Project Description

Project title	Dashboard for Semantic Data
Term	Winter term 2023/24
Participants	Zhiwei Zhan, Undine Stelter
Supervision	Dr.-Ing. Annett Mitschick, Tom Horak, Sarah Alaghbari
Course	Team Project User Interface Engineering [KPUIE]

Table of Contents

Introduction

Welcome to our project description for the development of a dashboard application designed to visualize semantic data. This project description will lay out our design decisions and implementation plans for the application. Furthermore, it is a living document and will undergo multiple revisions thoughtout the project.

Task description

The primary goal of this project is to develop a proof-of-concept prototype for a web-based dashboard application. The application's focus is to serve as a flexible visualization tool for semantic data. With it, the user will be able to create and customize visualizations and arrange them on a dashboard, both in an effort to facilitate data exploration. Additionally, the prototype is built with extensibility in mind, so new visualization types or customization options can be added in the future.

Use Cases

In this section, we present an example use case for each of the four visualization types.

ID	**[UC0010]**
Visualization	Bar Chart
Data to be visualized	The different skills listed in job postings "Polymechaniker" and how many of those job postings list each skill.
Process	1. Count all Job Postings with "Polymechaniker" in their title 2. List all skills associated with "Polymechaniker" Job Postings (distinct and only German names) 3. Count the number of occurrences for each skill

Example queries	1. ``` SELECT (COUNT(*) AS ?postingCount) WHERE { ?s rdf:type edm:JobPosting. ?s edm:title ?title. FILTER contains(?title, "Polymechaniker"). } ``` 2. ``` SELECT distinct ?skillName WHERE { ?s rdf:type edm:JobPosting. ?s edm:title ?title. FILTER contains(?title, "Polymechaniker"). ?s edm:hasSkill ?skill. ?skill edm:textField ?skillName. FILTER (lang(?skillName) = "de"). } ``` 3. ``` SELECT ?skillName (COUNT(?s) AS ?occurrences) WHERE { ?s rdf:type edm:JobPosting. ?s edm:title ?title. FILTER contains(?title, "Polymechaniker"). ?s edm:hasSkill ?skill. ?skill edm:textField ?skillName. FILTER (lang(?skillName) = "de"). FILTER (?skillName IN ("CNC-Schleifen"@de, "Steuerungskenntnisse"@de, "Selbstständigkeit"@de, "Qualitätsbewusstsein"@de, "Problemlösungsfähigkeit"@de, ⋯)) } GROUP BY ?skillName ```
Notes	Since the total number of skills in the data set is quite large, the user should be prompted to select a specific job title to do this particular analysis for.

ID	**[UC0020]**
Visualization	Pie Chart
Data to be visualized	The distribution of Job Postings listed as fulltime over the entire dataset.
Process	1. Count all Job Postings 2. Count all Job Postings that are fulltime 3. Count all Job Postings that are part time

Example queries	1. `SELECT (count(?s) AS ?fulltimeCount )` `WHERE {` `?s rdf:type edm:JobPosting.` `?s edm:title ?title.` `}` 2. `SELECT (count(?s) AS ?fulltimeCount )` `WHERE {` `?s rdf:type edm:JobPosting.` `?s edm:title ?title.` `?s mp:isFulltimeJob "true"^^xsd:boolean.` `}` 3. `SELECT (count(?s) AS ?fulltimeCount )` `WHERE {` `?s rdf:type edm:JobPosting.` `?s edm:title ?title.` `?s mp:isFulltimeJob "false"^^xsd:boolean.` `}`
Notes	Since the isFulltimejob field can be missing, we shouldn't calculate the false count from the total and true counts.

ID	**[UC0030]**
Visualization	Line Chart
Data to be visualized	The number of Job Postings created between November 6 2023 and November 10 2023.
Process	Count the job postings for each day.
Example queries	`SELECT *` `WHERE {` `?s rdf:type edm:JobPosting.` `?s edm:title ?title.` `?s edm:dateCreated ?created.` `FILTER (xsd:dateTime(?created) = xsd:dateTime("2023-11-06T00:00:00Z")).` `}` ... `SELECT *` `WHERE {` `?s rdf:type edm:JobPosting.` `?s edm:title ?title.` `?s edm:dateCreated ?created.` `FILTER (xsd:dateTime(?created) = xsd:dateTime("2023-11-10T00:00:00Z")).` `}`
Notes	We are still working on querying all counts in one query, similar to the skill names in [UC0010].

