is a stand-alone application with its code written on Java, which allows cross-platform capabilities and having open API encourage users to develop various plug-ins for the app, expanding possibilities, it does not match with our scope. The separate branch of the project, called initially Cytoscape Web and later renamed into Cytoscape.js, uses a different code for the operation, sharing with the main project only design concepts. This JavaScript library is not intended to be a separate web application, rather being a component, which is embedded to the webpage for complex data visualization. The interoperability between Cytoscape 3 and Cytoscape.js is listed as a planned feature, but in the scope of this project is not considered as an important addition. The library was studied and implemented, investigation results are described in this work later in the section 3.3.4.

3.2 Fail-Safe RDF-PoC

3.2.1 Project overview

Fail-Safe IT Solutions OY – this organization is an IT service provider operating in Finnish and International markets, providing B2B (Business-to-Business) services in spheres of enterprise data collection, aggregation and analysis. Both providing assistance on configuration and support of complex data-driven environments and serving own software solutions as services, Fail-Safe helps with managing applications, devices and server ecosystems, analysis of records and log files, overall boosting other organizations' performance and security.

The FS RDF-PoC states to Fail-Safe Resource Description Framework – Proof of Concept – is an experimental project developed by the R&D branch of the Fail-Safe OY. The main goal of the project was to create an environment for operating graph-based configuration data for other systems. The project was developed during several years and contains separate parts, including database design, server backend implementation, API for interactions and various UI tools for management and end-users. The design was split into small milestones (targets), with the purpose to analyze the cost-efficiency of the research and development of such a product. The project is still in development and is not released to the public, therefore several things described in the work can change, such as: UI layout, data formats, code and technology stack. Moreover, this work describes only one part of the project – the web-based UI.

The iteration of the project, in which the implementation of the GUI was started, was named as SPQ-02. The project has two main views on the usage of the potential product: a short-term planning and long-term plans, defining the vision of the project in the closest release and vision of the project in some possible future. After refining all requirements, the picture of the initial release would be as a UI for a rapid access to the configuration storage system, its analysis and configuration manipulation in the form of visualized graphs. It also should serve as a control tool for managing user accounts. The long-term plan is to transform and enlarge the area of usage into a tool which allows to visualize and interact with any other graph data incoming into the system. Therefore the direction of the project development should be towards the short-term plan, but leaving enough possibilities for a further expansion. Another approach can be an implementation of the general solution which can be rapidly configured to serve the first version, but also can be later reconfigured for the long-term plan as well.

3.2.2 Environment overview

Usually, the the project environment has major impact on the project. IT consists of tools and technologies preferred by project owner, expertise of the developer and work practices enforced by the market. While some points can be influenced by the developer, some are set in the requirements and cannot be changed.

3.2.2.1 Development environment

Usually, the development environment in IT can be described by the tools that the developer uses for work, which consists of the operating system, IDE, redactors and other software. While most of it is solely a choice of the developer, the project owner and company can recommend or enforce some of the tools as common practices in compatibility or security purposes. Usually, the more developer is familiar with the tools, the more efficiency can be achieved.
In the current project, the hard rule was the usage of the Linux OS family as the development environment, for the sake of security and cost-efficiency. The actual Linux distributive choice was done for Fedora Linux with the GNOME desktop environment. However, the OS choice also affects

the choice of other tools, as they must be compatible with the system. In case the tool is not available in the operating system, analogs should be found or solutions for making the tool compatible, such as virtualization and emulation.
The other tools included:

- IDE: Visual Studio Code – a cross-platform code redactor with a broad extensibility with a variety of plugins and add-ons;
- Browser: Google Chrome for testing environment;
- Text processing for documentation: LibreOffice – a free and open source office software solution;
- Graphical redactor: GIMP;
- Git (source control system): GitGUI for analysis and easier access to git UI;
- Screen capturing tool: Flameshot;
- Collaboration: Google Hangouts – tool for video conferences and screen sharing;
- Feedback collection: Google Forms; andAdditional tools provided by the company for documentation and code hosting, inter-chat system and production pipeline.

3.2.2.2 Technology stack

The technology stack is a description of the technologies used in the project, which developer should be familiar with. It usually consists of the platform, on which project is developed, programming languages and several most important libraries and technologies that are used in the development. The technology stack that was specified in the requirements included several technologies:

- JavaScript
- React
- Redux
- REST
- HTML5
- CSS3

A bit more detailed description of the core technologies:
JavaScript (JS) – high-level object-oriented programming language. It has curly-bracket syntax close to Java and implement dynamic-typing. Initially designed as a scripting language for client-end webpage interactivity, JS was popularized and nowadays is used as a self-sufficient programming language. Despite the fact that it is possible to use JS outside of web development (example: Unity Game Engine uses JS as one of the possible coding language), the major application for this language is still web development. The typical web technology stack nowadays usually consists of HTML+CSS for webpage markup, PHP or Python (or any suitable back-end programming language) for server development and JS for front-end development. The part of JS code is executed by the browser on the user side, which provides a full application development possibilities for the webpage. However, the JS was not meant for complex programming solutions, being executed by the browser, and is not optimized for such a use-case, therefore the complex computations should not be performed on the user side and that is why the server side is used.

React - a JS library created for providing a complex framework for building user interfaces. It was initially developed by FaceBook, before getting a huge popularity and support from the community of web developers. The main idea of the React was the creation of one-page web applications similar to the desktop applications, with dynamic content change and updates to the webpage, directed by the engine. It can be described as a dynamic HTML generator based on JS that implements component modular philosophy, where each app component is represented by a

separate class, which are combined by other components and re-rendered only if changes occur inside of them.

Redux – JS library that was created for managing the web application state and data-flow. It utilizes actions, action listeners and reducers, as well as global data store to maintain a single-direction dataflow inside the app. That means that at every point of the time there is only one version of the application state and any action aimed at the state change will be added to the dataflow. As a result of any change, all actions will be combined and a completely new state will be calculated, avoiding mutations, which could be dataflow breaking. After that the new state is distributed across applications elements, so every element has the only one recent version of the state. This library is combined with React, providing the state machine and allowing a precise dataflow, reducing inherited and cascaded data passing and providing good communications between distant components.
REST (Representational state transfer) - A client-server architectural style for web services. It defines the relations between front-end and back-end and enforces common practices and templates in front-end - back-end communication.

HTML5 (HyperText Markup Language version 5) – a markup language for web page formatting, while is not a programming language by itself, is required for creating and operating content in the WebApp. With the React environment, it is inserted right into the JS code, as JSX functions, allowing a full merge with the code providing a flexible way of development and dynamic content manipulation.

CSS3 (Cascading Style Sheets language version 3) style description language, which often comes inseparable from HTML, being not a programming language itself, offers a possibility to describe the style, look and animation of the web page. During the development it is implemented in different forms, and can be inserted into the HTML code directly or applied from a set of rules, collected in separate files.

The required technology stack clearly formed a project into a Web Application development that also applied several additional technologies as dependencies [Fig. 19]:

- Node.js [21]
- NPM [23]
- Webpack [24]
- Axious
- Bootstrap

Node.js – a runtime environment for JS code outside of the browser. It can be described roughly as an analog of interpreter for some other languages like Python or Java. [22] It allows to run back-end services written on JavaScript, which therefore allows to create a full operating web environment with less variety in technology stack, replacing other server-side programming languages. It is an open source and cross platform, wide-spread tool that brought the frontend-only script programming language on the level of universal programming languages, and a major popularity of this language nowadays in several other spheres, not strictly related to web design.

NPM – Node Package Manager, operating with Node.js, this tool acts as a manager for repositories and packages used in building a Node.js web application. It provides an interface for the Node tools and can install, update and run components of the web project. Also, NPM is a software library (registry), which contains over 800,000 code packages. All packages installed in the project are

collected in a separate folder and registered in a special file, managing versioning and NPM settings.

Webpack – while NPM brings all needed code libraries as dependencies to the project and React-Redux bundle allows to create WebApp as an assortment of scripts, making the project file structure more complex, Webpack allows to bundle everything into singular files as a result product. This means that with the help of this technology, the dependencies are analyzed and extracted, all code is sorted and combined together, forming just a few files needed to run the website, usually keeping HTML structure, and applying to it single CSS and JS files with all styles and scripts, plus assets in form of media files. Webpack requires several other libraries known as loaders, for different types of files, which need to be processed.

Axious – a popular library for handling back-end communication of the WebApp through the JS code. All REST API in the project uses it for making GET, POST and DELETE requests.

Bootstrap – the most popular CSS Framework for developing responsive websites. This is a collection of the CSS scripts and presets, which allows rapid and stable styling of the HTML code, providing many commonly used functions from the box with little of none configuration. The framework also heavily focuses on modularity and flexibility of the structure, creating a responsive design, which adapts to the users' screen size and interaction methods. In the project it was represented as a React-Bootstrap component library. More about it can be read further in UX related part of the research [section 3.5.2.2].

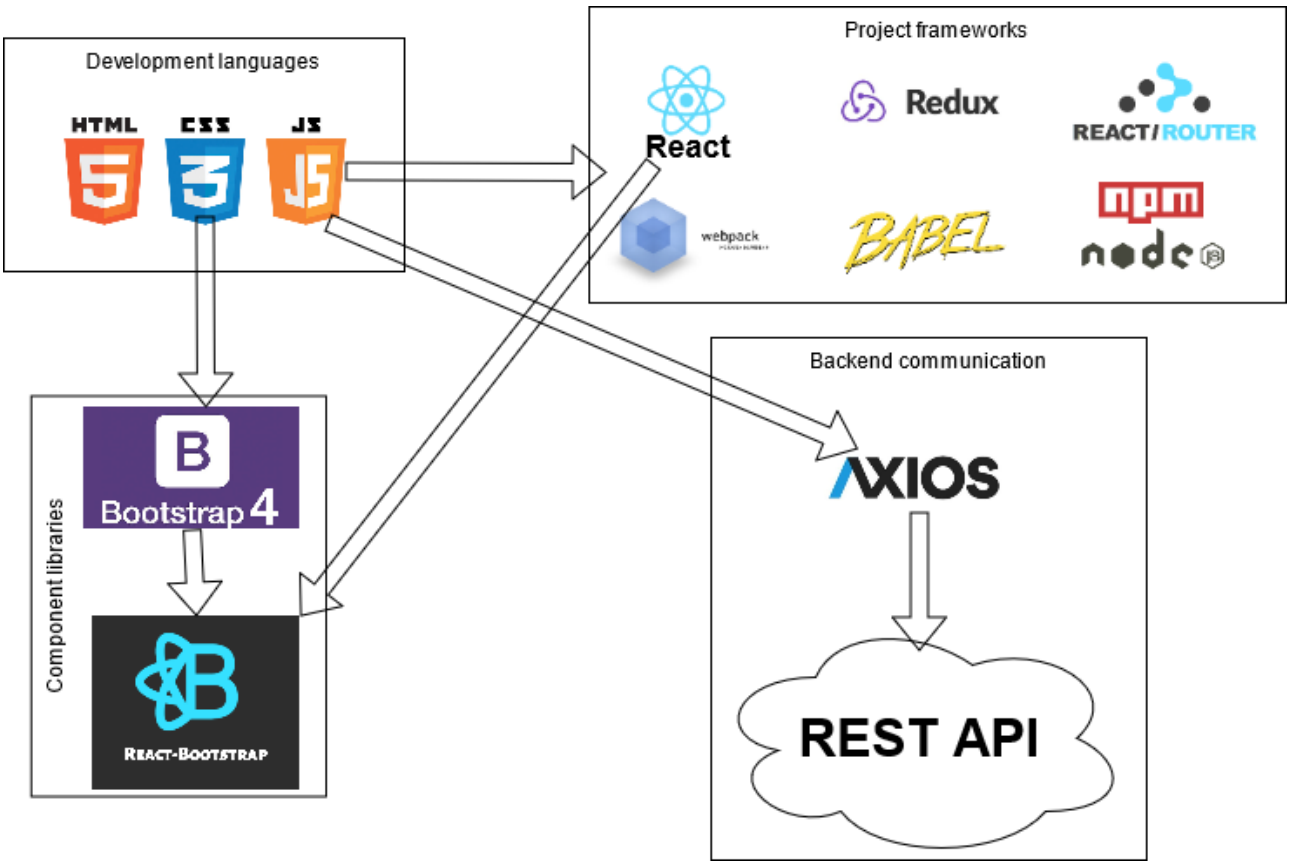


Figure 20: Grouped technology stack with relations

3.2.2.3 Production pipeline

The production pipeline is a description of the processes, performed with the project from the start to the end, where the actual product is released. It consists of several steps, each implemented with specific technologies used, and can be roughly divided into stages [Fig. 20]:

- Code is planned.
- Code is written.
- Code is submitted to the source control and pulled by integration services.
- New version of the project is created and loaded to the build environment.
- Build environment tests (if provided) automatically within the code, if tests passes – builds (compile) it into an artifact (ready-to-use product).

Next steps are optional, but usually implemented to continue the pipeline with delivery and integration:

- Product is packed to be easy-transferable as an installation package.
- Product is uploaded to the repository of the project.
- Product is delivered from the source to the target systems and is installed/updated on them.

In this project, the final result achieved is a web-server, which provides access to the WebApp, which is connected to another web server, providing access to the back-end. Going deeper in the details of this process would not affect the comprehension level of this work and will be omitted.

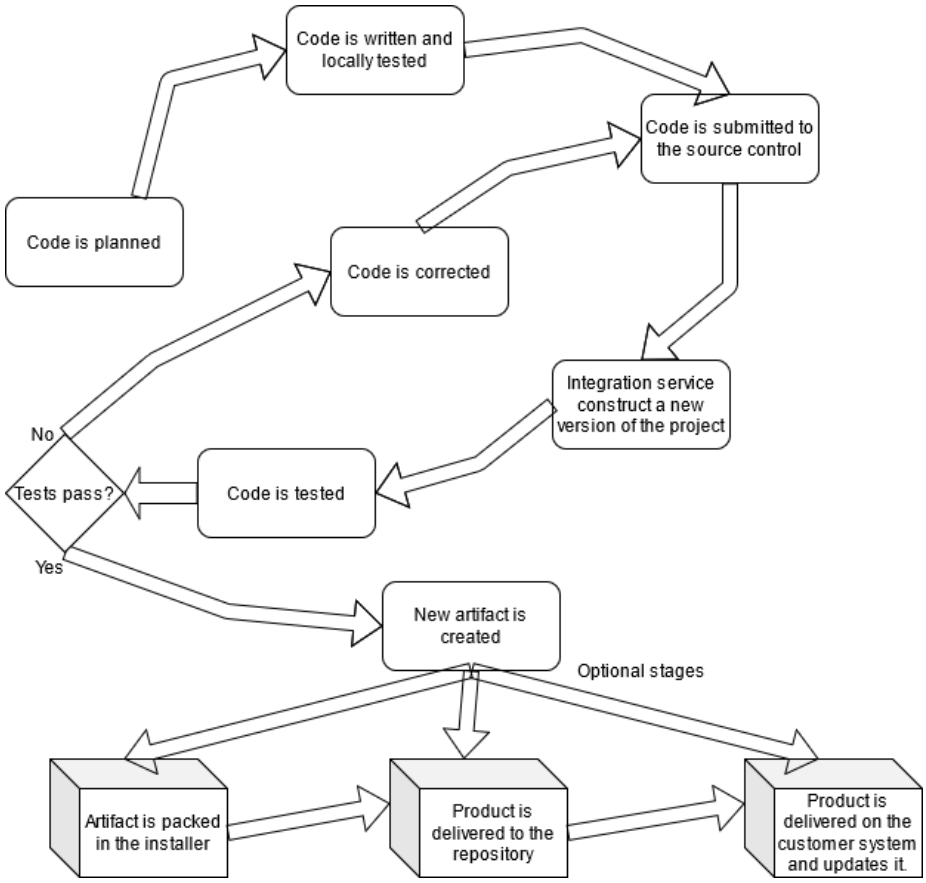


Figure 21: Production pipeline diagram

3.2.3 User stories

One of the possible ways to describe the planned product functionality and scope the requirements is called “User Stories”. It is a set of scenarios, which defines both who will use the product, and what they expect this product to do, in an understandable and human-readable way. The user stories are later refined into a backlog (feature implementation list and order) or any other sort of development management techniques.

The following is a list of the most common user stories:

- As any user, I want all my data secure.
- As a daily configuration manager, I want to visualize data relations and store visualizations.
- As an advanced user, I want to summarize data type occurrences.
- As a user, I want to be able to work with the system even if there is an outtake of some parts of the system.
- As a user, I want to be able to execute multiple parallel queries.
- As a user, I want to be able to share the link of the visualization with others.
- As a user, I want to have access to the system account base and authenticate with my login/password to get an access token.
- As a user, I want to login, so I can access to my work and profile.
- As a user, I want to sign out, so no one else can access to my account.
- As a user, I want the system to remember me, so I don't need to type my password every time I login.
- As a user, I want to create a new window for every task, so I can separate my work.
- As a user, I want to add, delete or modify an element of the graph.
- As a user, I want to view a separate element.
- As a user, I want to undo, so I can erase my mistake.
- As a user, I want to redo, so I can restore mistakenly undone element.
- As a user, I want to see my roles in the system.
- As a user, I want to change my password, so I can improve my information security.
- As a user, I want to save my work locally, so I don't have to send it every time to the server.
- As a user, I want to save my work to the server.
- As a user, I want to work offline.
- As a user, I want to access other services embedded to the UI.
- As an inexperienced user, I want to use simple mode, so I can use only main features.
- As an experienced user, I want to use advanced mode, so I can use additional features too.
- As a system administrator, I want to create, modify and delete users in the system.
- As a user, I want to be able to create new graphs from scratch and send them to the server.

3.2.4 Refining the backlog

After refining and discussing the user stories, several project artifacts were created. First was backlog – a prioritized list of nuclear (split to the smallest part) value-adding features, which should be completed during the project development. The backlog is a very important part of the most common Agile development technologies, but it is also helpful in any development planning, as it allows to create a fluent development process, predict resource costs and set up milestones. The second artifact is a collection of user portraits, describing the most common types of end users. This is important for user interface design, because it allows to adjust UI to the stated specifications or create different designs for different needs.

The short summary of backlog (list of prioritized features) looks like:

1. Environment based on React-Redux, which can securely communicate with the back-end.
2. Environment allows to authenticate and use security tokens.
3. Environment allows to access back-end data.
4. Environment visualizes data.
5. Environment allows to interact with data and modify it.
6. Environment allows to manage personal user profiles.
7. Environment allows to administrate the system.
8. Environment allows to create new data entities and submit them to the back-end.
9. Environment provides embedding functions, such as embedding other web services and web pages inside the application, rendering remote UIs and media.

10. Environment provides a standard set of web tools to work with UI (URL, undo-redo).
11. Environment provides means of tutoring, feedback gathering and other product connections.
12. Environment should have room for expanding more data variants and their applications.

