Graphalyzer

May1618

Table of Contents

Team Roles…………………………………………………………………………..2
Problem Statement…………………………………………………………………2
Proposed Design……………………………………………………………………4
Requirements………………………………………………………………………..5
Backend Functional Requirements……………………………………………5
Frontend Functional Requirements…………………………………………...5
Frontend Nonfunctional Requirements………………………………………6
Front End Specifications…………………………………………………………4
Back End API…………………………………………………………………………5
Concept Sketch………………………………………………………………………7
Interface Description…………………………………………………………....8
Testing Specifications……………………………………………………………8
Test Plan……………………………………………………………………………...9

Team Roles

- Andrew Bowler - Webmaster
- Alberto Gomez-Estrada - Communications
- Michael Sgroi - Key Concept Handler
- Richard White - Key Concept Handler
- Taylor Welter - Team Lead

Problem Statement

Graphs are an elegant way to illustrate the extensive web of connections between many points of data. As the data set becomes larger and more complex, it becomes more important to be able to visualize specific instances of various relationships. The goal of this project is to build a generic graph visualization, analysis, and search web application.

Proposed Design

The project shall be set up as a client-server model. The client (aka frontend) shall be a web application written in HTML and JavaScript with various libraries, with Angular as the application's front-end framework. It shall be responsible for displaying the graphs and receiving user input. The server (aka the backend) shall be written in D. It shall first be responsible for accepting newly generated graphs from various services via rest. It shall also be responsible for serving up the graph to the frontend. It shall additionally be responsible for performing all the graph calculations and sending the results to the client. Communication between the server and client shall use websockets sending json messages.

Requirements

Backend Functional Requirements

- The product shall allow for the upload of arbitrary graph files for analysis via REST
- The product shall index incoming information associated with nodes and edges

- The product shall allow for the searching of nodes by name
- The product shall process the graph to determine:
  - Depth
  - Breadth
  - Interconnectedness
  - Impact

Frontend Functional Requirements

- The product shall be able to visualize the graph
  - The graph's visualization shall clearly distinguish different nodes and their connections
  - The visualization shall clearly distinguish properties of the graph through color coding, such as impact and interconnectedness
- The product shall allow for the exploration of the graph via click or touch
- The product shall display the ability to search the graph in the user interface
- The product shall maintain an immutable search history easily accessible to the user for reuse of search terms
- The product shall display to the user when it is processing a user's search query

Nonfunctional Requirements

- The product shall be able to process, visualize parts of, and analyze graph files with sizes up to 32GB
- The product shall not crash or lose connection to the REST service while in use
- The product shall not have any memory leaks or any loss of graph data.
- The product shall not voluntarily or involuntarily modify the graph data.

Front End Specifications

Important Objects:

- Graph
    - An object holding a set of Nodes and Edges, and a **graph name** property
- Node
    - An object that holds a list of properties describing itself
    - Has a set of Edges
- Edge
    - An object that describes two Nodes it connects
- Selected Node
    - The Node in the Graph that has been selected from the visualization via click
- Search Query
    - Contains a **graph name**, and two optional properties **Node name** and **n-degrees connections**

Modules:

- Graph Panel
    - Takes up most of the UI display, placed to the left of the Search Utility and Node Properties panels. Contains a Graph object.
    - Vis.js renders the graph model from the Graph and properties received from the Handler to the screen
    - Vis.js allows graph nodes to be clickable (setting the Selected Node), and information on the Selected Node is displayed in the Node Properties Panel
- Search Utility Panel
    - Placed at the top right of the UI display, above the Node Properties Panel
    - Search for a graph by **graph name**.
        - A new Search Query object is created and passed to the Graph Search Handler
        - Optional **Node name** and **n-degrees connections** parameters can be provided
        - If **graph name** is blank and there is already a graph being displayed, the currently displayed graph's **graph name** will be provided as the parameter. If a graph is not being displayed, the UI will deny the search query and request a **graph name**.