ID	**[UC0040]**
Visualization	Star Plot

Data to be visualized	For multiple job titles, compare how many of their Job Postings have a user-made selection of "soft skills" listed.
Process	For each of the job titles 1. Count the total number of Job Postings 2. Get the skill counts for the set of "soft skills" 3. Divide each of the skill counts with the total number of Job Postings to "normalize" it to a number between 0 and 1
Example queries	1. `SELECT (COUNT(*) AS ?postingCount)` `WHERE {` `?s rdf:type edm:JobPosting.` `?s edm:title ?title.` `FILTER contains(?title, "Polymechaniker").` `}` ... `SELECT *` `WHERE {` `?s rdf:type edm:JobPosting.` `?s edm:title ?title.` `?s edm:dateCreated ?created.` `FILTER (xsd:dateTime(?created) = xsd:dateTime("2023-11-10T00:00:00Z")).` `}` 2. `SELECT ?skillName (COUNT(?s) AS ?occurrences)` `WHERE {` `?s rdf:type edm:JobPosting.` `?s edm:title ?title.` `FILTER contains(?title, "Polymechaniker").` `?s edm:hasSkill ?skill.` `?skill edm:textField ?skillName.` `FILTER (lang(?skillName) = "de").` `FILTER (?skillName IN ("Teamfähigkeit"@de, "Kommunikation"@de, "Selbstständigkeit"@de, "Leitungsbereitschaft"@de, "Motivation"@de))` `}`
Notes	At this point in time, it is still unclear whether the star plots are on the same tile or each gets their own tile.

Requirement Specification

Functional Requirements

Required Functional Requirements

ID	Name	Description
[FR0010]	Generate visualizations	The user is able to choose out of four visualization types for each visualization the application generates. The visualization types are picked from the categories Evolution, Ranking and Part of a whole.

ID	Name	Description
[FR0011]	Data selection	The user is able to select which instance data is used to generate visualizations ([FR0010]).
[FR0012]	Domain limiting	The user is able to filter the instance data which is used to to generate visualizations ([FR0010]) by limiting the domain, e.g. by specifying a time frame for the data.
[FR0013]	Mapping customization (color)	The user is able to customize the color scheme mapped onto the data when generating visualizations ([FR0010]).
[FR0014]	Data filtering	The user is able to filter the instance data used to generate visualizations ([FR0010]) by selecting attributes, values or relationships from a list generated by the application.
[FR0015]	Visibility toggle	The user is able to toggle the visibility of different values, e.g. displaying only certain skills out of a list of skills.
[FR0020]	Dashboard	The app contains a dashboard, with each user-generated visualization being represented by a tile. The application supports up to 20 tiles at once without crashes. The number of tiles the user can generate is not limited.
[FR0021]	Arrange visualizations	The user is able to arrange the tiles representing visualizations ([FR0020]) freely on the dashboard using a drag-and-drop system.
[FR0022]	Resizeable tiles	The dashboard tiles can be resized freely by the user, in unit steps. The minimum size is 1x1 unit.
[FR0023]	Labeled visualizations	Each visualizations on the dashboard incorporates labels appropriate for the visualization type, e.g. axis labels and color keys.
[FR0024]	Delete visualizations	The user is able to delete visualizations by deleting the tile containing them.
[FR0030]	Database querying	The application communicates with the SPARQL endpoint of an existing GraphDB database via premade SPARQL queries that are then configured by the user.
[FR0040]	Mouse support	The application supports user interaction via both mouse and keyboard inputs, enabling users to navigate and select options using the mouse, while allowing text and numerical data entry through keyboard input in designated fields.
[FR0041]	Display Language	The application's display language is English.

Optional Functional Requirements

ID	Name	Description
[FO0010]	Mapping customization (totals)	The user is able to add totals and additive values when generating visualizations ([FR0014]).