Later the backlog was grouped into several milestones:

- Milestone 1: Creation of the environment base, expanding technology stack and solving the basic abstract problems.
- Milestone 2: Creating a secure communication procedures for connecting UI to the back-end and managing it.
- Milestone 3: Creating a system for acquiring data from the back-end and visualizing it.
- Milestone 4: Developing interactions with the data, both graph data and user data.
- Milestone 5: Optimizing and enhancing the UX.
- Milestone 6: Creating a set of tools for data creation, feedback gathering, additional functionality and project expansion.

The milestones were executed in order. In the end of Milestone 6 the release of first public version of the product is set.
The user story refinement also provided several user portrait descriptions:

- The business owner: a person, who implements F-S services in order to enhance the business processes. This person does not need most of the SPQ-02 functionality, putting in the first place simplicity and a possibility to have a full view on the current situation.
- System administrator: usually an IT-skilled person, who knows what is happening in the system and is aware of technical details. This type of user prefers more information and control over simplicity, needs more fine tuning and access to various low-level functionality and system reports. Must also access tools for system management.
- Daily user: a person with an undefined skill set, which uses SPQ-02 for its main task – graph data (configuration files) examination, manipulation and creation. The average user must not have system administrator privileges, prefers simplicity and speed of the work over everything, needs automation of the most repeating tasks to the most possible point.

The user portrait description highly emphasizes the need of separations or several UI approaches, which should be hard to mix. Based on the portraits, it was decided to design several variants of the UI and to let the user to choose from. However, this was set beyond Milestone 6 and is planned as a future feature. For the first release version, the UI should suit the two last user portraits, at the same time allowing to navigate smoothly through the daily routine, but also providing enough additional control based on the user access level and tasks.

3.2.5 Challenges

Any project requires a clear and thoughtful analysis of risks and challenges. Forseeing the possible problems and causing factorsbefore starting the development allows to evade some of them or prepare for solving them with little to none surprise effect.
Common challenges:

- The data model does not yet have wide-spread popular applications or any well-known successful problem solutions.
- Project-specific challenges:
- The project was requested to be done in a very specific environment, using predefined technologies.
- The project is at the PoC state, and does not have a user base for feedback and field testing.
- The project development requires at least basic knowledge in multiple fields, knowledge of specific libraries and constant adaption to new technologies.
- The project was supposed to be usage specific, but later the requirements were expanded to a much broader functionality.

3.3 Visualization frameworks

After the research on the already implemented solution of the problem, several development plans were made and the expected results of the project were clear. However, as it was already mentioned, the researched examples of linked data visualizations were not suitable for the project environment or would apply various limitations and heavy modification of environment, which counted in the same way. Therefore a new research of the possible technologies, suitable for problem solving was done. The requirements for the technology were:

- Technology should be web-oriented front-end based, written in JS or translated/connected to JS code with adequate amount of effort.
- Ideally, technology should be React-compatible.
- Technology should allow interpreting linked data in some format and return a set of visuals, which can be displayed on the page.
- Technology should provide possibility to expand its functionality with additional programming.
- Ideally, technology should provide graphical-engine-like functionality, allowing to interact with visual data or provide an API for that.
- Technology should be open-source.

For the visualization implementation in our environment, several frameworks were found.
After testing and comparing them, two were chosen to take further part in development. They were implemented in a prototype-format for the project-specific tasks and evaluated on the performance, as well as on the overall cost-efficiency of the implementation.

3.3.1 D3

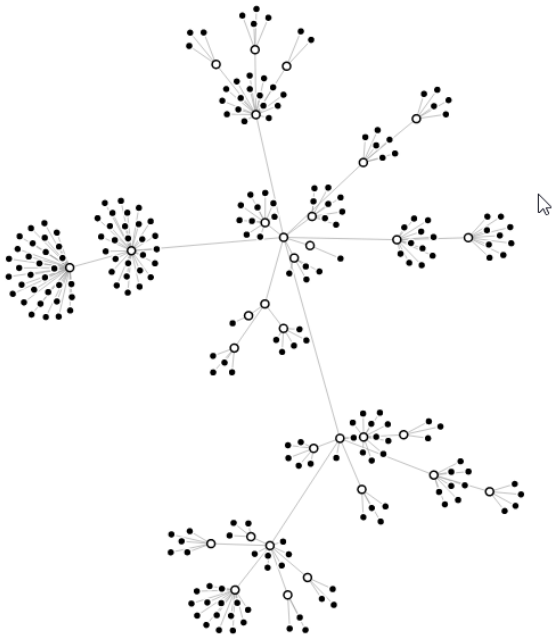


Figure 22: A Force-Directed Tree implemented with D3 library. [25]

Data-Driven Documents (also known as D3.js) is a JS library for dynamic graphic rendering inside web applications. It is vastly popular and provide interfaces to all basic graphical drawing functions. It utilize SVG (Scalable Vector Graphics), HTML5 and CSS standards. It contains no graphical

engine logic and user needs to implement it from scratch, combining from basic graphical primitives. It is not React-specific library, so it requires an additional work for implementing it React way, however, since the whole D3 code logic would be implemented from scratch, it is not a big difference.

3.3.2 Vis.js

Figure 23: Vis.js community edition logo.

A dynamic, browser based visualization library. The library was designed: to be easy to use, to handle large amounts of dynamic data, and to enable manipulation of and interaction with the data. [26] The library has developed logic for graphical rendering and works more like a framework for data visualization and interactions. It has several subparts, each related to a separate utilization of the library, including:

- DataSet (data organization and management)
- Timeline (customizable, interactive timelines)
- Network (dynamic, automatically organized, customizable network views)
- Graph2d (dynamic graphs and bar charts)
- Graph3d (interactive, animated 3d charts)

This project will use and describe DataSet and Network components.

3.3.3 Comparing and final choice

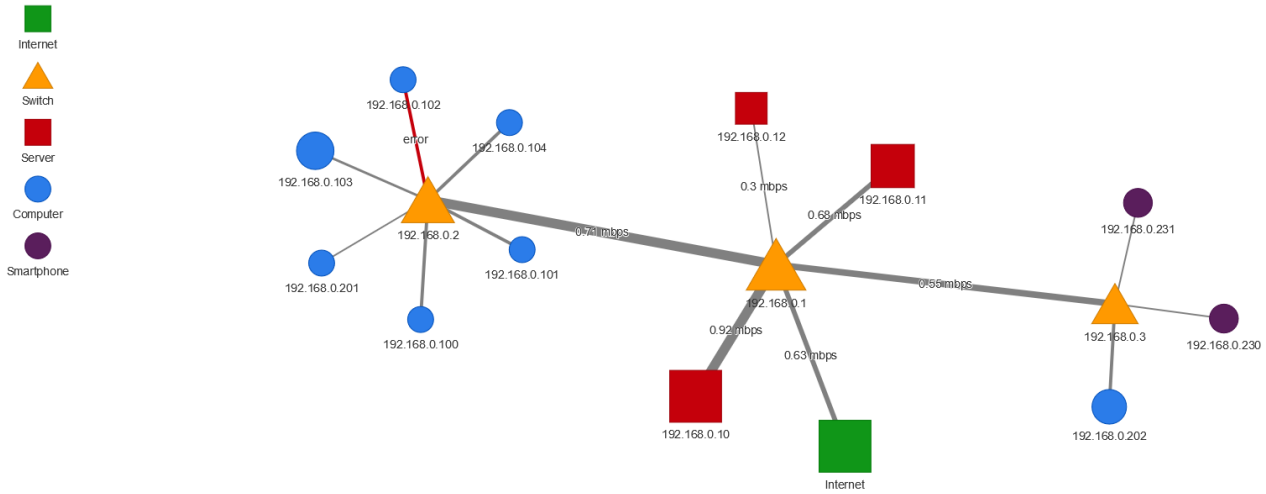


Figure 24: Example of the Network component utilization in implementation of force-driven graph with various elements customization. [27]

Based on testing, manuals and documentation provided to the both variants, an analysis was performed and a table was formed to a clearer overview on the benefits and drawbacks of both solution's implementation in the project [Table 2]. This table was presented to the project owner for a final design decision.

	Vis.js	**D3**
Description	A data-driven graphical engine.	A library for operating SVG elements and creating graphical engines.

Need a parser?	Yes. (Operates DOT language – simple node + link notations). Beta-version of parser is implemented.	Yes/no. Depends on graphical module implementation. It can include some parsing algorithms inside. Basic parser done for current implementation.
Complexity:	Low in implementation. High in modification.	High in implementation.
Customized by:	Option file, which can be moved to separate source for configurability.	Engine-customized and fully depends on implementation.
Extensions:	Various modules from the same project.	Plentiful variety of community-written libs and add-ons.
Amount of hand-written code:	Low	High.
License:	MIT	BSD
Working with React?	Integratable with wrappers.	Require customization.
Advantages:	Easy to implement. Efficient and flexible in customization. Networking module suffice almost all our needs. The engine is already done, needs only the correctly structured data and options to operate.	The engine can be created under very flexible requirements, every spot in it is configurable. Popular library, which mean plentiful amount of manuals and troubleshooting recommendations.
Disadvantages:	The engine expansion and modification might possess a challenge, if they are not implemented or were not planned. Everything relying on the engine API.	Library must be studied from the basics to create anything sufficient. There is no engine, only a typical graphical libraries with functions and procedures. Everything must be set up from basics, all logic must be hand-written, which means that the graphical engine should be built from scratch.

Table 2: Comparing Vis.js to D3 in an early stage report.

Based on the analysis results a conclusion was made: if a project needs a hand-written own graphical engine, D3 will suit the purpose better because it can be expanded, it is flexible and can be adjusted to any raised needs. However, it will require a full study on D3, which may not be time/ cost efficient to the results. At the same time, Vis.js is already providing a simple and feature-full solution, if it is implemented as-is. It can provide most of the needed functionality “from the box”

(e.g. without complex configuration) .It only requires a code responsible for handling the user interactions output back to the system in addition to the input graph data translator. It can be a time & cost efficient solution, but if in the future a project will need to expand the functionality, it might be more difficult to implement.
The final decision was clear: the project already has fixed requirements, so a more cost-effective solution is considered preferable. The D3 development was discontinued and Vis.js libraries were put to full integration.

3.3.4 Cytoscape.js

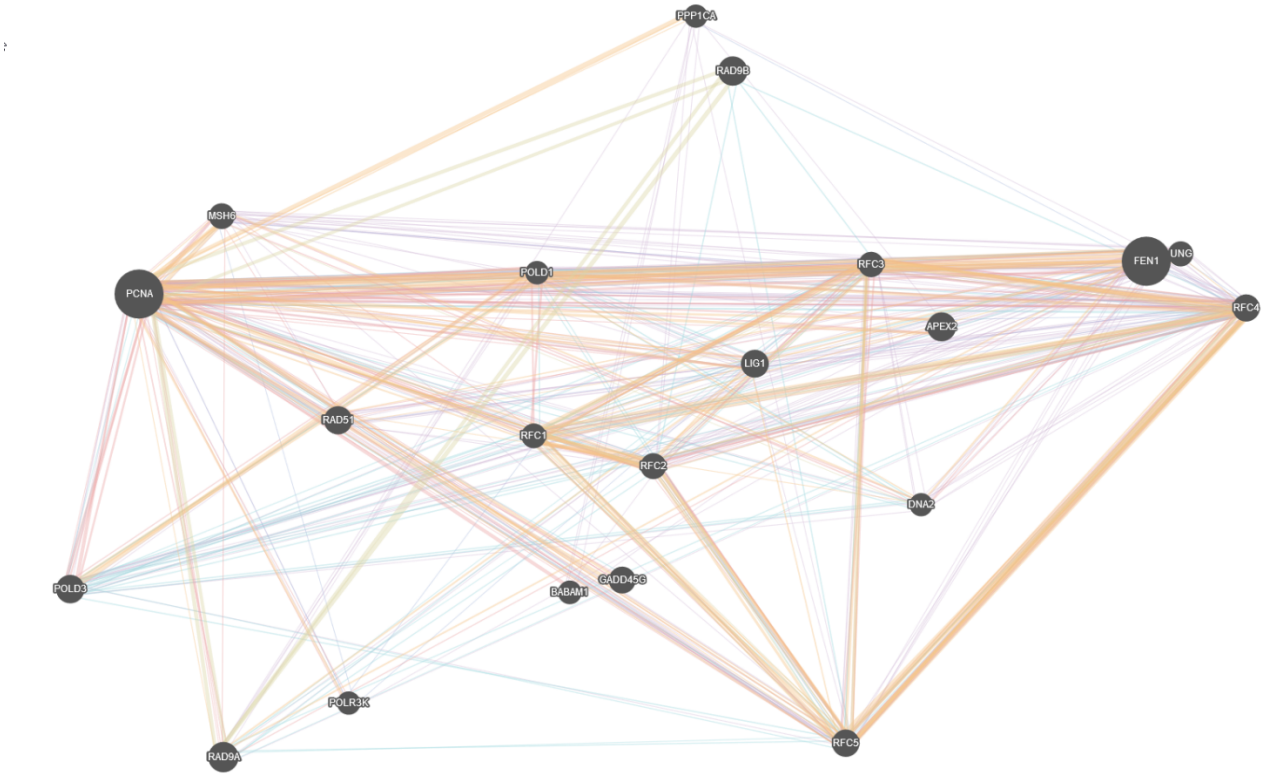


Figure 25: Visualisation of the graph of gene-gene interactions, utilizing Cola.js for layout and Cytoscape.js for visualization. [28]

The first release of the product was planned to be an open-beta. The main visualization library of the first release was Vis.js, which performed well and had proven to be simple and reliable solution to the problem. After the first pre-release version testing, a new suggestion was made, to try and implement another alternative to the Vis.js library, which was Cytoscape.js library. While the Cytoscape project itself is positioning as cross-platform ready-to-use from the box solution based on Java, which is not suitable for our specifications, the Cytoscape.js is a separate, completely based on JS project, which can be easy to integrate into the current Web application. The library was studied and tested in integration to a separate branch of the product, dedicated for that.
While having a lot in common, Vis.js and Cytoscape.js have several completely different ways of feature implementation, so a thoughtful evaluation was done to estimate, whether the costs of the new library implementation instead of the old one match the benefits it will provide.
At one point of the testing opened up that there are two ways of implementing the Cytoscape.js, the traditional way of implementation, similar to the Vis.js implementation, and as a React component, which had its own specific approaches to implementation. The React version was finally separated to yet another project branch for conflict prevention.

3.3.5 Comparing Vis.js and Cytoscape.js

The first impression the library gave was very similar to the one of Vis.js. The traditional approach of the library implementation in the project was much the same as in the old one – the library was imported to a separate handling component-class, initialized and, after creating a DOM object on the page, an instance of the visualization was created, provided the access to the object, visualization data and configurations.

The visualization data was based on almost the same syntax as Vis.js data, while not separating the nodes and edges, combining them altogether in one array of objects, where each object represented one element of the graph. This lead to modifications of the data parser, previously used for Vis.js. However, the main algorithm stayed intact, only correcting the syntax and returning a correct set of data.

Right from the point when the new parser started to work correctly, the library started to create visualization. However, the first visualizations were much less impressive than the ones Vis.js was doing from the start. The visualizations looked more like what D3 was producing after some point of implementation on early stages. That was the point, where major differences of these two libraries showed up.

	Vis.js	**Cytoscape.js**
Configuration	Stored as one config object. The object contains all basic options, split by sub-objects and allows a complex on-the-go configuration of the visualization.	Stored as separate objects and combined together at the visualization initialization. Each object represents a feature or a basic set of options. Every expansion requires a separate configuration, which can create a massive overall configuration process.
Access to the options for user?	Provided as ready-to-use panel, fully customizable and integrated.	Requires additional coding to implement. Will require a completely new design from scratch for on-the-go adjustment, may become troublesome to make it work.
Visualization data	Requires two sets of data: node array and edge array, wrapped into corresponding Vis.js datasets. Each element is represented as object in the dataset. The node can be described as: {id: “id of the node”, label: “text of the node”, + additional cosmetic parameters}, and edge: {id:”id of the edge”, from: “id of the parent node”, to: “id of the child node”, label:”text of the edge”}, which provides clear and rather simple way of graph creation and manipulation. The graph is bound to two elements/primitives.	Requires one set of data: an array of objects, each containing excessive description of the graph element. The description of the node is very similar: {id: “id of the node”, label: “text of the node”}. However, the description of the element must be wrapped in “data” property. The final element would look like {data:{example above}, additional props}. The edge format is almost the same and would be: {data:{id: “id of the node”, label: “text of the node”, source: “id of the parent node”, target: “id of the child node”}, which allows to convert datasets from Vis to Cytoscape with just

		merging edge and node arrays, encapsulating the data in object properties and renaming few fields. The same can be done in a reverse way.
Features	Most of the features are included and ready-to-use from the box, including • physics; • interactions; • navigation; and • various layouts.	Features are subjects for expansion and can be added via existing feature add-ons or coded. Basic version includes • Few simple layouts; • Panning navigation; and • Drag-n-drop reordering. However, multiple expansions are available and can expand the capabilities of Vis.js.
Interactions	Several listeners for element manipulation events, enough to implement the basic interactive environment. A set of tools to manipulate and modify the graph data. Additional functionality may require complex coding and library modification.	Core library provides a collection of tools and listeners similar to Vis.js, however, they are harder to access and control. Extensions provide their own, feature-related set of interactions, each implemented separately, widely scaling from easy to use to unobvious.
Works with React?	Already implemented with a wrapper code.	The same wrapping code can be implemented or a completely functional React-based fork of the project can be used.

Overall summary	Currently the library is used at 65%-70% of its capabilities, several advanced visualization and interation features are yet to be implemented. However, expanding functionality of the library would be hard.	From one point the library looks promising and can offer more flexibility and growth potential in the future. The styling and CSS rework may provide the biggest troubles, as it is implemented in a different way from Vis.js, also feature-based implementation and configuration can make library-handling code overloaded and hard to read/navigate. The correct requirements for the current and future development plans should be made and list of extensions for implementation should be created.

Table 3: Comparing Vis.js to D3 in an early stage PO report.

3.3.6 Cytoscape-React

During the first implementation of Cytoscape.js, an extension, which opened up as a different implementation approach, was found. This module is imported and implemented as a completely functional React component, with all related lifecycle modes and methods and in fact is being a wrapper around Cytoscape library, providing a smooth and easy integration into the React environment. While the implementation of the original Cytoscape.js is similar to the Vis.js and provides a broad access to the library control, it also provides several limitations and inconveniences, such as container management and additional wrapping complexity. The extension implementation in the React component is less simple and intuitive, but after finding a stable way of doing it, as well as a callback to the network instance via a callback object, this method of implementation is considered supreme. The challenges that may happen in further development are assumed to be problems of incompatibility of extensions with the React wrapper.