- Node Properties Panel
    - Placed on the right of the UI display, below the Search Utility Panel.
    - Displays all the information of a Selected Node from the Graph Panel (selection by clicking), otherwise is empty.
- Graph Data Handler
    - Receives graph data from the server via a WebSocket connection, and transfers the data to a Graph object that is visualized through the Graph Panel
- Graph Search Handler
    - Receives a Search Query from the Search Utility
    - Sends the query out to the server
        - A new Graph will be returned to the Graph Data Handler based on the parameters provided.

Back End API

Class Listing:

Websocket Server
Handler (and subclasses)
REST Server

Class Details:

Websocket Server:

Contains two parts. One part will handle the connection [e.x., handleConn(scope WebSocket sock)]. The other part will start the server [e.x., startServer()].

Handler:

Is a base abstract class that handler subclasses will extend with the types of requests, [i.e., assigning a session ID].
Examples of base classes that will extend handler include:

SessionIDHandler - Assigns client unique session ID
ListGraphHandler - Returns list of all graphs in db
GetGraphHandler - Return graph by name

Accepts graph name string

GetGraphFromNodeHandler - Return the graph around the node
(within reason)
Accepts a node
GetGraphFromNodeNDegreeHandler - Return a graph comprised
of the nodes of degree N around the specified node.
Accepts a node, a number representing the degree

REST Server:

Contains logic to scan a folder for new files and if a new item is found, calls
Neo4J with the file location for input in order to store the file contents in the database.