ID	Name	Description
[FO0011]	Mapping customization (grouping)	The user is able to define groups of values when generating visualizations ([FR0014]).
[FO0012]	Custom titles	The user is able to give custon titles to generated visualizations.
[FO0013]	Stashing visualizations	The user is able to "stash" visualizations that are not needed at the moment, without having to delete them.
[FO0014]	Download visualizations	The user is able to download visualizations or the means to re-generate them as a file. Such files can be uploaded into the program to re-generate the visualizations. Visualizations can be shared between users by sharing these files (e.g. via Email).
[FO0015]	Interactive Visualizations	The user is able to interact with visualizations, e.g. adjust the portrayed time frame. These adjustments may alter the SPARQL query underlying the visualization. This does not include interaction between visualizations, e.g. brushing and linking techniques.
[FO0016]	Aspect Ratio Locking	The user is able to lock the aspect ratio of a particular tile to prevent accidental resizing.
[FO0017]	User guidance (Filtering)	The application offers the user guidance for selecting the data to be visualized ([FR0012], [FR0014]), e.g. by displaying lists of available attributes for the user to select some or by displaying counts of instances affected by a filter.
[FO0020]	Touchscreen support	The application's support for mouse and keyboard ([FR0040]) is extended by touchscreen support.
[FO0021]	German Language Support	The application additionally offers German as a display language, with a switch to toggle between languages.

Non-functional Requirements

ID	Name	Description
[NF0010]	Interchangable dataset	The application is able to generate visualizations for any semantic data set retrievable from a SPARQL Endpoint.
[NF0020]	Usability	The application can be used without SPARQL knowledge and without extensive training.
[NF0021]	Reactiveness	The application strikes a balance between minimizing user interface response times and maximizing the number of data points displayed at once.
[NF0030]	Extensibility	The application is able to be extended easily, e.g. by adding more visualization types or by adding further customization options to existing visualization types.

ID	Name	Description
[NF0031]	Readability	The code should be readable and well documented to facilitate extensibility ([NF0030]).

Our focus lies on the interchangability of the dataset, the application's extensibility and the responsiveness of the user interface.

System Components

This section will detail the different components of the applications, our design considerations for each component and a our implementation goals, ranging from the minimum to the ideal version.

Visualization Generator/"Wizard"

General Rule: Immutable aspects of the visualization are picked in the Wizard, the rest is done in the Editor.

- a page with multiple sections that build onto another
- user clicks on "Generate new Visualization" to navigate there
- its purpose is to guide the user through generating a visualization and help them explore the data
- first step: choosing the visualization type
- the further steps and their layout depend on the visualization type, to not overwhealm the user with potentially irrelevant options
- the user should be able to go back to change the visualization type without leaving the Wizard page
- being able to go back has implications for the layout of further steps, so this has to be explored thoroughly

Minimum	• the same data selection process for all visualization types • text and number fields for data pre-selection, without feedback for the user as to how those filters impact the data set • user can choose which property to map to the axes ([FR0011])
Implementation Goal	• icons for the visualization types • data pre-selection steps depend on the visualization type • data preview in a table when the user puts in a filter
Ideal	• presets for the user to choose from and adjust • user guidance for selecting the right filters, e.g. through recommendations ([FO0017])

Visualization Editor

- user gets to the Editor by completing the Wizard
- when clicking the "Edit" button in a tile, they also get taken there
- user can choose the attributes to visualize and how they are mapped
- user can also choose the time frame to visualize, if applicable
- this is a lot of settings, so related settings should be grouped together
- the settings made here are saved in a config file (JSON file, human-readable)
- there is an "Apply changes" button and once that is pressed, a SPARQL query is generated, sent to the database and the visualization is generated (and visible on the side)

Minimum	• user assembles a SPARQL query with assistance (e.g. using dropdown menus) • settings and query are saved in a config file • when pressing "Apply changes", the editor is closed and a tile containing the visualization is generated ([FR0010]) • user can apply/change filters to define the range of the axes ([FR0012]) • user can choose which attributes to display as data points ([FR0011])
Implementation Goal	• the user only chooses options, no directly visible SPARQL query • user can choose from a set of predefined color schemes ([FR0013]) • visualization to be generated is visible on one half of the editor page and pressing "Apply changes" re-loads the image • related settings are grouped together • user can give custom titles to visualizations ([FO0012]) • include a "Delete" button here, in addition to the "Delete" button on the dashboard tile
Ideal	• user can do certain adjustments directly on the visualization (e.g. axis label is a dropdown menu with a selection of attributes) ([FO0015]) • more mapping options ([FO0010], [FO0011]) • user can define a custom color scheme ([FR0013]) • user can save color schemes and has them available for other visualizations