3.4 Starting code analysis

The project had early prototypes which implemented some basic PoC visualization mechanics based on D3. The code was thoroughly analyzed and commented, the analysis was documented and presented to the project owner, accompanied to the development plans. The code was not implementing any other features except processing input data in the form of RDF triplets, converting them into simple D3 instructions and drawing a non-interactable forced tree based on that. After discovering other methods of visualization implementation rather than D3, the old code was mostly decided to be disregarded and the development took place from the clean start. However the fallbacks for the D3 solution was maintained, which can be found in the early architecture. [Diagram 1]

The initial project planning took the form of an architectural planning and several basic mock-ups in order to find the main strategy of implementing the UI. The React-Redux environment was considered quite unusual regarding the previous system design experience and ways of documenting it were slowly developed over iterations.

3.5 Progress

3.5.1 Milestones

Milestone 1

Milestone 1 was dedicated to the creation of the project platform, learning and getting used to the principles of the development environment, making the main design decisions and concepts. During this milestone the React and Redux technologies were explored, their potential and drawbacks learned. Since the environment was rather new, good practices and coding templates were learned and developed alongside the project.

In Milestone 1, the technological stack was also expanded, the NodeJS development environment was set up, combining NPM package management, webpack with modules for hot-load development and React-Redux stack. In the end of the milestone, the prototype of the React-Redux app was created, which was able to render dynamic webpage and operate several data structures via the Redux store. The basic architecture of the app was planned and several UI designs were tested, discussed and approved for implementation.

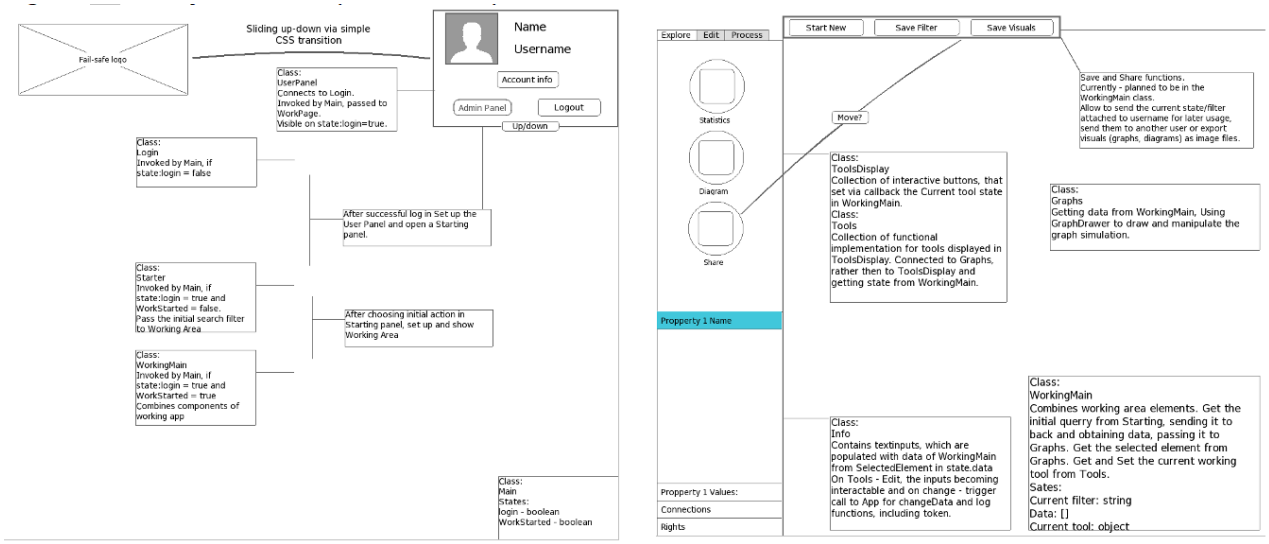


Figure 26: Early versions of the UI wireframes

Milestone 2

Milestone 2 was dedicated to the creation of the modules for handling the backend-frontend connection. This included REST API studies, exploring and implementing the Axious library and writing front-end API, which not only sends requests to the server, but also receives replies, formatting and extracting required information from them and distributing it with the Redux store to the rest of the system. Also a local data storing system was created, which allowed user to preserve data, like session token, in the browser storage, allowing reloading the page with minimal data loss. The milestone did not bring much to the UI itself and was more technically oriented, so additional details are out of the scope.

Milestone 3

Milestone 3 was dedicated to handling linked data on the front-end and implementing visualization libraries. A lot of effort was spent on learning the methods of working with linked data, methods of integrating non-native to the React libraries in JS into the React-based system. During Milestone 3, the prototypes of Vis.js and D3 were tested and Vis.js got the approval to the continuous development. At the same point the JSON-LD parser and node naming-navigation system were created [Appendix Text 1-3], which were initially designed to suit the Vis.js input data format.

Milestone 4
During Milestone 4, the application was massively expanded with additional features and logic, including a new personal user-management system, administrating tools and graph manipulation tools. Features required new logic, new libraries and new API endpoints to operate, developed simultaneously with back-end. Effort was also spent for refactoring and updating the system, as the amount of experience in the development of such an environment increased immensely, which brought up a need for standardization of the code, cleaning the old experiment's residues, commenting and documenting everything under one standard. The meanings of user-graph interaction were studied and explored, several prototypes of new features were presented for testing. A lot of code was created around the implemented visualization libraries to allow the user work with the acquired data. The UI started to shape into its final design, however the overall look and feel of the UI was not high.

Milestone 5
An important point of Milestone 5 was the introduction of the new developer to the project (making now the team of 2), Ronja Kyrölä, who took part in the development as the main graphical designer. In this milestone the UI was brought up to a new level, layout corrected and optimized, both the HTML and CSS structure of the project were reviewed and enhanced. Graphical assets of the project were re-implemented via additional graphical libraries, CSS naming was documented and refactored to a newly developed standard. The program logic was not changed much during the milestone.

Milestone 6
Milestone 6 was planned as the last milestone in this project lifecycle's iteration, before the first major release. However, due totechnical difficulties, the release date was shifted and the project is currently at the end of Milestone 6. The main points of the milestone were to create a new way for the user to create and modify the graph data, based on scenarios, as well as overall testing, polishing and optimizing the system for the production pipeline. The feedback collection and user aid features were also introduced in this milestone. The project also got new technology for visualization, which was Cytoscape.js and was forked into two branches for developing and testing the new technology, comparing it to the existing one.

3.5.2 Artifacts
During the development process, several artifacts were created. This includes documentation, planning, modeling and snapshots on application progress:

Artifact	Description	Development started
Backlog	List of features with the status of implementation. Each feature has status such as: "waiting for approval" – for possible features, "approved" – for planned features, "in development" and "done" – for different stages of development.	Milestone 1
User stories collection	Processed collection of "use-case" scenarios, which represents a requirements list. [Section 3.2.3]	Milestone 1
Code modeling	Collection of UML-based diagrams used for code	Milestone 1

	planning and UI prototyping. [Section 3.5.2.1; Appendix Diagram 1; Fig. 26]	
Entity relations diagram	Created for React-Redux structure planning. Also needed for navigation in React-Redux code. [Appendix Diagram 3]	Milestone 2
REST endpoint descriptions.	For back-end development planning. Each description contain method, parameters and output.	Milestone 3
Axious API	For front-end communication. Each point describes which endpoint is contacted, what data is sent and what output is expected.	Milestone 3
Dataflow diagrams	Examples of dataflow communications, one per endpoint pair. [Appendix Diagram 2]	Milestone 3
Reusable code list	Collection of code fragments description, which can be reused in other projects.	Milestone 4
CSS naming list	Includes naming standards and classes name list. Needed for navigation and management of CSS files.	Milestone 4
HTML structure schematics.	Displays the component hierarchy and tag nesting. [Section 3.5.2.2]	Milestone 4
Icon library	Collection of links and descriptions of icons used in UI. Since most of the graphical assets are taken from icon library, they are accounted in a register with information about place of implementation.	Milestone 5
Help panels collection	Documentation in form of tutorials, visualization interactions and user panel.	Milestone 5
Modular layout library and wizards API	Since these features require correct input from back-end, the syntax for the input is described in these documents.	Milestone 5
Cytoscape.js plug-in planning.	A collection of Cytoscape.js plug-ins descriptions and links to the projects for further development and implementation.	Milestone 6

Table 4: Artifact collection with description, sorted by starting milestone

3.5.2.1 Code planning

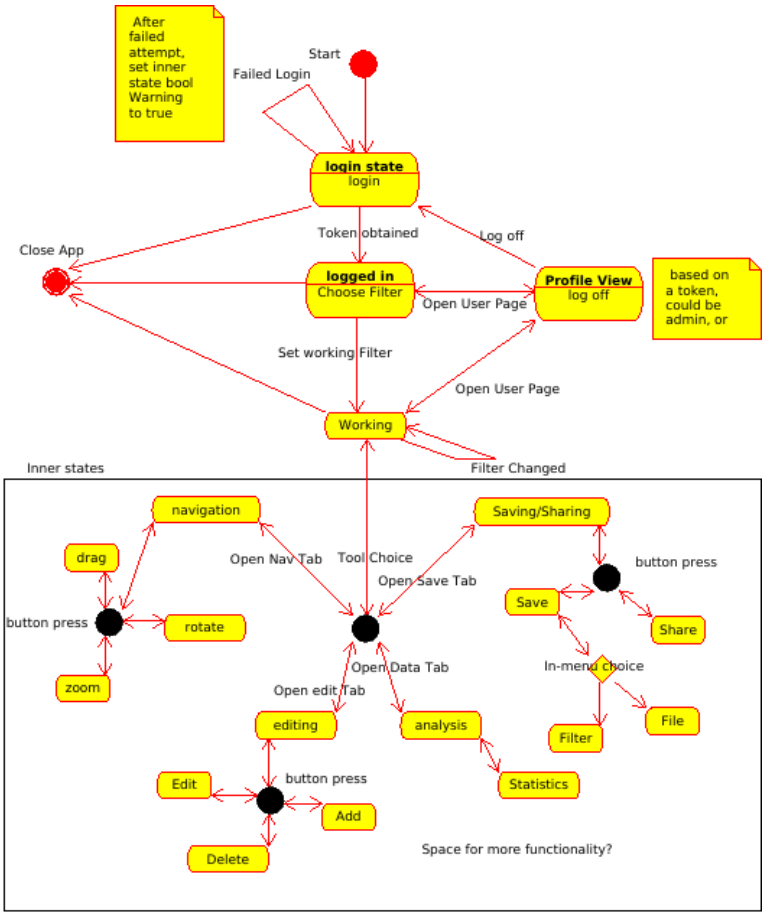


Figure 27: Early state diagram describing possible user's actions.

From the very start of the project, various planning activities were done and documented in order to simplify and clarify the development process. Examples of such activities are user stories, wire-frame modeling, UML models and prototyping, described in this and following chapters [Fig. 26]. The documentation produced in the process often took the form of diagrams and mind maps.
A significant amount of resources was spent in the early stages in order to understand better the new rules of React-Redux environment, which worked in unusual for the traditional web-development perspective way.
The React technology makes the process of web application design to be straight-forwarded, similar to to the GUI Applications development. It is achieved by making the app one-page document, where new elements generate dynamically and update after new data is received, it mixes HTML and JS into a combination of both. This might provide some confusion for a developer, who is used to traditional web development. Instead of making HTML documents hierarchy, the project required a JS class hierarchy. The Redux technology adds complexity to this by introducing completely new way of dataflow in the app.

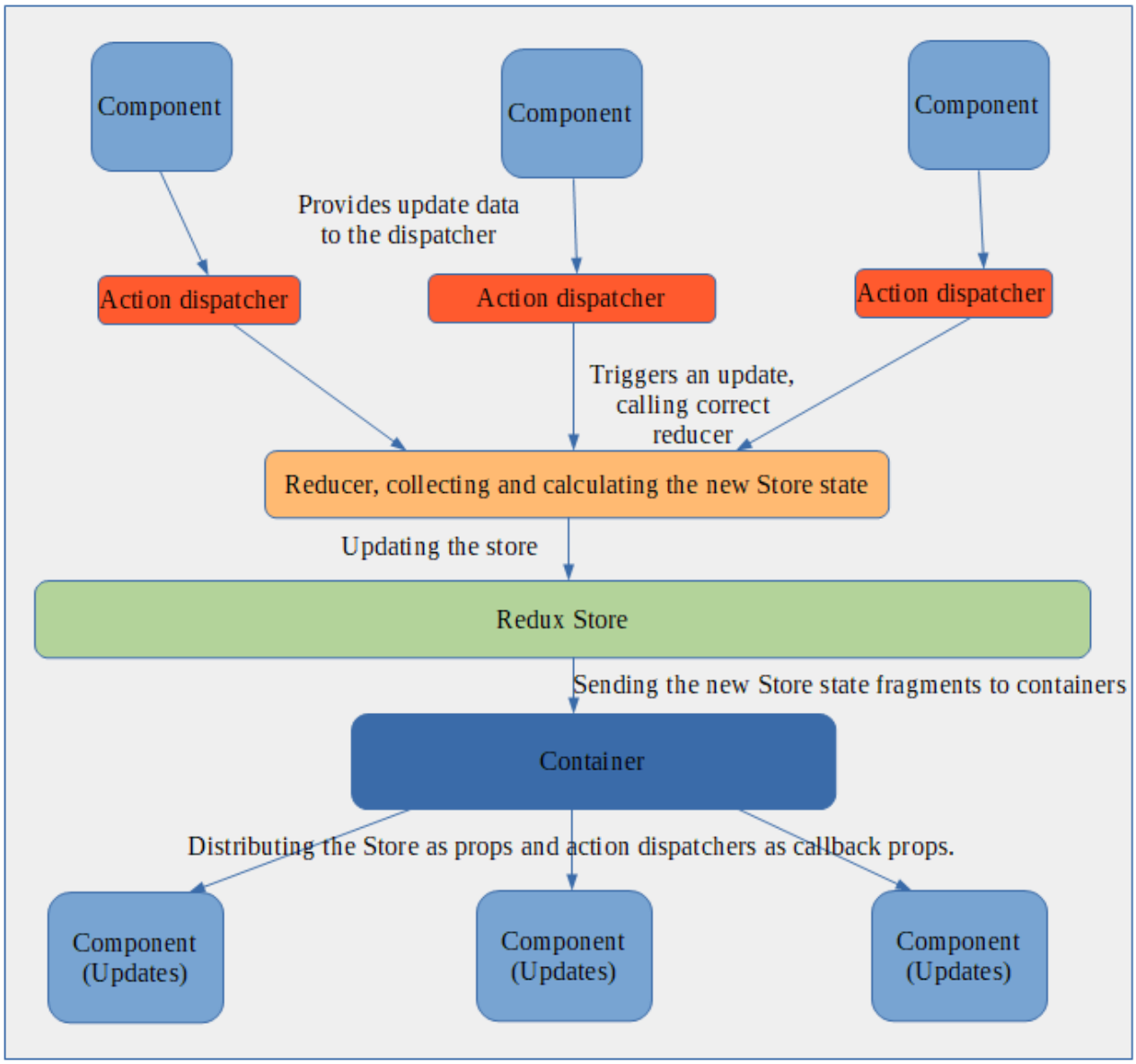


Figure 28: Redux dataflow principle

The structure of the React application consists of one (or more, if needed) web pages, which are mostly headers that combine technical code, not displayed on the page. That includes security policy, metadata, global scripts and files import, such as global CSS files, dependencies and third-party scripts. The actual body of the document is usually an empty block with a known identification. At this point, the React engine takes control and populates the empty block with all other content of the App. This is done via React routing and rendering, when various classes, called “React components” are executed. Each component has possibility to have a “render” method, inside which the actual HTML structure should be inserted and which will be displayed in the main document. The component also includes several methods with reserved names to manage a lifecycle of itself and run the code during lifecycle stages. The lifecycle of the code starts, when the React engine “mounts” the instance of the component on the page. Then the code receives its properties aka. “props” – the inherited variables, callbacks and objects, provided from the parent. Then the component is cycling though updates, re-rendering every time inner or props data is changed and finally the component got removed, clearing the data related to it. Next time the new, fresh instance of the component will be created and the lifecycle will repeat. The data of the component is stored within it as initially inherited “props” as well as component “state” – a special object, that contain

data during component lifecycle. Each change in the state (inside) or props (outside) triggers component update.
Therefore, the dataflow in the original React app is hierarchical in the form of parent-child and it could be redundant and cumbersome to pass data back and forth, especially when a distantly-related component needs to pass data to another component in separate hierarchic branch. Also, managing all different components with their own lifecycles, all updating and generating data asynchronous is difficult and dangerous in terms of data integrity and security. The Redux was introduced to the project to solve these problems. Initially inspired by the Facebook Flux architecture, Redux introduces a centralized dataflow with a predictable global state. The main idea is that there is a global data storage, which combines sections of data. The storage is operated by reducers, which, every time a change is required, gather the new data, calculate and create a totally new object, which replaces the old state and is distributed around the app as an atomic update [Fig. 27].
Redux also introduces new elements to the system:

- Reducers with reducer combiner – each reducer represents a part of the Store and namespaces for functions that update the subparts. It also includes the reducing logic. The combiner combines the reducers, forming the Store as a one main object and manages data distribution.
- Actions and action dispatchers – to trigger the Store update and corresponding reducer, it should be called via dispatched action, provided with update data.
- Containers – the same React component, with new purpose: this component can (but usually does not) contain React component logic, instead it collects Store data and dispatcher callbacks in form of props and distribute them to components it invokes.
- Reserved namespaces. By recommended design practices, each reducer call is assigned a reserved namespace, to which it would listen. The common practice for that is using capital letters. All namespaces are collected in separate resolving script, which consists of declarations of export constants in the form of:

```
export const DATA_UPLOAD_START = “DATA_UPLOAD_START”;
```

Therefore, creating any new interaction with Redux system should follow such pipeline: Create a section in the store by creating new or expanding existing reducer. If created new, add the reducer to the reducer combiner. Create initial store value. Create a namespace for reducer call. Create a reducer logic for updating the store section and assign it to the reserved namespace. Create a dispatching action for calling the reducer. Map the new store section and the dispatcher to containers, which hold the components designed to use the store or the dispatchers. Transit mapped dispatcher and store data in form of props to the corresponding components. The complexity of the interaction affects only the complexity of the action dispatcher and reducer’s logic. Every element in the system should go through the same pipeline in order to integrate into the Redux environment. This proves to be very tedious, if the system accumulates a large amount of relatively small, separated interactions.
This approach has several benefits and drawbacks. The benefits:

- The dataflow is one-way and is synchronized for everything.
- The data distribution is more convenient – since the Store is global, it can be accessed (but not modified) from everywhere.
- Every change triggers an update for every component that is subscribed to the part, which was updated.
- However, there are several drawbacks of the technology:
- This technology introduces a huge overhead in coding. If all dataflow in the app will be under Redux philosophy, every minimal data change would trigger a significant amount of computations. Moreover, to implement a minimal interaction, instead of few lines of code it is now required to write literally 10 times more code, as every interaction now includes not

only interaction logic, but also a store place, corresponding reducer, corresponding namespace, action logic and dispatcher.

- The dynamic variable managing of JS is one of the most criticized aspects of this programming language, and the nature of the handling objects and variables can introduce the whole new horizon of errors and malfunctions in the work of the Redux Store, especially when trying to store and manipulate complex objects.

