Project Description

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

This section will be expanded upon in V2 of the project description.

Requirement Specification

Functional Requirements

Required Functional Requirements

ID	Name	Description
[FR0010]	Generate visualizations	The user is able to generate visualizations of four different types.
[FR0011]	Data selection	The user is able to select which data is used to generate visualizations ([FR0010]).
[FR0012]	Data filtering	The user is able to filter the data which is used to to generate visualizations ([FR0010]).
[FR0013]	Mapping customization (color)	The user is able to customize the color scheme mapped onto the data when generating visualizations ([FR0010]).
[FR0020]	Dashboard	The app contains a dashboard, with each user-generated visualization being represented by a tile.
[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.
[FR0023]	Labeled visualizations	The visualizations on the dashboard tiles include labels appropriate for the visualization type.
[FR0024]	Delete visualizations	The user is able to delete visualizations by deleting the tile containing them.
[FR0030]	Database querying	The application communicates with an existing GraphDB database by constructing SPARQL queries.
[FR0040]	Mouse support	The application is operated using mouse and keyboard.

Optional Functional Requirements

ID	Name	Description
[FO0010]	Mapping customization (totals)	The user is able to add totals and additive values when generating visualizations ([FR0010]).
[FO0011]	Mapping customization (grouping)	The user is able to define groups of values when generating visualizations ([FR0010]).
[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]	Share visualizations	The user is able to share visualizations they generated or the means to re-generate the visualization with other users.
[FO0015]	Interactive Visualizations	The user is able to interact with visualizations, e.g. adjust the portrayed time frame. This does not include interaction between visualizations, e.g. brushing and linking techniques.
[FO0020]	Touchscreen support	The application's support for mouse and keyboard ([FR0040]) is extended by touchscreen support.

Non-functional Requirements

ID	Name	Description
[NF0010]	Interchangable dataset	The application is able to generate visualizations for any semantic dataset.
[NF0011]	Versatile visualization types	The visualization types provided by the application can be used to visualize a variety of data types.
[NF0020]	Usability	The application can be used without SPARQL knowledge and without extensive training.
[NF0021]	Responsiveness	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.
[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 Editor

- when creating a new visualization, user chooses data set and visualization type first, those are immutable
- then they get taken to the editor page
- 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 choose which property to map to the axes ([FR0011]), as well as applying filters to define the range of the axes ([FR0012]) • user can choose which attributes to display as data points ([FR0011]) • user can choose from a set of predefined color schemes ([FR0013])
Implementation Goal	• the user only chooses options, no directly visible SPARQL query • icons for the visualization types • 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 define a custom color scheme ([FR0013]) • user can give custom titles to visualizations ([FO0012])
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 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

Minimum	• a dashboard with tiles ([FR0020]) • tiles are resizeable and draggable ([FR0021], [FR0022]) • the user can delete tiles ([FR0024])
Implementation Goal	• "inventory" for tiles, so the user can stash unused tiles away ([FO0013])
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

Implementation Goal	• a graph gets generated • the axes are labeled ([FR0023])
Possible improvements	• threshold colorization • can be extended into stacked area chart

Bar chart

- good for nominal data
- easy to understand

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

Boxplot

- useful for analyzing the distribution of an attribute over the entire dataset

Implementation Goal	• a single boxplot gets generated • the axes are labeled ([FR0023])
Possible improvements	• allow multiple boxplots in the same graph • allow for custom thresholds

Star plot

- good for comparing multiple similar objects

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

Backup: Pie Chart

- backup option in case one of the other visualization types cannot be realized
- good for getting a general sense of how the data is composed and exact amounts are less important

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

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 unused visualizations ("inventory") that the user can drag onto the dashboard

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 added to the "inventory" via file browser ([FO0014]) • inventory 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 responsive ([NF0021]) and provides feedback to the user

Implementation

This section will be expanded upon in V2.

Libraries and Frameworks

- Angular for the app
- gridstack for the dashboard tile logic
- Bootstrap for the styling
- D3 for the visualizations
- GraphDB for the SPARQL queries

Top Level Architecture

GUI Prototype

The final version of the GUI prototype will go here.

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
- start on config file system, save query to config file
- generate visualization from config file (with sample attributes)

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