Dashboard

- anticipated challenge: reload the image in the tile once the user has changed the size, look out that it doesn't get too slow
- performance overall will likely be an issue here
- use inbuilt gridstack.js functionality
- the page can expand downwards as needed, no limited grid size

Minimum	• a dashboard with tiles ([FR0020]) • tiles are resizeable and draggable ([FR0021], [FR0022]) • the user can delete tiles ([FR0024]) • Application asks user, if they really want to delete the tile, to avoid accidental deletion
Implementation Goal	• "inventory" for favorited tiles, so the user can stash tiles away ([FO0013]) • user can mark tiles as favorites by clicking a "Favorite" button • favorited tiles are marked • long tile titles are handled appropriately, e.g. by scaling down the font or shortening them
Ideal	• a auto-align or auto-format button (e.g. instantly making a tile take up half the screen) • the user can directly interact with visualizations, e.g. adjusting the portrayed time frame, without going to the editor page ([FO0015])

Visualizations

Line Graph

- we want to include a line graph, because it is incredibly versatile
- good for visualizing a value over time

Implementation Goal	• a graph gets generated • the axes are labeled ([FR0023])
Possible improvements	• threshold colorization • can be extended into multiple lines or stacked line chart • user can hover over a data point to get an exact value

Bar chart

- good for nominal data
- easy to understand
- the number of bars that can comfortably be understood by the user is limited, so if there are too many values for the x-axis, the user should be prompted to filter more (most likely in the Wizard, since this counts towards data pre-selection)
- our skill count example already implements this limitation by having the user input a job title (with a required text field)

Implementation Goal	• a bar chart gets generated • the axes are labeled ([FR0023])

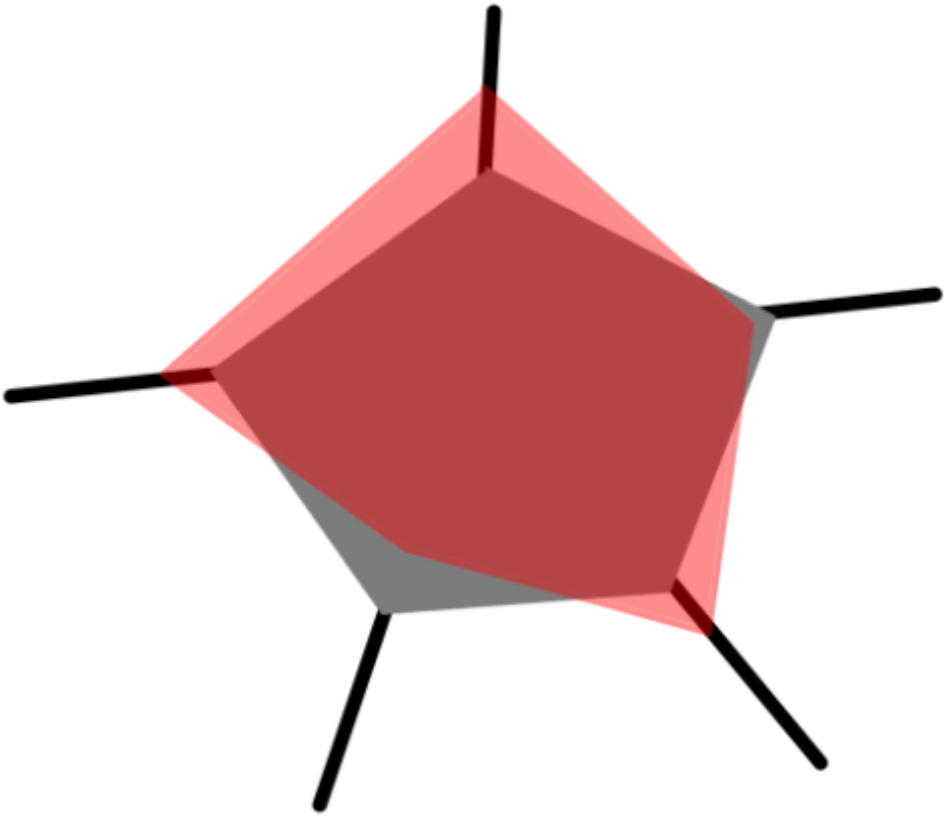
Possible improvements	• guide lines parallel to the x-axis to improve readability • can be extended into stacked bar chart (with visibility toggle for the different stack colors) • overlay 2 bars with transparency (variant of the stacked barchart) • have multiple bars right next to each other for each value on the x-axis • user can hover over a bar to get more information • combine multiple (compatible) bar charts into one stacked bar chart (either replacing the two source tiles or generating a third tile)