The overall hierarchy of the inner components relationship can be described as an Entity-relation diagram. It is a modified class diagram, which depicts React components interation, data transition and groups components into categories in the Redux logic. The additional diagrams are made to describe the Redux relations with components and are grouped by reducers. The most recent version of mentioned diagrams can be found in the Appendix section [Diagram 2- 10].

3.5.2.2 Models and wireframes

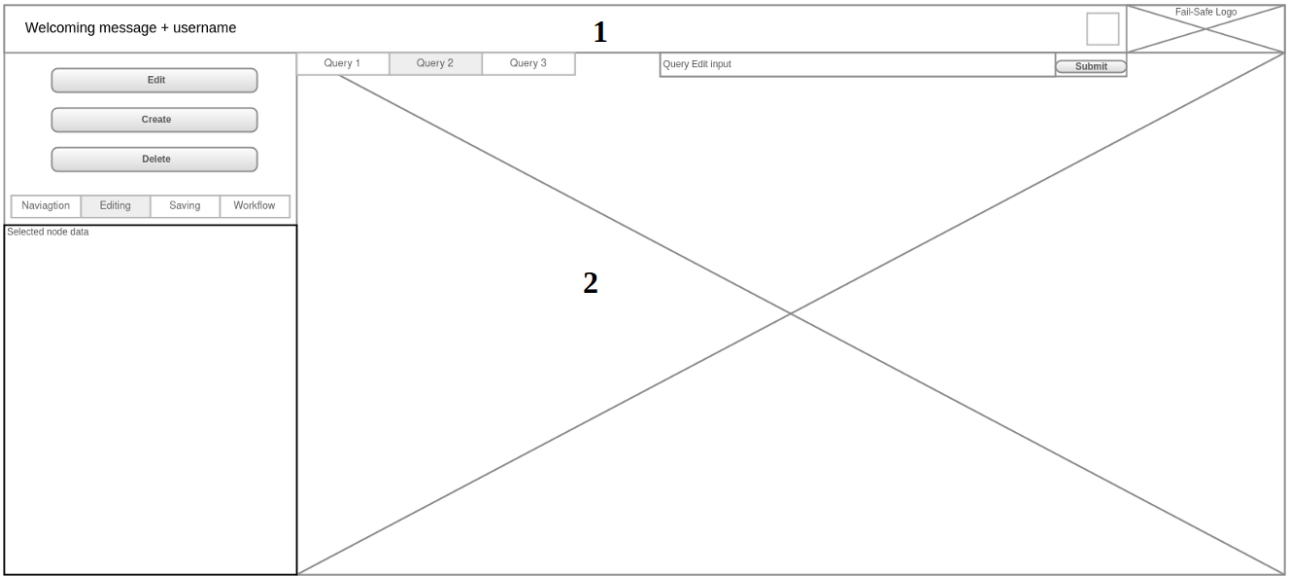


Figure 29: First approved version of UI wireframe (the Footer is not present)

During development, the UI was modeled and wire-framed in order to find a good solution for interface elements placement. The early mockups depicted only positions of the core elements and their functions:

The first models of the app were representing the structure of the page. The initial plan was to split the page into three parts [Fig.28]:

1. Top part (Header), which included the logo, the interactive user panel with possibility to expand and collapse it and open additional sub-panels for profile management;
2. Middle part (App working area), which contained the main working area and side-panels for tools and information;
3. Bottom part (Footer), which contained navigation, copyrights and links. Most of the initial philosophy was later implemented and optimized, but significant changes were applied to the Middle and Top part in order to maximize the working area.

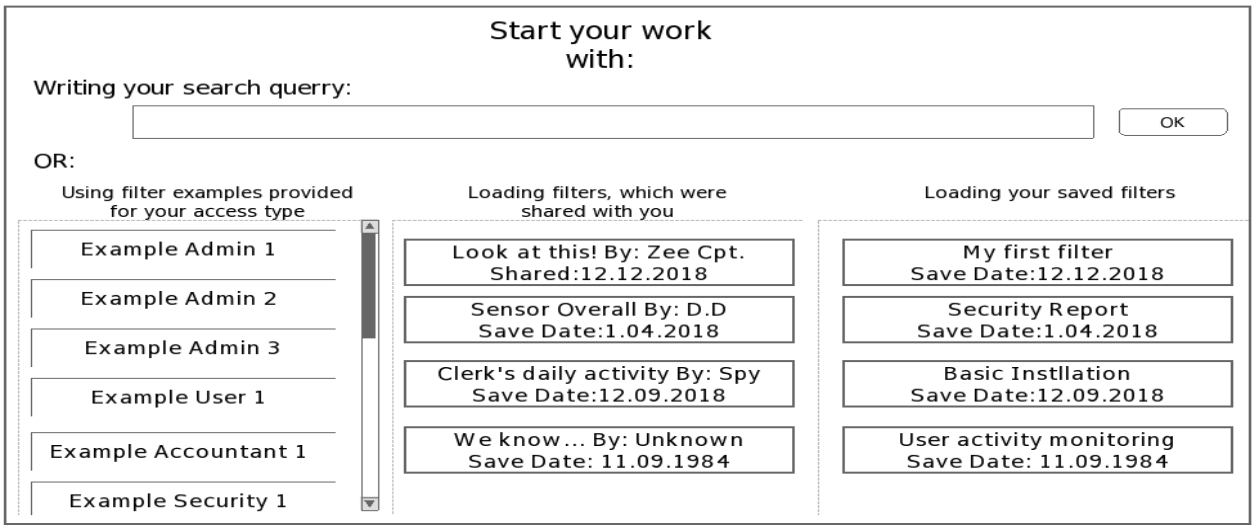


Figure 30: Example of Starting panel wireframe.

Also, several designs of the tools sub-panels were done, including the login panel, the starting panel, saving and sharing panels (removed in the current version), wizard and editing panels, user panels [Fig.29]. Most of this tools were later configured as modal overlay windows, rendering above the main app, when the user needs them.

Figure 31: HTML+CSS naming Bootstrap-based layout planning [Optimized and implemented by Ronja Kyrölä]

The later refinements in the model of the main UI were pointing towards the increase of the visualization area. This tendency can be seen through the whole development process, as more optimizations were done in order to enlarge the working area without losing the overall functionality. Another important model created was the HTML structure model. The importance of

this model is related to the technology used for creating and manipulating the web app structure – Bootstrap, which will be described in the next section. The initial design was overutilizing the Bootstrap technology, making the page structure overloaded with unneeded complexity and meaningless elements. The clarification of the structure, visualization and thoughtful planning was proven to be an effective solution to this problem [Fig.30].
As it can be seen in Figure 30, the main structure of the page remain the same from the early versions, however, enhanced by Bootstrap, it achieved better modularity, ergonomics and responsiveness.

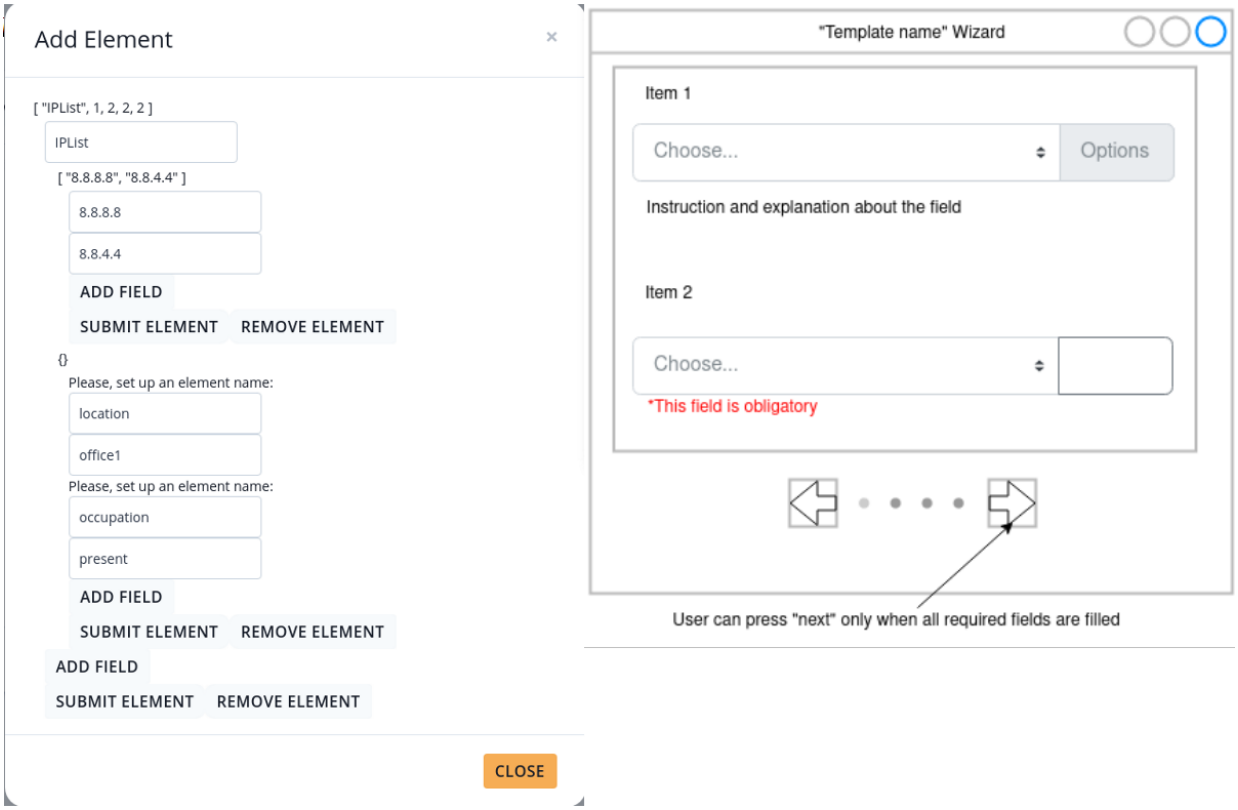


Figure 32: Two different approaches in solving the graph creation functionality

3.5.2.3 Snapshots and important design decisions

Early stages
The start of the project is covered within milestones 1 and 2. In the early stages of the project the UI looked very similar to its wireframe model [Fig. 28]. The most important design decisions were:

- Usage of webpack technology for managing and building the application.
- As mentioned in the section 3.5.2.1, the full Redux-based philosophy in the project structure was proven to be not cost-effective and did not provide enough benefits to the system to overcome the complications it brings. Therefore, a combined approach was designed, where most of the small interactions and logic were done inside the components, storing the all needed data in inner state storage, while the global data, such as data from back-end, state machine operations and some variables, needed for many components operations were moved to the Redux Store with all corresponding logic. This provided minimal needed code overhead, while still allowing to use most of the Redux benefits.
- Usage of browser session storage for storing session token and user data. In order to maintain the consistency of user experience, after authentication and receiving the session token and user data, they were stored in the local storage. After reloading the page, if the token is still valid, user can skip the log-in process and go straight ahead to the work.

Main functionality implementation

During milestones 3 and 4 the main functionality of the application was developed, along with the UI improvements. After several stages of refinement, the UI was optimized to increase the space of the working area. This was done by introducing several design decisions:

- Collapsible user panel. The static user-panel above the working area was taking too much of the workspace and distracted user, so it was changed. Most of the time the user panel was collapsed into a thin row above the working area, but in case the user needed several user-profile functionality, it was able to expand.
- Modular, categorized toolbar. The toolbar from the first prototypes was taking too much of important work space. Too many buttons in one place were dispersing the focus of the user and provided unnecessary confusion. It was collapsed into a set of several grouped buttons, where the top persistent group was defining the category, and the bottom, dynamic group was changing according the category of tools chosen. The categories were formed logically and included: window management, graph management, post processing and settings. Moreover, big text buttons were replaced by compact icons.
- Window approach. To implement possibility of working with several graphs at the same time, window logic was created. After querying several graphs from back-end, the user was able to switch between them via tabs.

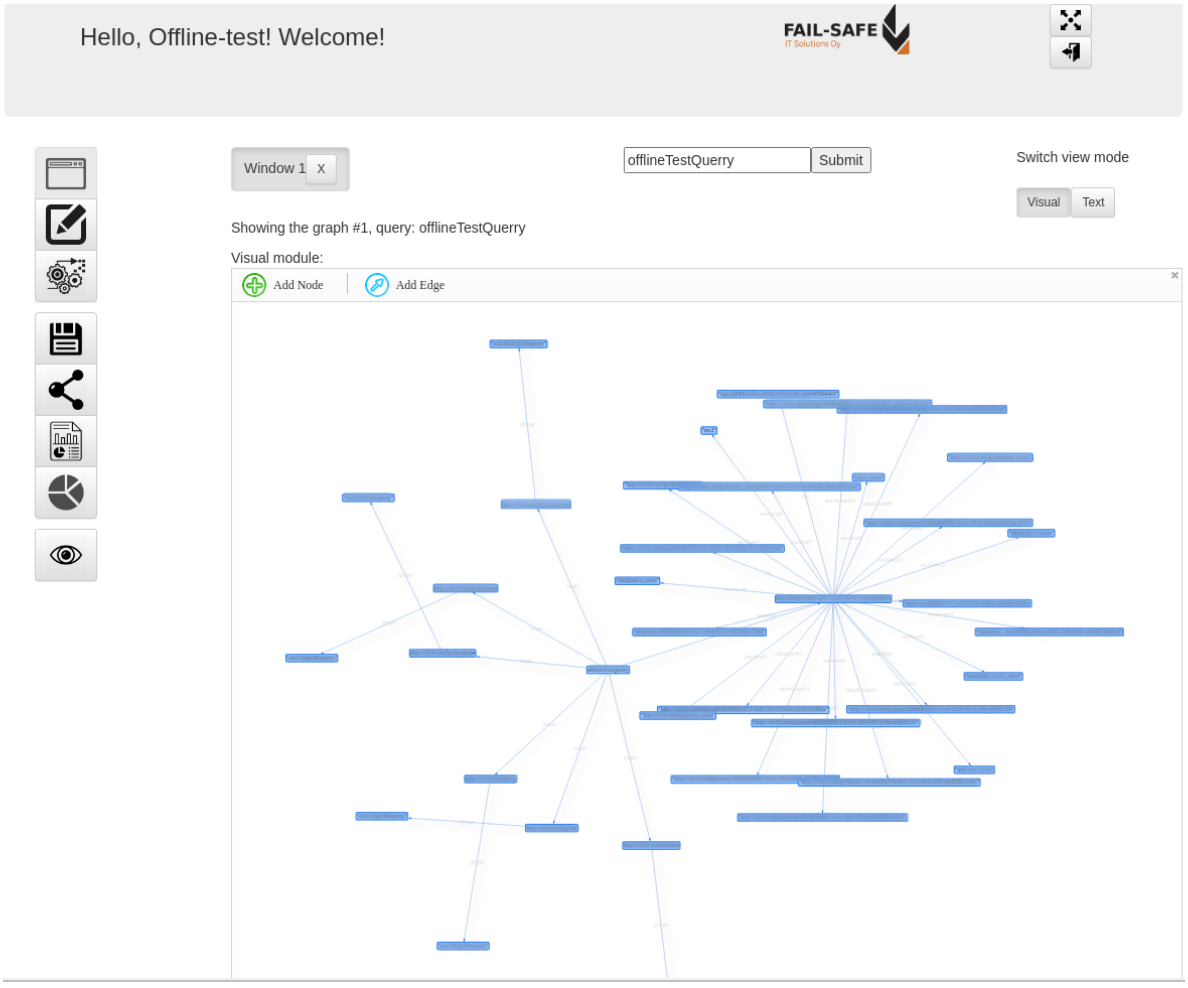


Figure 33: SPQ-02 UI in the middle stage of development

Several other important design decisions that were planned in milestone 4 were done during the middle stage of the development:

- The decision on which visualization library to use (described in the section 3.3) was done during this stage.
- URL-based graph ID storing. While the app is a single-page application, the URL line is not used for navigation between documents, like in several other websites. Instead, there was a decision to use URL for storing the graph ID for sharing and reloading. However, the graph ID is a considerably long string with a repeating starting patterns, therefore in the URL it was decided to shorten them up, leaving only the unique part of the ID. The repeating pattern is configurable and can be set in the configuration files of project. This feature allowed user to copy&paste the URL link of the working app anywhere. After pasting it into the browser and authentication, it would automatically open up the working area and query the graphs from server. In case the session is already stored and valid (for example if user reloads the page), it allows to automatically skip the authentication part and proceed straight to work.
- Additional view modes. For the professionals, who understand graph logic and syntax, it was made possible to view the graph in the plain text, almost the same way it is transmitted. However, the text was formatted and highlighted in order to make it more pleasant to read.
- Implementation of the Bootstrap technology in the form of React-Bootstrap. The Bootstrap, as it was mentioned before, is a popular CSS framework for creating modular and responsive web designs. In this project it was implemented in the form of React-Bootstrap – a collection of interactive React components, which allows easy and fast implementation of all Boostrap features in the project. It contained all basic web app elements, such as buttons, forms, animated sliders, dropdowns, popups and overlays. But the most important - a Grid system based on flexbox (aka. Flexible Box Module) technology [29], which allows easy-maintainable responsive design that adapts to the user screen size automatically and allows a very precise element positioning on the web page. The main idea of this technology is that the page is split on a configurable amount of rows, each including 12 columns of dynamically calculated size. Each set of columns can be assigned to define a size of the component and can contain another Grid or sub-elements. This design doesn’t only satisfy the responsiveness in a cost-effective way of development, but also allow all means of modularity, which is used a lot by React.
- Implementation of modal overlays. With Bootstrap it became possible to create functional overlays for React components, which allowed to simplify and optimize the UI. Multiple space-consuming features were moved into pop-upping windows, which triggered by corresponding button clicks. This included all complex interactions with graph data (modifying/creating graph elements), saving and sharing features, opening of new windows etc.
- Implementation of tooltips. Another Bootstrap feature, which was implemented and heavily utilized. It allowed to add tooltips for any UI elements, which materializes near the mouse pointer while hovering over the needed element. This provides help to the user and should considerably increase UI clarity.
- The feature related to the querying graph data with user-defined queries was considered unimportant to the first release and was removed.
- The graph parsing and navigating was developed during this stage. The custom node naming system was created for easier node tracking and debugging.

Figure 34: Overlay panel containing the Element Editing tool

During this stage, the challenge from the programming and algorithm creation side was to create a system of conversion, data manipulation and interaction with the visualized graphs. The chosen library provided a simple way for visual rearrangement and navigation in the data structure, but technically it was rather difficult to implement meaningful data manipulations.

The first data manipulation tool was created in form of graph redaction. The visualization gave the user a possibility to select any graph element, after which its ID was stored and user can open an edit panel, where the element could be modified at will [Fig. 33].

This method was considered good for fast error correction, as it allowed both changing the whole structure of the graph, if the root node was selected, and the precise visual redaction, when user was selecting the leaf node, which needed redaction. However, this technique required skills and knowledge in linked data and was not simple enough for majority of the clients.