Concept Sketch

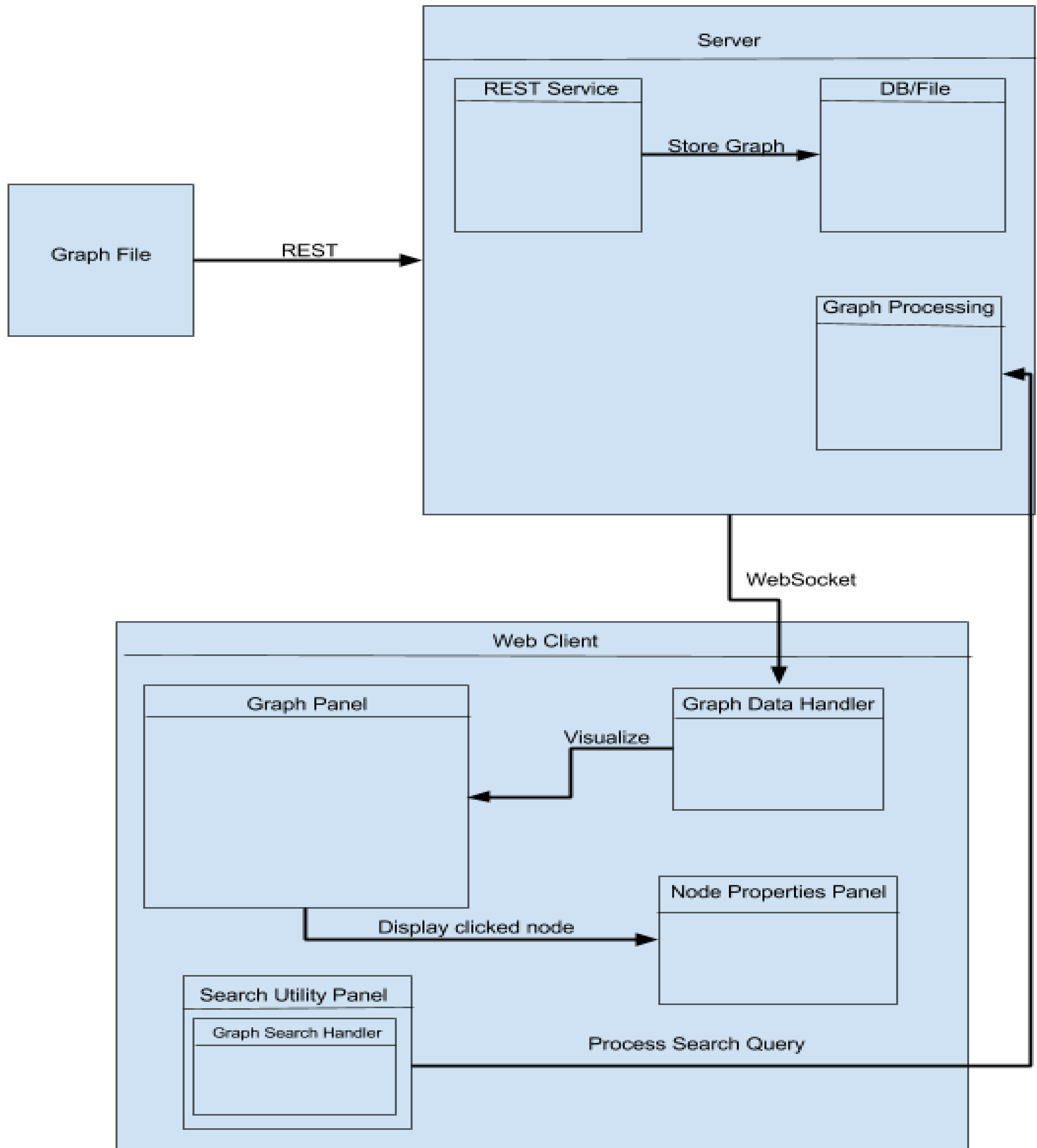

Figure A. Concept Sketch

Interface Description

Further negotiation with Workiva is required for what they expect for a user interface, but for now, we have come up with our own user interface.

The UI displays the graph panel on the left side of the screen, taking up the majority of the display. The graph panel displays the visualization of the graph, with each Node being clickable by the user. The right side of the screen has a Search Utility panel on the top right, and below it, the Node Properties panel. When a user clicks on a Node, its properties are displayed in the Node Properties panel. When the user types in a search query in the Search Utility panel, a new graph is displayed reflecting the user's input.

Testing Specifications

Backend Testing Specifications:

1. All packages, classes, and functions will be unit tested.
2. Web Socket connections will be tested via a dummy client.
3. Class mocking will be performed for easier testing using a more elaborate testing library located here: https://github.com/nomad-software/dunit

Frontend Testing Specifications:

1. All objects and modules will be unit tested with the built-in AngularJS testing framework.
2. Integration testing with the server will be done with use case testing, such as search querying and graph displaying.

Test Plan

<table>
<tr><th>Requirement</th><th>Validation Test</th></tr>
<tr><td>The product shall allow for the upload of arbitrary graph files for analysis via REST</td><td>Upload sample graph files</td></tr>
<tr><td>The product shall index incoming information associated with nodes and edges</td><td>Store the information in the database, assert it against the expected outcome</td></tr>
<tr><td>The product shall allow for the searching of nodes by name</td><td>Query the name row in the database and confirm it against the search results</td></tr>
<tr><td>The product shall process the graph to determine:
Depth
Breadth
Interconnectedness
Impact</td><td>Unit tests based on these various qualities of graphs</td></tr>
<tr><td>The product shall be able to visualize the graph
The graph's visualization shall clearly distinguish different nodes and their connections
The visualization shall clearly distinguish properties of the graph through color coding, such as impact and interconnectedness</td><td>This will all be validated with visual confirmation tests</td></tr>
<tr><td>The product shall allow for the exploration of the graph</td><td>Interactive testing by the way of attempting to explore the graph</td></tr>
<tr><td>The product shall display the ability to search the graph in the user interface</td><td>Visual confirmation of the search dialog</td></tr>
<tr><td>The product shall allow for exploration of the graph via click or touch interface</td><td>Click/touch the visualization in various states</td></tr>
<tr><td>The product shall maintain an immutable search history easily accessible to the user for reuse of search terms</td><td>Attempt to change the search history, shouldn't be able to</td></tr>
<tr><td>The product shall display to the user when it is processing a user's search query</td><td>Visual confirmation test</td></tr>
</table>

Table 2: Test Plan

Appendix

Table 1: Risks..3
Figure A. Concept Sketch...7
Table 2: Test Plan...9