Star plot

- good for comparing multiple similar objects
- has a rather square aspect ratio, so it works well with a tile system

Implementation Goal	• a star plot with a variable number of axes gets generated • the axes are labeled ([FR0023])
Possible improvements	• allow the user to define "templates" for a set of properties to map onto the axes to ease the process of creating multiple small visualizations • alternatively, give the user the option to generate a batch of small multiples on the same tile • overlay multiple star plots (with transparency to help with occlusion issues) • display an overlay for an ideal starplot shape or the medium of all starplot shapes

"ideal shape" for a starplot in grey:



Pie Chart

- good for getting a general sense of how the data is composed and exact amounts are less important
- as with the bar chart, the number of different values that can be visualized is limited
- pie charts work great for booleanFields, but fields with more different values should be filtered more

Implementation Goal	• a pie chart gets generated • the segments are labeled appropriately (e.g. inside our out, depending on the segment size) ([FR0023])
Possible improvements	• generate a donut chart • user can hover over a segment to get more information

SPARQL Queries

- start out with simple query builder
- detect available attributes from database
- potentially blacklist attributes not suited for visualization

Implementation Goal	• the query gets generated from settings the user made in the Visualization Editor ([FR0030]) • the user gets shown all available attributes • the query gets saves inside the config file
Possible improvements	• available attributes are filtered, depending on whether they can be visualized • available attributes are filtered, based on options the user chose previously • the application makes recommendations based on the visualization type

Configuration File System

- JSON file, human-readable
- contains the SPARQL query and all settings concerning the visualization
- using only the config file, you can generate the exact same visualization
- there is a list of "favorite" visualizations that are save from deletion and that the user can drag onto the dashboard
- user can also download any visualizations by clicking a download button

Minimum	• settings and SPARQL query get saved in config file • fixed file path for the config file directory • import config files by copying them into the config file directory ([FO0014])
Implementation Goal	• imported visualizations can be selected from a list and dragged onto the dashboard • the user can specify the config file name
Ideal	• config files can be uploaded into the applocation to re-generate visualizations ([FO0014]) • favorite tiles section contains tiles with preview images

User Interface

- since the focus lies on the visualizations, the majority of the screen space should be dedicated to them
- the Visualization Editor will get a separate page to not take up too much screen space
- this will also allow us to have a preview image of the to be edited visualization

Implementation Goal	• user interface with a dashboard and editor page • user interface is operated by mouse ([FR0040]) • the interface does not crash during use

Ideal	• mouse and touchscreen support ([FO0020]) • the interface is reactive ([NF0021]) and provides feedback to the user

Implementation

This section will be expanded upon in V2.

Libraries and Frameworks

- Angular for the app, since elevait is already using Angular
- gridstack.js for the dashboard tile logic
- Bootstrap for the styling
- D3 for the visualizations
- GraphDB for the SPARQL queries

Why gridstack.js?

When deciding how to build the tiles for our dashboard, we quickly came to the conclusion that we preferred learning to use a framework over building the tile system ourselves, especially because we want resizeable tiles and to not burden the user with reorganizing their tiles manually whenever they move a tile.

We found three different frameworks to choose from:

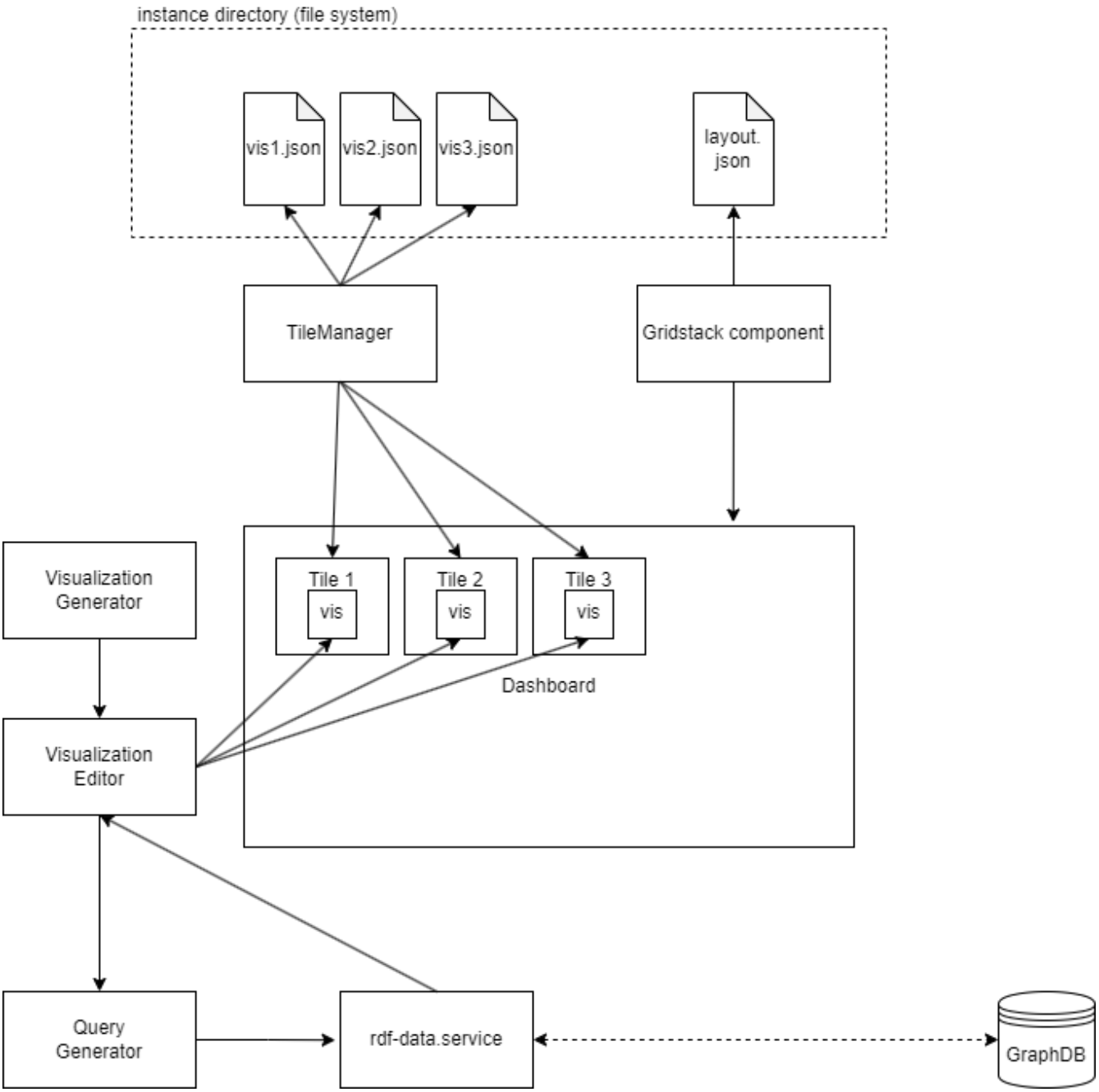
- Angular Dashboard Layout by Syncfusion
- ArchitectUI
- gridstack.js

The Angular Dashboard Layout comes with a lot of premade components and styling, though we would probably be confined to the free trial version. We really liked their tile dragging system.

ArchitectUI offers grreat responsiveness and layout options, along with a Slack channel for questions. It seems to focus on static dashboard applications though, so we had concerns about getting the tiles to be flexible enough.

We ultimately landed on gridstack.js, since it provides the same tile functionality, but is much more light-weight than the Angular Dashboard Layout. On the downside, it provides *only* the tile system, so we included Bootstrap in our project as well to helo with styling the components and tiles.

Top Level Architecture



GUI Prototype

For the GUI prototype images, please look into the doc/images folder in the GitLab repository.

Implementation Plan

- build a running Hello World application
- build a simple dashboard
- generate a visualization from dummy data
- display visualization inside tiles
- brainstorm GUI ideas

--- hand in project description V1 ---

- establish communication to elevait DB, run some example queries (once we have access to the data)
- draw GUI prototypes
- make a very simple query builder

--- presentation on the 6th of November ---

- start on config file system, save query to config file
- generate visualization from config file (with sample attributes)
- plan out Wizard and Editor for each visualization type
- start implementing Wizards and Editors
- write configurable SPARQL queries for each visualization type

--- hand in project description V2 ---