Further development

The late stages of project development started from massive look&feel CSS overhaul mostly designed by Ronja Kyrölä. It included optimization of the HTML Grid structure, reworked color palette and minor rearranging of the interaction elements (Milestone 5).

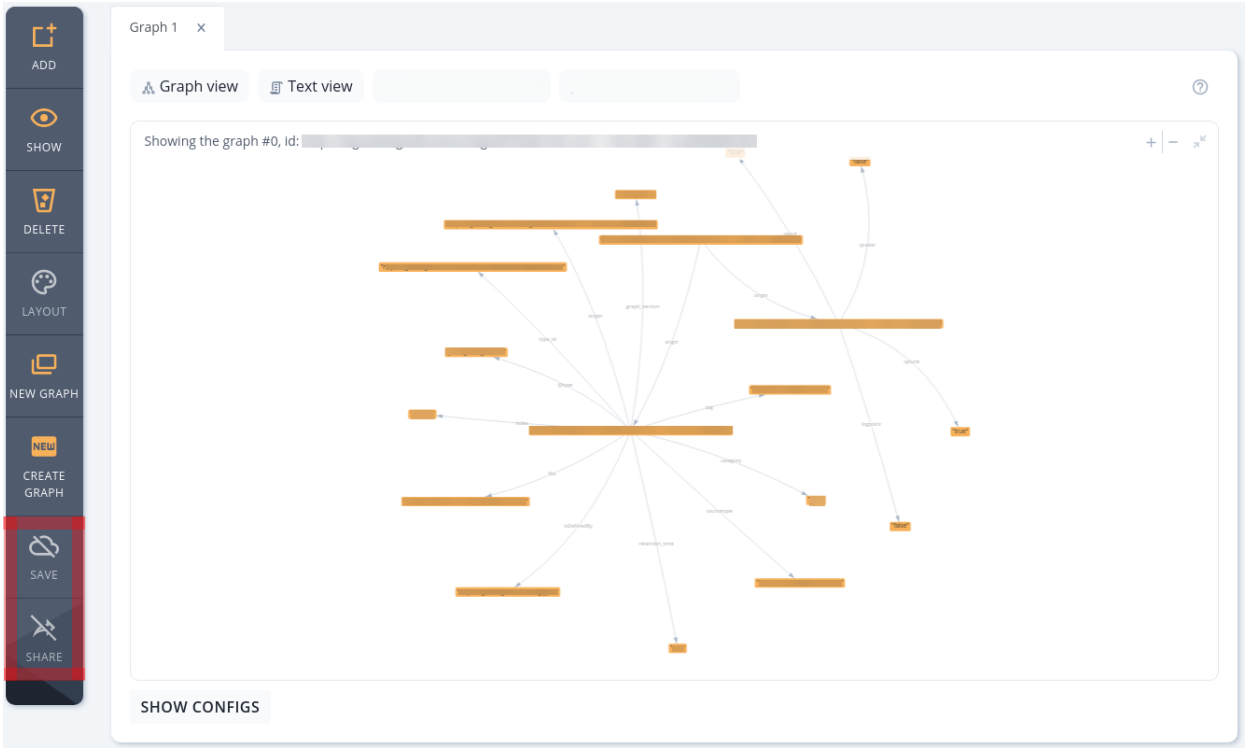


Figure 35: UI design after major CSS improvement

The color palette from this stage started to match the common color palette designed for all Fail-Safe products. If affected everything from buttons to visualizations. The tools panel was rearranged several times during last stage, as several features were removed from the first release. Additional functionality planned in the Milestone 6 was also added in this period.
Important design decisions of this stage:

- The collapsible top panel was reduced to a simple thin panel with company Logo and dropdown button, which provided access to all needed user-profile and administrator features via overlay panels [Fig 35]. The project owner, development team and project reviewers evaluated that as a major improvement, which provided not only increase in space of the working area, but also was more user-friendly and obvious.
- Implementation of help panels. The increase in complexity of the UI created a need for some form of tutoring the new user around the app. While tooltips provided the minimum of the information, a more sophisticated way of teaching the basics of the UI was required and was implemented in the way providing the user with modular tutorial overlay panel. The panel was done in the form of a slider, with several non-interactive pages, including texts and pictures, which explain to the user the possibilities of the program.
- The button design was redone, switching from simple icons to icons with text. It was additionally enhanced with tooltips. Moreover, all icons were replaced from separate image assets by a dedicated icon library, which provided a consistent styling and standardized implementation and management.
- The first approach to the problem of new graph creation was to allow user a full possibility of creating any graph user desire. This was proven to be over-complicated both in UI perspective and coding, because with broad possibilities it was difficult to predict, which actions user would perform, and how to mitigate potential errors, both in code and in user input. Therefore, a new approach was taken, limiting the user freedom in graph creation to the scenario-based procedure, also known as Wizard [Fig. 31]. The wizard would conduct user through a set of predefined interactive forms, where user is able to input all required

fields, checked for validity of the input and proceeded further. In the end of the procedure, all user input is automatically gathered and formatted into a correct graph instance, after which the user can review, modify and ultimately send the new graph to the back-end for storing. After several iterations of prototyping and testing, this solution was found faster and more efficient, also providing better UX in overall, comparing to the free-form creation.

- New visualization library Cytoscape.js was introduced and taken into testing.
- New feedback gathering feature was introduced as an overlay panel. It allows user to submit a categorized feedback, automatically forming the feedback report, based on if it is a commentary on user experience or notice of malfunctioning. It also provides links to other contact information and company resources.
- Implementation of modular server-controlled panel design, supported by layout library.

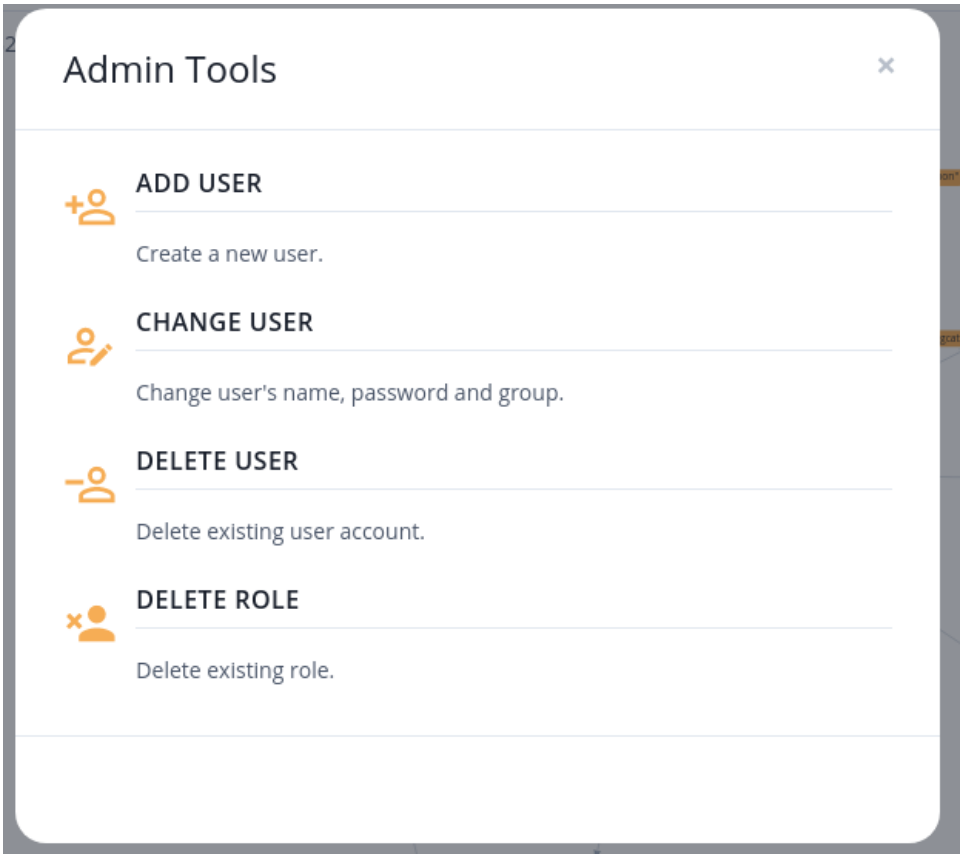


Figure 36: A new version of Administration Panel overlay

4. Results

4.1 Overview on the work done

After almost two years of development, the project has reached the first release point. However, this web app is only a part of a bigger project and the whole development and release cycle is synchronized to meet requirements on both sides. Therefore it cannot be immediately released and tested whenever it is ready.

Figure 37: Most recent look of the SPQ-02 UI

The results of the continuous research and development are considered significant, as the project matured through all development stages from early planning and prototyping to the almost finished design, providing a solid user experience (based on users' feedbacks) and performing actions it is required to do.
The basic functionality required from the project is considered fulfilled and the development status is reaching beyond the minimal viable product. Now the project continues the development, passing into the next stage, dedicated to expanding and broadening application area of the product. The backlog was expanded with new planned features after the previous backlog reached 90% of completion.

4.2 Assumptions and results correlations

As expected, the problem of graph data visualization and interaction have proven to be a challenging task. While the way of visualization is rather straight-forwarded and is solved in a similar way shared alike between different unconnected solutions, the way of interacting with graph data is dictated by the type of the data, its purpose and situational requirements.
Despite that, there are several consistent points that can be made on how user experience should be designed, when there is a problem of interactions of Linked Data:

- The visualization of the data takes the most natural form of a graph, with nodes, representing atomic segments of the data, connected to each other with edges, representing relations between these data segments.

- The graph segments should have expansion/collapsing mechanics, which helps to organize the working space, as well as allowing to switch between simplified general view and more detailed view, preferably controlled by the user in complexity.
- The graph visualization design should from one side informative, trying to represent as much information in the compressed visual format as possible, and from another hand it should be simple for user to understand and not keeping in mind all designations. This can be achieved by diversification of the elements via color and shape, but this process should be moderated to prevent excessive usage and overcomplication. This depends on user-stories and actual data, which will be represented.
- In is beneficial to add mechanics for automatic elements grouping and positioning, so that user already can start working with convenient workspace, prevent unnecessary time spent on workspace optimization and errors that could be caused by overlapping elements.
- The interface should not be overly complicated with controls, button and other UI elements, their amount should be reduced to a possible minimum. The main priority should be given to the visualization area, which should occupy the most of the app space. At the same time, there should be a way to view or interact with any data element of user choice in details, which brings up a complex task to create dynamic UI elements, intuitive enough to use, plentiful to provide enough information to the user, but limited enough to not becoming a burden.
    - A good practice in solving this problem was using less text present permanently on the screen and replaced with intuitive icons. When it is impossible to evade text, present it to the user in a form of pop-ups.

The practical implementation of the project shows the importance of environment and technology stack choice in solving specific problems. While web-based application development environment introduced several new challenges, such as security, need for platform-compatibility and responsiveness of the design, it also allowed this part of the project to be interface-centered with rapid prototyping and modeling. It also simplified the way of deployment and partially testing the prototypes.

Careful approach to choosing the core library for graph visualization proved to be of extreme importance. Even though the first implemented library Vis.js was successfully performing the basic tasks as a proof-of-concept, the implementation of another library Cytoscape.js was considered more beneficial to the project. The reimplementation of the visualization library is a process that takes considerable amount of resources, there should be avoided in further development, therefore this findings and detailed analysis of these two libraries have straight value.

I also managed to find a rather uncommon solution for the graph management, which was not seen in the examples that were previously researched, implementing three different ways of managing linked data: node editing, free graph tree building and wizard interaction. The last one is considered to be the most promising, as it simplifies and speeds up the process of graph creation [Fig 37]. However, it is still too early to make a solid claim on its superiority, as it requires more testing.

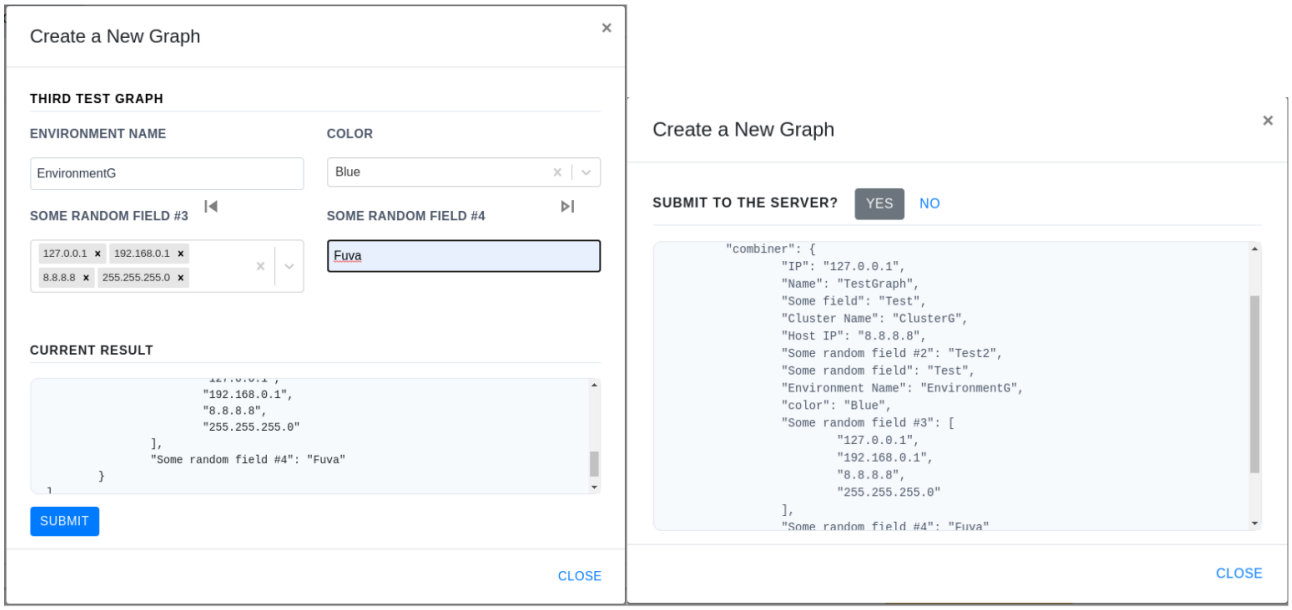


Figure 38: Working wizard prototype in two different stages - creation and review.

4.3 Product analysis

4.3.1 Testing

In this project the unit testing was performed by library called “jest”, but the code coverage is insignificant, testing only the complex logic elements, such as a parsing algorythm. No complex UI automation testing is used as well and the most of the UI testing is performed manually. This goes through the normal developing routine, when new elements are introduced to the UI, after that they go through the manual user-testing on the local environment and after everything runs without errors, the code is uploaded to the source control, where other developers can load and test on the demand. There is no mandatory cross-developer testing and code review in the project development routine established.

The implementation of usability testing in the project took form of a combination of several manually designed surveys. The surveys are based on several popular templates available online [30], combined and modified to match the list of requirements to the project, as well as prioritization of the requirements. Also, a procedure for conducting the testing was developed and is now in the process of refinement. The procedure include several stages, each with a different purpose and implementing different approaches.

The testing procedure for SPQ-02 include 4 stages:

- Introduction to the project and preparation.
- Free testing session 1, following with the survey #1.
- Scenario-based session, where user receives one of the scenarios from the collection, each related to a separate part of the project and followed with scenario-specific survey. Tester can afterwards volunteer for another scenario.
- Free testing session 2, following with the survey #2.

The reasons for this structure is an intention for capturing the learning curve and evolution of user experience during the system exploration. In simple words, we want user to first try it on their own and get an initial impression on the system. This will show how intuitive and friendly the system is, whether it creates a positive response in the user from first moments. Then we put user into a scripted test, where everything is clear and steps are easy, so the user performs a typical user task, both getting an impression on the fluency of the process and learning the system a bit more. After that, a user already a bit familiar with the system is allowed to free roam a bit more to test out any

ideas that came during the scripted test or just to try something not mentioned. This will create a more “mundane” impression, pointing out more thoughtful on the deep aspects of the UI.
Comparing the first and the last replies, as they contain similar questions, we can extrapolate the tendency of user-interface relations development, which defines the final UX user will have after continuous usage.
The actual testing was performed with a less amount of subjects that was anticipated, however, the project still didn’t get an official release, so the major testing and data gathering is only planned in close future.
With the results collected on the moment, it is possible to do first assumptions and evaluate an overall performance. To be noted, the testing was done on the first working build running in the production environment, which differs drastically from development environment in many ways. Some errors and malfunctions were noticed related to the specifics of the new environment, to the back-end configuration as well as to the software errors. While this also allows to do a error-testing and session was very helpful in finding several hidden malfunctions, this affected the UX part of the results, which means that they are not perfectly corresponding to the real situation and the overall UX score should be expected higher, once the testing would be conducted in (mostly)error-free system.
The surveys created for the procedure can be found in the appendix section. After summarizing the testing results, an overall picture was made and several potential design problems, along with technical problems were found.
In the summary, the overall picture is quite decent for the first testing attempts. The average UX score for first testing session is around 6, which is normal. The interesting point found was several people answered on “time to learn” question with very low score (2), while some other put there very high score (9-10). This can indicate on either there is some initial troubles with learning the system, or this particular survey question was done wrong and people misunderstood it. It also can point out that there is a group of users, which is not expected by our user-portrait assumptions and might need attention. However, against the last version goes that the intuitivity of the interaction for the first session is marked above middle (7) with most of the respondents, and as a support to the first version, in the second survey some people mentioned that several parts of first survey were confusing. This might also benefit from the fact that numerous respondents recognized familiar interaction patterns (marked as 8 by majority). In overall, testers didn’t need help during the first testing sessions and the terminology of the texts in UI was found clear enough (6). The UI was found effective enough (6) and fast-responding (8) which also benefit overall UX. Several users remarked clean and pleasant-looking UI in positive sides, while others found several elements alignment issues. These issues were mostly production artifacts, not present in the developer version, and will be fixed soon. One good point worth to mention was that footer of the web page and company logo were not linked to the company main website. This is taken to the next iteration change.
The second session shown a positive increase in the learning. While overall experience somehow didn’t change, the learning time, satisfaction and speed of navigation raised from previous on 1-2 points in average. This means that application has some learning curve, which should be overcome in order to achieve the maximum UX. Other points extracted from survey analysis were that most of the users liked the color balance of the app, and that amount of new information user need to learn is manageable and no-one found it to be too much. Some criticism was made about app not allowing to revert actions performed in it and it was also taken into the next iteration plan as a high-priority improvement.

4.3.2 Feedback

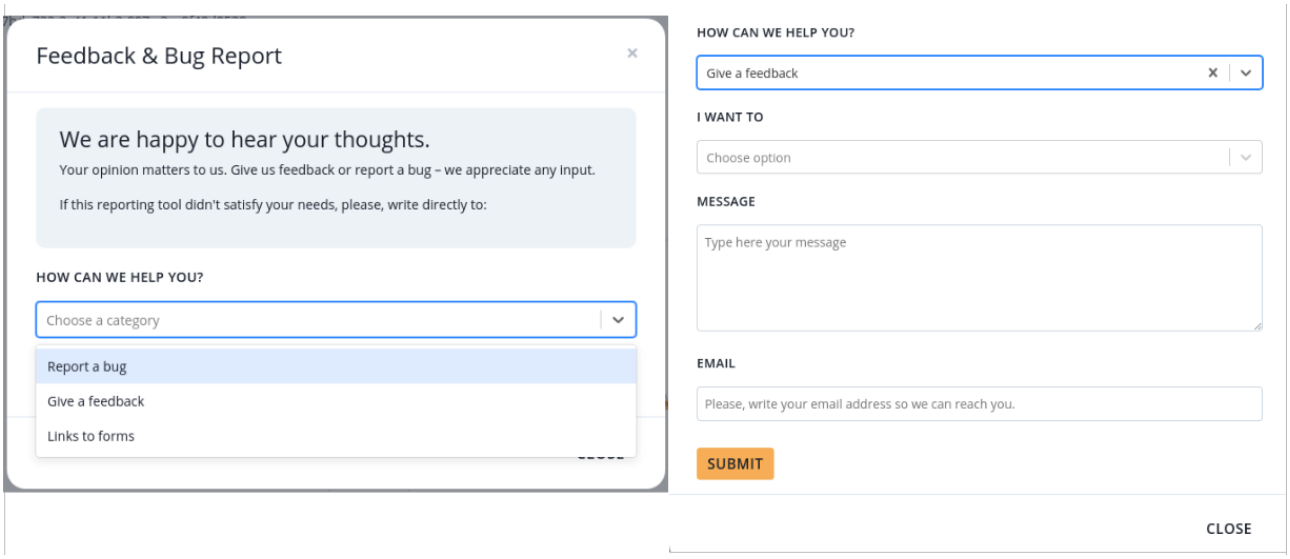


Figure 39: Feedback capturing panel in two different states

User feedback is an important part of project's continuous improvement and development. It is necessary to give users a way to share their opinion and react on it actively. However, this must be done on voluntary basis, and it must not affect user privacy in any way. It is especially true after new regulations, GDPR rules, which are now used in the EU zone. [31]
This means that service must not collect any user data aside the one it clearly states and must have user permission on collecting it.
In the last iterations of SPQ-02, a feedback feature was developed to simplify communication between client and service provider. However, to eliminate complications and maintain user's security, the feature does not do any automatic data sending, only helping the user to fill necessary information and form a contact document for easy sending and processing [Fig. 38]. It allows to categorize subject, give it a priority and provides all needed information to reach company. After completing the form, user will download a feedback in a form of a file, which can be later sent to Fail-Safe (or the service provider) via contact information. The feature has additional development plans to include debug console output and system information, if user explicitly states that it should be included.

4.4 Unsolved problems and plans for future development

During the development process most of the practical tasks were solved. However, there is still several directions in which project can be improved and developed further.. Here is what can be done next:

- Expand the interaction capabilities of the visualization library. The new library Cytoscape.js is in testing for now, but it is assumed that it provides more possibilities for user-interactions and therefore new ways of interacting with graph data might be found and implemented in the future.
- Graph data conversion and alternative ways of visualization. While in the project the most used form of plain-text linked data was JSON-LD and the type of visualization is force-simulated graph tree, it doesn't mean it is the only way of solving this problem. Other forms of data might be applied, which may need other approaches for visualization and manipulation with that data.

- Perform a large-scale open testing with actual product users, not testing groups and start gathering a real feedback from the daily usage. This data will help to continuously improve the project further.

The recent testing opened up several problems occurring only in the production mode:

- URL handling is not working the same way it is working in the development version.
- Setting a button for the graph view is missing.
- Some elements misalignment such as in graph creation.
- Protocol usage during login is not secure enough and should be reworked.
- Logo and footer need to lead to the company website.
- User update does not change the local value.
- Graphs naming is confusing (sadly cannot be fixed as it is a part of the design)
- Search function is deprecated and should be removed.
- The + - in graph view was blocked by some other button (basically by the same duplicated buttons.)

These problems are now in the highest priority and will be solved soon.
The backlog for the second iteration is in development. The vision of the second version is already quite clear and it includes implementation or completion of several features:

- In-system user-to-user data communication and storing, which will be represented by completing saving and sharing features.
- Creation of UI visual presets. This will allow user a rapid switch between types of UI and between different sets of settings for graph visualization.
- Graph depth-level view. A new approach in handling visual graph data, by organizing it in levels and hiding/expanding on demand of user.
- Localization. Project owner stated that at some point it would be necessary to localize the UI, adding several languages to switch between. In this type of interactive web-UI, it is possible to do it seamlessly, via language sheets. While in old-fashioned page-based websites it usually requires to create a clone of the page translated, in React environment, it is possible to dynamically update text on the page from the preset configuration files. This will require additional coding and refactoring, but is proven to be more pleasant for the user and efficient in general.
- Server control on various UI layouts. It is already working in the current version, but require more back-end support.
- Data extraction and statistics.

As already mentioned, SPQ-02 is only a part of a bigger project, so its development is influenced by the trends of the major project development, and various things can be changed during time, reacting on changes to other parts of the project.

5. Conclusion

The increasing digitization of daily life, alongside with the raise in importance of such factors as the availability the world-wide-web connectivity and integration of the network services in the business processes and work routine shaped the world in a new way, forcing people to find new or re-evaluate old ways for data management. The amount of data produced and processed every day is already enormous and it continues to increase exponentially. While the old-fashioned ways of handling data structures is still being effective in many applications, new methods of storing and managing data are created, to satisfy new needs and applications. One of the rapidly increasing in popularity is a linked-data approach for description, classification, storing and analyzing the processes and big data. We learned that linked data can be represented as graphs and complex graph logic and mathematics can be applied to it in order to simplify the processing and allowing new possibilities in data handling. Taking it all in account, I came up to the problem of human – graph data interaction, which should be solved to enable efficient ways of working with mentioned data. I explored already existing solutions available and found similarities between them, as well as created a list of good and flawed design decisions that should be regarded in own practical project. Based on this research I created a view on the project implementations, influenced by the local environment specifications.

After the planning phase, the project development was started. It required not only knowledge and skills acquired during the university studies, but also a large amount of self-conducted research and experimentation. After several iterations, the project reached the first release state. The project then was tested in limited amount of test sessions and the test results gathered and analyzed. The review from the project owner and testing showed that the development of the project was rather successful and most of the requirements were met. The problem of visualization and interaction with the linked graph data proved to be challenging, but solvable and we developed and evaluated several practices for solving it. I also acquired a large amount of new skills in various technologies and fields during development. Moreover, testing results showed that it is still possible to develop the project further. The new iteration plan was designed for continuous improvement of the project and the interaction mechanics.

References

[1] Bordens Kenneth, S.; Abbott, B. (2008). Research design and methods: a process approach, tenth edition ISBN 978-1-259-84474-4
[2]Coffman, K. G., & Odlyzko, A. M. (2002). Growth of the Internet. In Optical fiber telecommunications IV-B (pp. 17-56). Academic Press.
[3] J.H. ter Bekke (1991). Semantic Data Modeling in Relational Environments. ISBN 90-900-4132-X
[4] Halpin, T., & Morgan, T. (2010). Information modeling and relational databases. Morgan Kaufmann.
[5] Oracle.com. A Relational Database Overview. https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html Accessed 10.02.2021
[6] Ambler, S. (2009). Relational Databases 101: Looking at the Whole Picture
[7]Ordonez, C., Song, I. Y., & Garcia-Alvarado, C. (2010, October). Relational versus non-relational database systems for data warehousing. In Proceedings of the ACM 13th international workshop on Data warehousing and OLAP (pp. 67-68).
[8] Grolinger, K., Higashino, W. A., Tiwari, A., & Capretz, M. A. (2013). Data management in cloud environments: NoSQL and NewSQL data stores. Journal of Cloud Computing: advances, systems and applications, 2(1), 1-24.
[9]Batra, S., & Tyagi, C. (2012). Comparative analysis of relational and graph databases. International Journal of Soft Computing and Engineering (IJSCE), 2(2), 509-512.
[10]Aalto University. RDF Grapher service. http://www.ldf.fi/service/rdf-grapher
[11] Meindertsma, J. (2019). What's the best RDF serialization format? https://ontola.io/blog/rdf-serialization-formats/ Accessed 26.11.2020
[12] Ontology definition by W3C. 2015. https://www.w3.org/standards/semanticweb/ontology
[13] Creately online service . Cinergix Pty Ltd. https://app.creately.com
[14] Kleppe, A.; Rensink, A. (2008). University of Twente. A Graph-Based Semantics for UML Class and Object Diagrams.
[15] Amit (2018). All You Need to Know About UML Diagrams: Types and 5+ Examples. https://tallyfy.com/uml-diagram/ Accessed 16.02.2021
[16]WebVOWL online service. http://vowl.visualdataweb.org/webvowl.html
[17]OWL Working Group. (2013). Web Ontology Language (OWL). https://www.w3.org/2001/sw/wiki/OWL
[18]LodLive project homepage. http://en.lodlive.it/
[19] European Central Bank Exchange Rates Common Procurement Vocabulary. linked.opendata.cz (Unavailable).
[20] Cytoscape project homepage. https://cytoscape.org/
[21]Node.js org, 2020. https://nodejs.org/en
[22] Chowdhury. T. (2020). What is Node JS. https://tamalweb.com/what-is-nodejs Accessed 29.04.2020
[23]W3School, What is NPM? https://www.w3schools.com/whatis/whatis_npm.asp Accessed 15.08.2020
[24] Webpack official website. https://webpack.js.org/
[25] Bostock M. (2019). D3. https://observablehq.com/@d3/force-directed-tree
[26]Vis.js official website. https://visjs.org/
[27] Several contributors, https://visjs.github.io/vis-network/examples/network/exampleApplications/nodeLegend.html Accessed 12.09.2020
[28]GeneMANIA. https://js.cytoscape.org/demos/colajs-graph
[29] MDN contributors 2020. Basic concepts of flexbox. https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout/Basic_Concepts_of_Flexbox Accessed 14.08.2020

[30]Perlman, G. (2018) User Interface Usability Evaluation with Web-Based Questionnaires https://garyperlman.com/quest/
[31]General Data Protection Regulation. https://gdpr-info.eu/
[32]Michailidis, G. (2008). Data visualization through their graph representations. In Handbook of data visualization (pp. 103-120). Springer, Berlin, Heidelberg.
[33]Angles, R., & Gutierrez, C. (2018). An introduction to graph data management. In Graph Data Management (pp. 1-32). Springer, Cham.
[34]Trendowicz, A., & Münch, J. (2009). Factors influencing software development productivity—state-of-the-art and industrial experiences. Advances in computers, 77, 185-241.
[35] Wieringa, R.J. (2014). Design Science Methodology for Information Systems and Software Engineering. Springer.
[36] March, S. T., Smith, G. F., (1995). Design and natural science research on information technology. Decision Support Systems.
[37] Wieringa, R.J. (2014). Introduction to Design Science Methodology. https://wwwhome.ewi.utwente.nl/~roelw/microtutorial.pdf

Appendix A
Code fragments.

Parser code fragments
This code fragments were used to parse row JSON-LD graph data, converting it into datasets, used in the visualization libraries:

```
export function RecursiveVisParser(data){
   if(data){
   var nodes = [{id:"query", label: data.query}];
   var connections = [];

   var recursion = (path, object, key)=>{
    var newPath = path +"|" + key;
    if(Array.isArray(object)){

      for(let n = 0; n<object.length;n++)
       {
                recursion(path, object[n], key +"#"+ n)
       }
    }else{
     if(whatIsIt(object) === "Object")
     {
        nodes.push(
                {id:newPath , label: JSON.stringify(key)}
        )
        connections.push (
                {id: "Con|" + newPath,from: path, to: newPath, label: "object"}
        )

        console.log("found JSON object: ",object);
        for (var nkey in object)
        {
                recursion(newPath, object[nkey], nkey)
        }
     }
        else
        {
                nodes.push(
                        {id:newPath , label: JSON.stringify(object)}
                )
                connections.push (
                {id: "Con|" + newPath,from: path, to: newPath, label: key}
                )
        }
}}
```

```
var path = "";
if(data.graph){
	for (let i= 0; i<data.graph.length; i++)
	{
		nodes.push(
			{id: "graph" + "#" + i, label: data.graph[i]["@id"] }
		);
		connections.push(
			{id: "Con|" + "graph"+ "#"+i,from:"query", to: "graph"+ "#"+ i, label:
"origin"}
		)
		path = "graph"+ "#" + i;
		for (var key in data.graph[i])
		{
			if(key !== "@id")
			{
				recursion(path, data.graph[i][key], key)
			}
		}
	}
console.log("Nodes: ",nodes);
console.log("Connections betweend nodes: ",connections);
var visData = {
	nodes: new vis.DataSet(nodes),
	edges: new vis.DataSet(connections)
}
console.log("Finalizing Vis Data: ", visData);
return(visData);
}}}
```

Text 1: Parser for Vis.js, converting to DOT language.

```
export function RecursiveCytoParser(data){
if(data)
{
	var nodes = [{data:{id:"query", label: data.query}}];
	var recursion = (path, object, key)=>
	{
		var newPath = path +"|" + key;
		if(Array.isArray(object))
		{
			for(let n = 0; n<object.length;n++)
			{recursion(path, object[n], key +"#"+ n)}
		}else{
			if(whatIsIt(object) === "Object")
			{
				nodes.push(
				{data:{id:newPath , label: JSON.stringify(key), shape:
CytoShapes[getRandomInt(0,22)]}}
				);
```

```
                                nodes.push (
                                {data: {id: "Con|" + newPath, source: path, target: newPath, label:
"object", arrow: ArrowShapes[getRandomInt(0,9)]}}
                                );
                                console.log("found JSON object: ",object);
                                for (var nkey in object)
                                {recursion(newPath, object[nkey], nkey);}
                        }else{
                                nodes.push(
                                {data:{id:newPath , label: JSON.stringify(object), shape:
CytoShapes[getRandomInt(0,22)]}}
                                );
                                nodes.push (
                                {data:{id: "Con|" + newPath, source: path, target: newPath, label:
key, arrow: ArrowShapes[getRandomInt(0,9)]}}
                                );
}}}
        var path = "";
        if(data.graph)
        {
                for (let i= 0; i<data.graph.length; i++)
                {
                        nodes.push(
                        {data:{id: "graph" + "#" + i, label: data.graph[i]["@id"] , shape:
CytoShapes[getRandomInt(0,22)]}}
                        );
                        nodes.push(
                        {data:{id: "Con|" + "graph"+ "#"+i,source:"query", target: "graph"+ "#"+ i,
label: "origin", arrow: ArrowShapes[getRandomInt(0,9)]}}
                        );
                        path = "graph"+ "#" + i;
                        for (var key in data.graph[i])
                        {
                                if(key !== "@id")
                                {recursion(path, data.graph[i][key], key);}
                        }
                }
        console.log("Nodes: ",nodes);
        return(nodes);
}}
```

Text 2: Parser for Cytoscape.js, converting to a singular element array.

Reverse lookup algorithm, finds and returns the object in dataset with selected path.

```
export function FindByPath(data, path){
	console.log("Searching the dataset: ", data, "with path: ", path)
	if(path ===""||path ==="query")
	{return data}
	var SelectedObj = data;
	var pathSplit = path.split("|");
	console.log("Path stages: ", pathSplit)
	try
	{
		pathSplit.forEach((item, index)=>
		{
			console.log("Path trace: ", SelectedObj)
			if(item.includes("#"))
			{
				var arrayTrace = item.split("#");
				SelectedObj = SelectedObj[arrayTrace[0]];
				for(var i = 1; i< arrayTrace.length; i++)
				{
					SelectedObj = SelectedObj[Number(arrayTrace[i])];
				}
			}else{
				if(index<pathSplit.length)
				{
				SelectedObj = SelectedObj[item];
				}
			}
		})
	}
	catch(e)
	{
		console.log("Problem in unparser:", e)
	}
	console.log("Selected object:", SelectedObj)
	return (SelectedObj);
}
```

Text 3: Reverse lookup algorithm

Appendix B
Diagrams

Class diagram

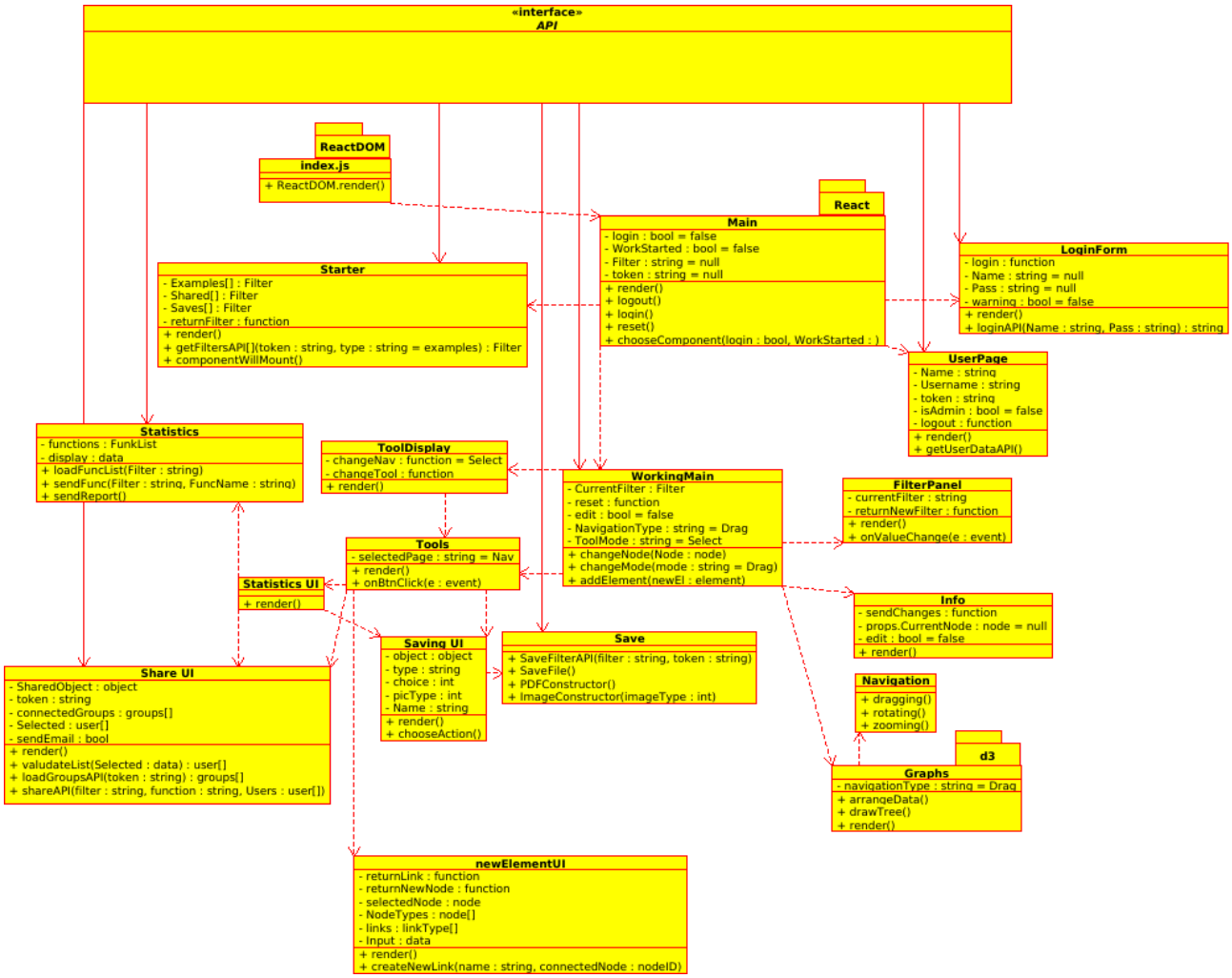
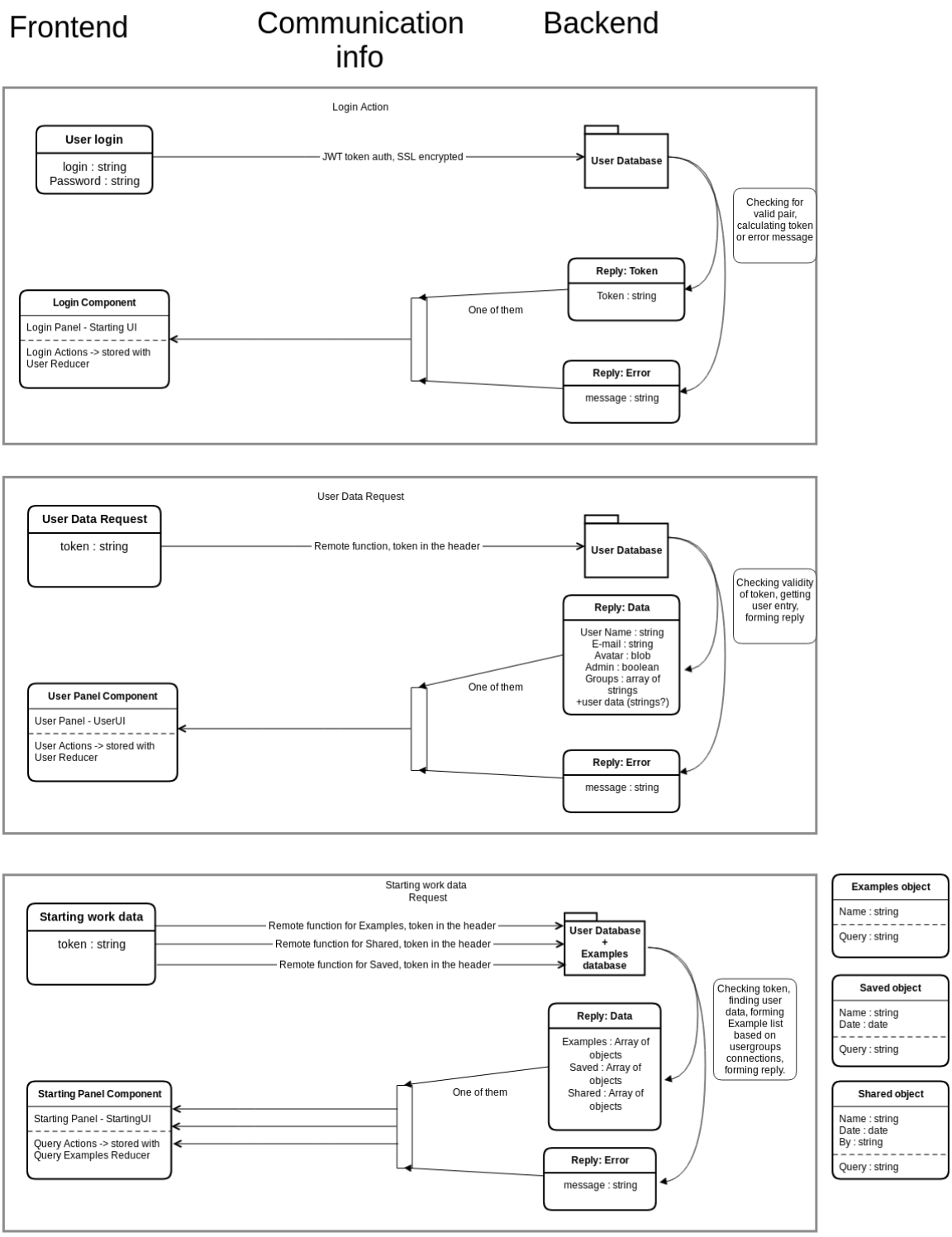


Diagram 1: An early system class diagram.

Data-flow diagram



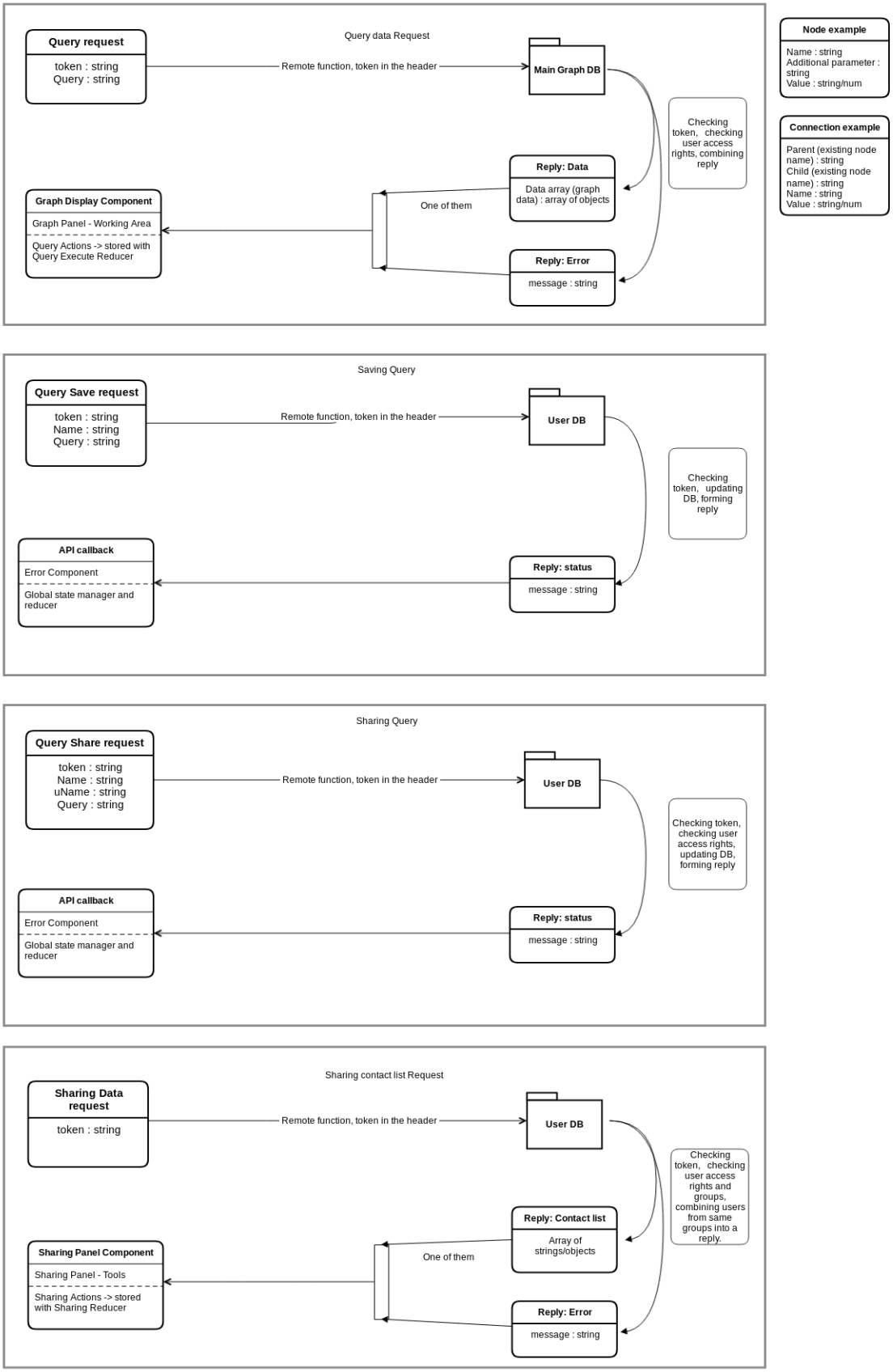


Diagram 2: Data-flow diagram describing front-end and back-end communications

Collection of Redux Entity-Relation graphs

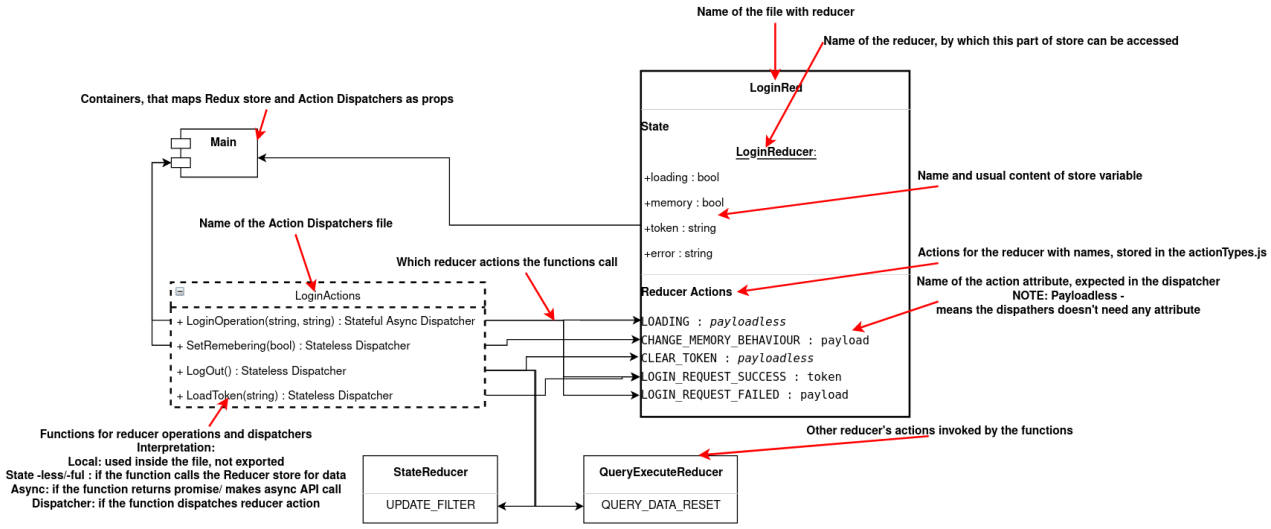


Diagram 3: Notation description for Entity-Relation graphs.

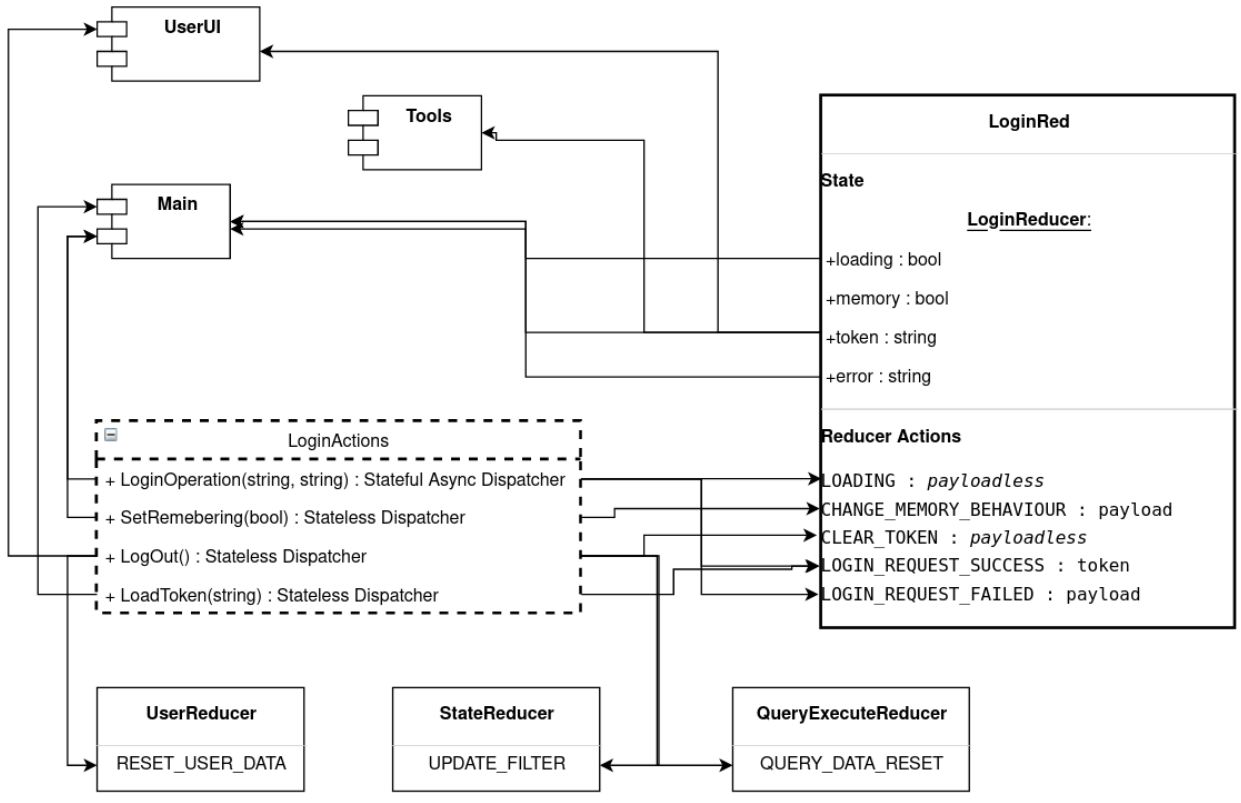


Diagram 4: Login feature entity relations

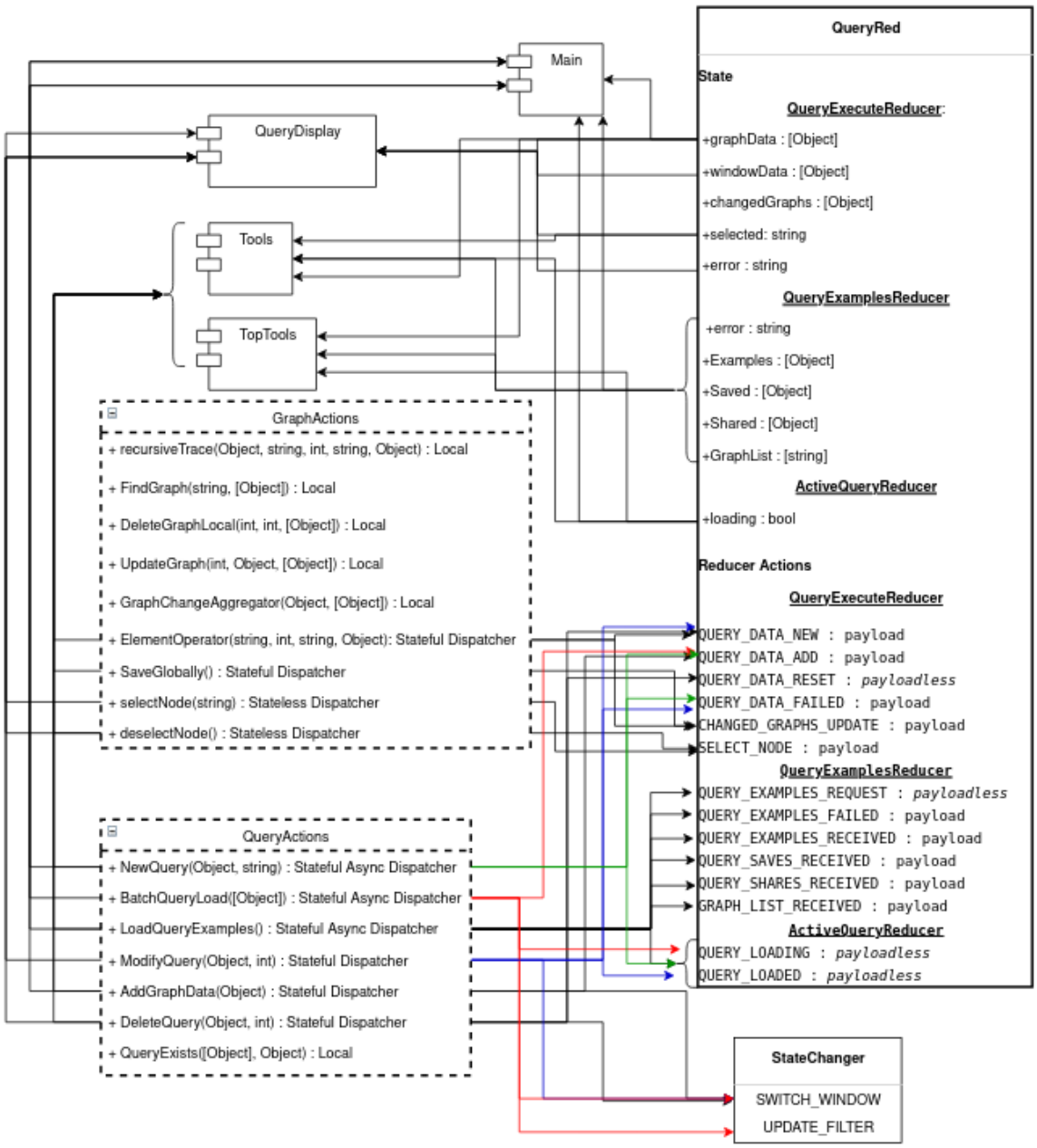


Diagram 5: Query-execution features entity relations

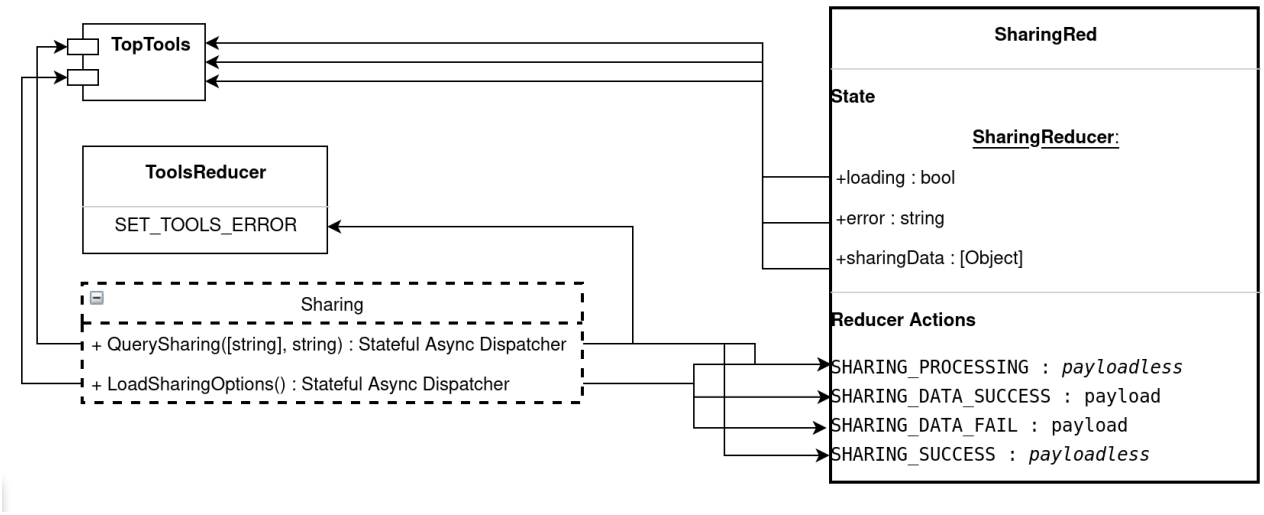


Diagram 6: Sharing feature entity relations

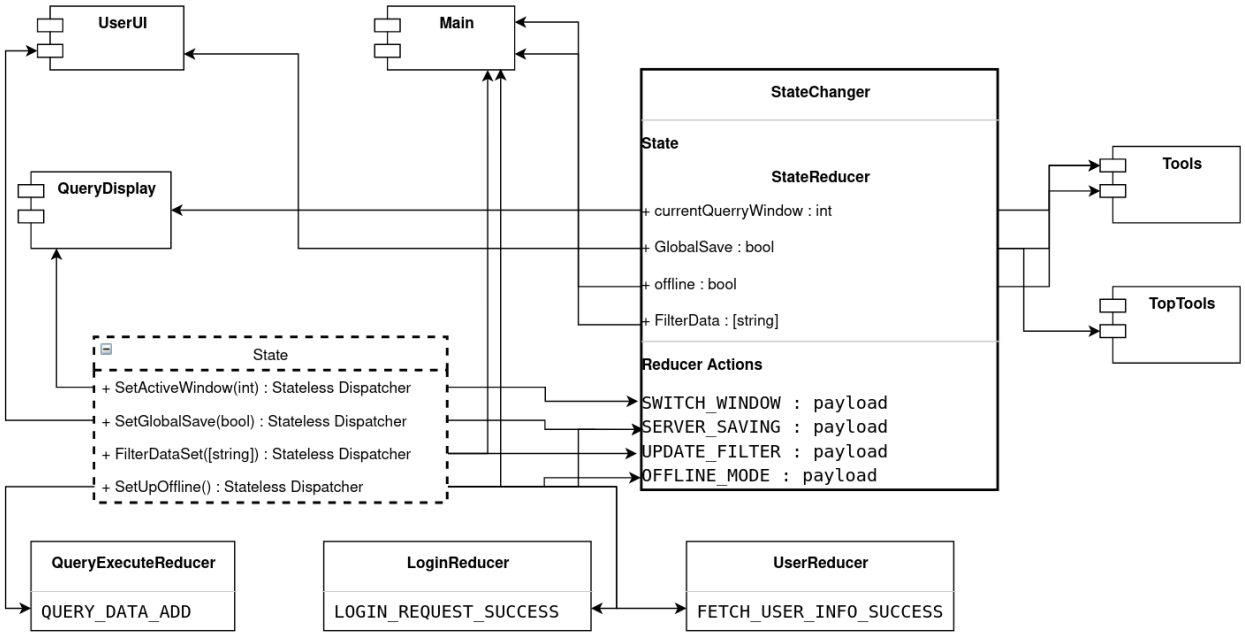


Diagram 7: Global state entity relations

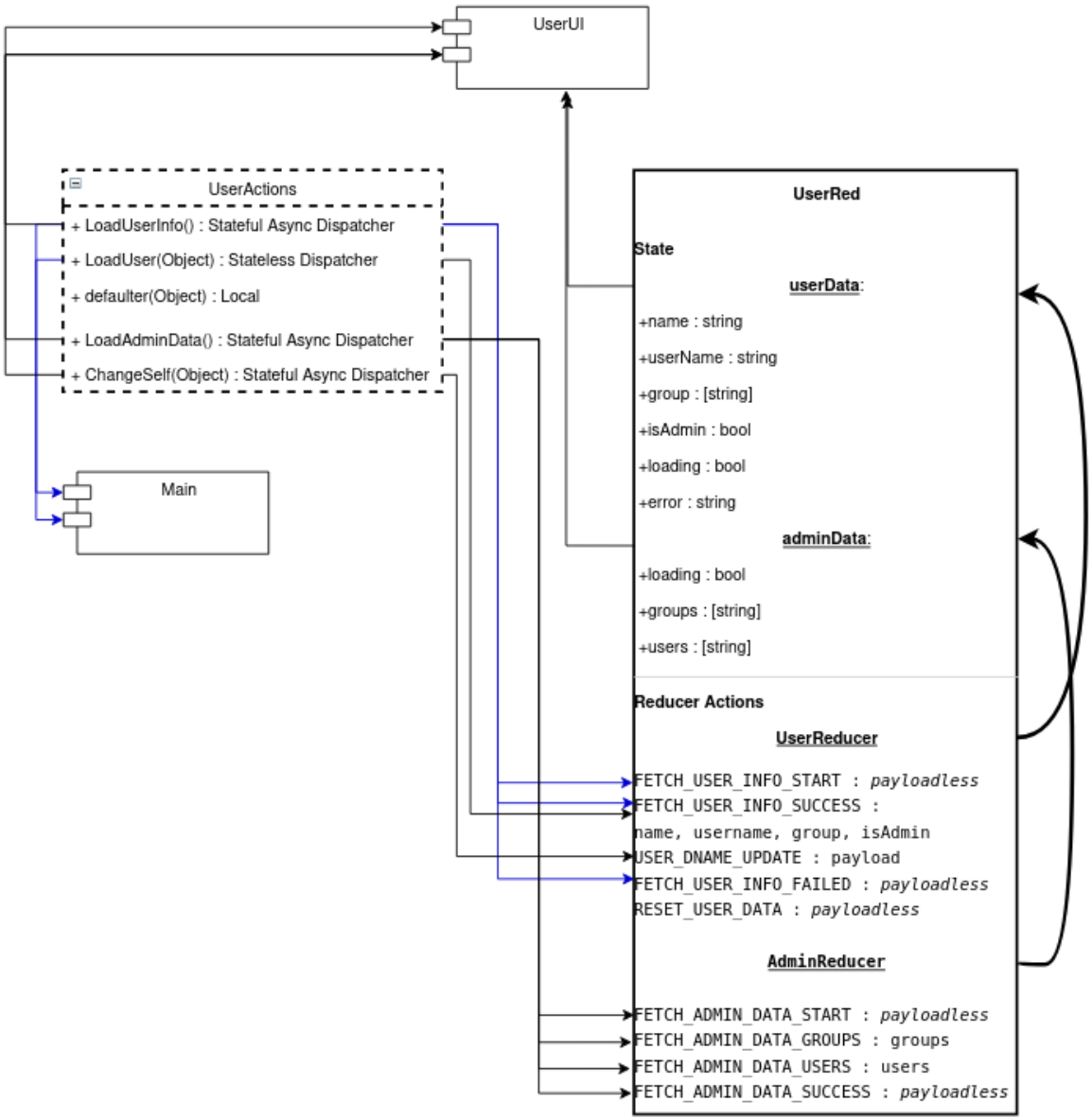


Diagram 8: User management feature entity relations

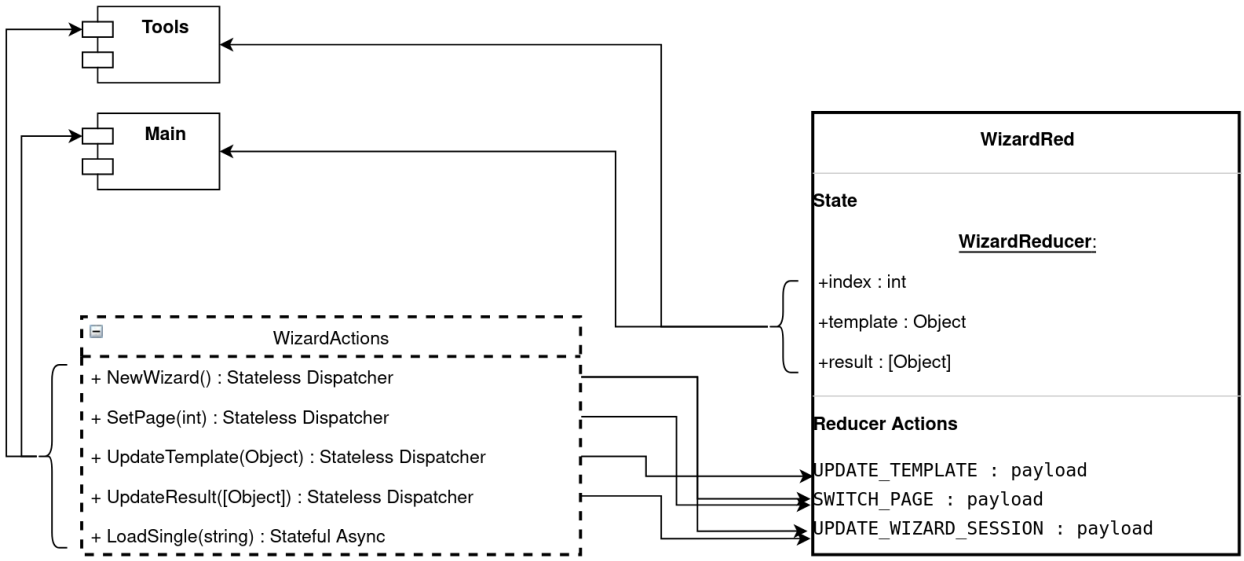


Diagram 9: Wizard feature entity relations

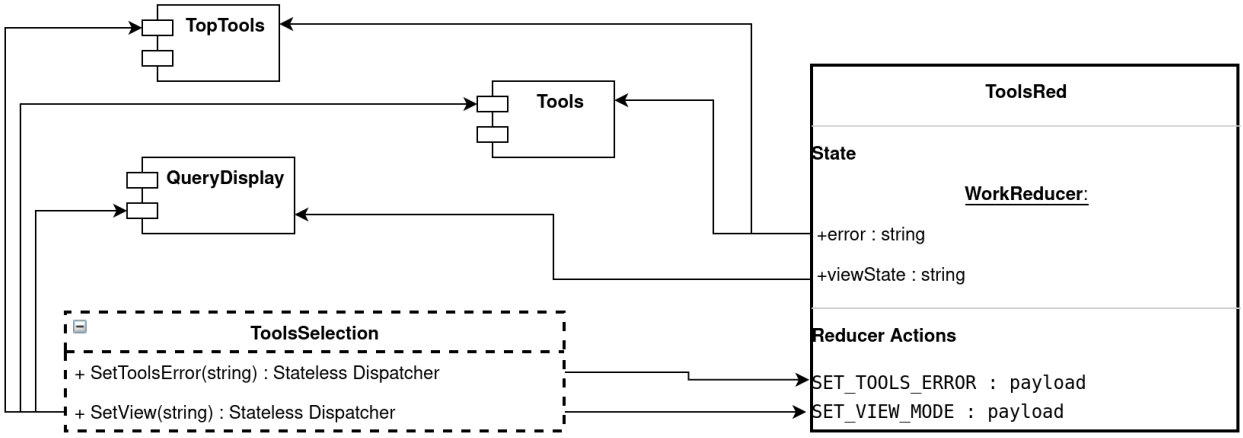


Diagram 10: Tools management feature entity relations

Appendix C

User testing procedures

Preparation instructions:

1. Make sure you are connected to service provisioning network.
2. Open your preferred browser. Make sure you are using recent and updated version of the browsing environment (e.i. supported versions starting from: Google Chrome ver. 29, Mozila Firefox ver. 28, Safari ver. 9, Opera ver. 30, Microsoft EI/Edge ver. 11)
3. Open provided link to the service.
4. You should see a log-in page.
5. You can log in within the system with provided login and password.

Free testing session #1 survey:

1. Overall summary of the experience (From 0 to 10, from terrible to wonderful)
   1.1. Describe your overall experience
   1.2. Describe your time to learn the program
   1.3. Describe your satisfaction of the usage
   1.4. Describe your speed of navigation in system
2. The interface was intuitive to use (From 0 to 10, from unclear to intuitive)
3. I have seen same interface patterns in other programs and are used to them. (From 0 to 10, from rare to often)
4. I wanted to ask help while using app (From 0 to 10, from rare to often)
5. The terminology in the UI texts was (From 0 to 10, from confusing to clear)
6. Amount of actions, required to do the basic task (From 0 to 10, from a lot to minimal)
7. The speed of the system was responding on my actions (From 0 to 10, from slow to smooth)
8. The most pleasant experience I had so far: (Free text)
9. The worst experience I had so far: (Free text)
10. During testing I encountered a problem (if any): (Free text)

Free testing session #2 survey:

1. Overall summary of the experience (From 0 to 10, from terrible to wonderful)
   1.1. Describe your overall experience
   1.2. Describe your time to learn the program
   1.3. Describe your satisfaction of the usage
   1.4. Describe your speed of navigation in system
2. The transition between working states was (From 0 to 10, from slow to smooth)
3. After all previous testing I was still confused with the parts of the app (From 0 to 10, from often to rare)
4. The terminology in the UI texts was (From 0 to 10, from confusing to clear)
5. Amount of actions, required to do the basic task (From 0 to 10, from a lot to minimal)
6. The color theme of the app is (From 0 to 10, from irritating to pleasant)
7. Amount of things, I need to remember to operate the software (From 0 to 10, from a lot to minimal)
8. The app tolerated my mistakes and allowed to correct them (From 0 to 10, from difficult to easy)
9. The questions and instructions in the survey were (From 0 to 10, from confusing to clear)
10. The part of the testing that gave me the most troubles (if any): (Free text)
11. The most pleasant experience I had so far: (Free text)
12. The worst experience I had so far: (Free text)
13. During testing I encountered a problem (if any): (Free text)
14. I would like developers to improve: (Free text)

Scenario #1 User panel and Admin tools testing
Instructions:

1. Log into the app with your login and password provided.
2. Test the top panel.
   a. In the top of the screen you can see a user panel.
   b. Try to press on the right button and open the dropdown.
3. Log-off and re-login.
   a. Try to log-off and log-in again.
4. Open user profile.
   a. Navigate to the dropdown again and open user profile.
5. Open user settings.
   a. Open user settings either from dropdown or from user profile.
   b. Please, do not change the name or/and password!
6. Open Admin panel.
   a. Open admin panel either from dropdown or from user profile.
7. Create test user.
   a. Open a feature "Add user" from Admin Panel.
   b. Fill in the form.
   c. Do not add admin as a role to the user!
   d. Use some easy-recognizable data for user.
   e. Submit the form.
8. Change user.
   a. Open a feature "Change user" from Admin Panel.
   b. Choose your recently created user from dropdown list.
   c. Change some info in the form and Submit it.
9. Delete user.
   a. Open a feature "Delete user" from Admin Panel.
   b. Choose your recently created user from dropdown list.
   c. Confirm deletion.
   d. Reopen the feature and try to choose user again.
   e. You should not be able to do that.

Survey:

1. How easy it was to perform the task? (From 0 to 10, from difficult to easy)
2. The interactions were intuitive and did not require more detailed instructions (From 0 to 10, from false to true)
3. Describe your overall experience (From 0 to 10, from terrible to wonderful)
4. Do you think this will be faster/easier to do with the app, rather than command-line tool? (Yes/no)
5. During testing I encountered a problem (if any): (Free text)

Scenario #2 Graph handling testing
Instructions:

1. Log into the app with your login and password provided.
2. Choose a graph
   a. In the main working area You will be provided with several big buttons.
   b. Choose one.
3. Explore the graph
   a. The graph should open in the centre of the screen.
   b. Examine it.

4. Basic graph interactions
   a. Try to navigate through graph, move elements, zoom-in and zoom-out.
5. Open graph settings
   a. In the bottom of the graph, there is a settings button.
   b. Click it.
6. Settings tweaking
   a. Try to navigate through settings and change something.
   b. Examine how graph changes.
7. Node selection
   a. Try to click a node and in the left tool menu choose "Show and Edit".
   b. Examine the result.
   c. Try to select another element and perform same action.

Survey:

1. How easy it was to perform the task? (From 0 to 10, from difficult to easy)
2. The interactions were intuitive and did not require more detailed instructions (From 0 to 10, from false to true)
3. Describe your overall experience (From 0 to 10, from terrible to wonderful)
4. During testing I encountered a problem (if any): (Free text)

Scenario #3 Workspace operability testing

Instructions:

1. Log into the app with your login and password provided.
2. Choose a graph
   a. In the main working area You will be provided with several big buttons.
   b. Choose one.
3. Open another graph
   a. In left tools section, choose "open graph" button.
   b. In the opened panel choose one of the lines.
4. Navigate through tabs
   a. You will have multiple graph tabs now.
   b. Open as many additional as you wish and switch between them.
5. Close the tab.
   a. Close one of the tab.
   b. Then choose another to open.
6. Use view states
   a. From above the graph, choose "Text view" button.
7. Switch tabs and view modes.
   a. Switch text tab to another.
   b. Switch view back to the "Graph view".
8. Open help panel
   a. In the top button panel, where you switched modes, navigate right and click "?" button.
   b. How easy was to find it?
9. Navigate through help panel
   a. Just click it through, no need to actually read it.

Survey:

1. How easy it was to perform the task? (From 0 to 10, from difficult to easy)
2. The interactions were intuitive and did not require more detailed instructions (From 0 to 10, from false to true)
3. Describe your overall experience (From 0 to 10, from terrible to wonderful)
4. How easy was to locate the help panel? (From 0 to 10, from difficult to easy)

5. During testing I encountered a problem (if any): (Free text)

Scenario #4 Graph creation testing
Instructions:

1. Log into the app with your login and password provided.
2. Choose a graph
   a. In the main working area You will be provided with several big buttons.
   b. Choose one.
3. Explore the graph
   a. The graph should open in the center of the screen.
   b. Examine it.
4. Open a creation panel
   a. In the left tools panel, press "Create Graph" button.
5. Choose a template
   a. Use the bottom option, choose a template from the dropdown.
6. Fill in the forms
   a. Just try to be creative, it doesn't matter what do you fill in.
7. Close before submitting.
   a. Close the form before submit.
   b. Then try to open it again.
   c. Open Create Graph panel and press continue.
8. Submit the graph
   a. If everything is done correctly, it should open in another tab.

Survey:

1. How easy it was to perform the task? (From 0 to 10, from difficult to easy)
2. The interactions were intuitive and did not require more detailed instructions (From 0 to 10, from false to true)
3. Describe your overall experience (From 0 to 10, from terrible to wonderful)
4. Do you find this approach comfortable for graph creation? (Yes/no)
5. Do you think this process will be faster, than manual configuration files creation? (Yes/no)
6. During testing I encountered a problem (if any): (Free text)