May1618

Advisor: Dr. Mitra

# Graphalyzer

Client: Workiva

Design Document Final Version

Andrew Bowler, Alberto Gomez-Estrada, Michael Sgroi, Richard White, Taylor Welter

October 19, 2015

# Table of Contents

1. Introduction....................2
    1.1 Team Roles....................2
    1.2 Problem Statement....................2
2. System Level Design....................2
    2.1 Requirements....................2
    2.2 Concept Sketch....................4
    2.3 Front End Design Specification....................5
    2.4 Back End API....................6
3. Interface and Testing....................7
    3.1 Interface Description....................7
    3.2 Testing Specifications....................7
    3.3 Test Plan....................8
4. Implementation Issues and Challenges....................9

# 1. Introduction

## 1.1. Team Roles

1.1.1. Andrew Bowler - Webmaster
1.1.2. Alberto Gomez-Estrada - Communications
1.1.3. Michael Sgroi - Key Concept Holder
1.1.4. Taylor Welter - Team Leader
1.1.5. Richard White - Key Concept Holder

## 1.2. 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.

# 2. System Level 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 React 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 Python. 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.

## 2.1. Requirements

### 2.1.1. 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

### 2.1.2. 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

### 2.1.3. 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.

## 2.2. Concept Sketch

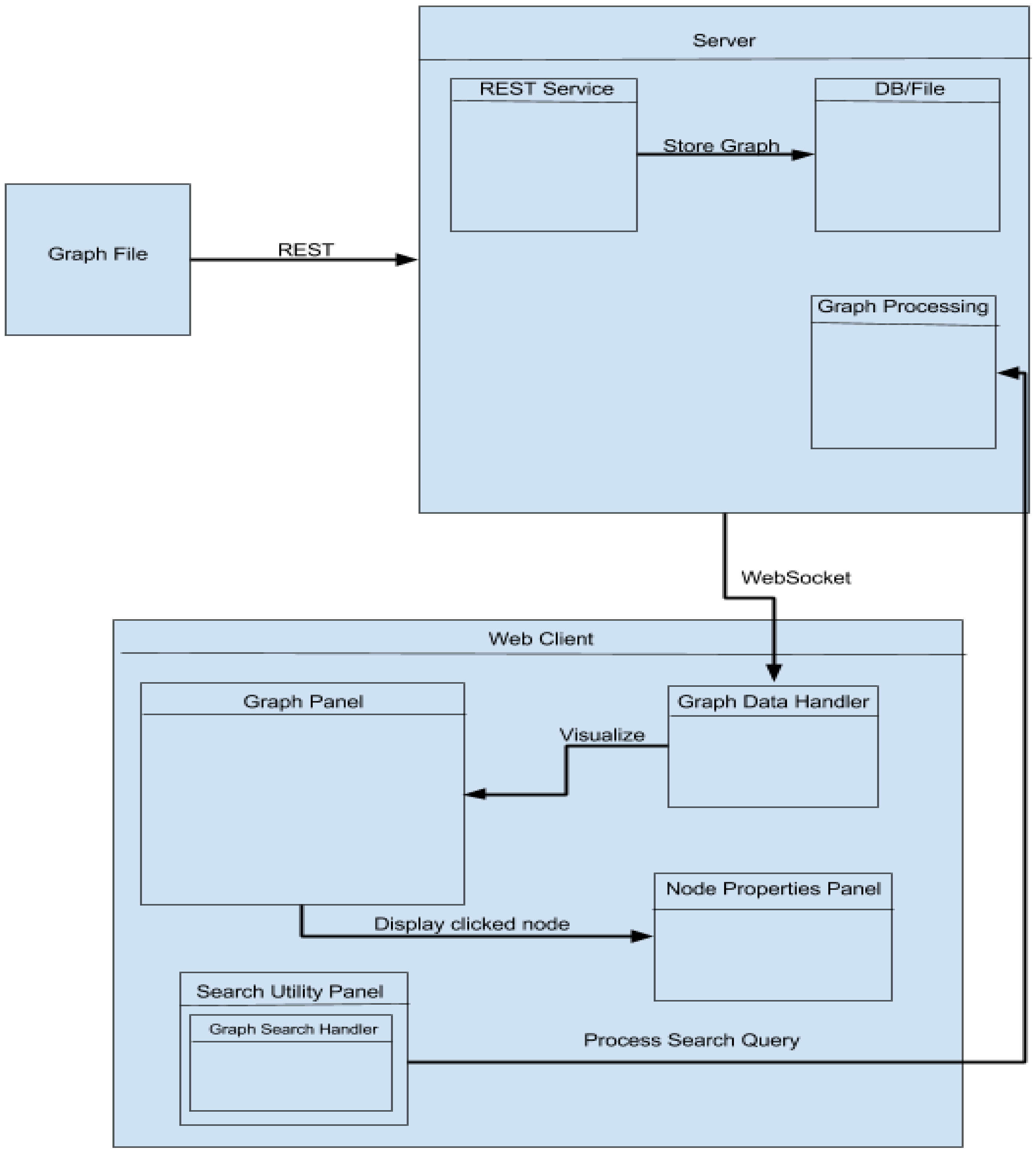

Figure A. Concept Sketch

## 2.3. Front End Design Specification

### 2.3.1. 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**

### 2.3.2. 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.

## 2.4. 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. The other part will start the server.

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.

## 3. Interface and Testing

### 3.1. 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.

### 3.2. 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. We will be using Jest for React unit testing.
2. Integration testing with the server will be done with use case testing, such as search querying and graph displaying.

## 3.3. Test Plan

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

Table 1: Test Plan

## 4. Implementation Issues and Challenges

### 4.1. Scalability

With a maximum input file size of 32GB, the scale of the operations the system has to perform is a major concern. If proper procedures are not followed, there is always the chance that our system will not be able to handle larger file sizes and more complex computations. This would impair the proper functioning of the system and would limit its usability and practicality. To attempt to mitigate this risk, we will design the project to only load relevant sections of a graph at a time. In addition, the frontend will only display a certain amount of nodes/edges.

### 4.2. Technology Support

Many of the technologies used in this project are fairly new, and thus could be poorly documented or lead to *uncharted waters* situations where a solution may not very applicable to an existing one in the language in question. To mitigate this risk, we have decided to extensively study up on the languages in each of our respective domains.

### 4.3. Backend Language

We had originally decided to use D programming language to program the backend. This, however, proved troublesome due to REST communication troubleshooting problems and general lack of support. Debugging the REST communication in D was problematic since there was little documentation or examples to base our REST communications off of. We resolved this issue by switching to Python. This change wasn't a huge time investment since we did it earlier in the project. It proved to be an incredibly beneficial change since Python has more documentation and active library support for Neo4J.

### 4.4. Frontend Framework

Originally, we were using Angular.js as the framework for the frontend, but not enough of us were familiar enough with it for it to be any more than a hinderance due to its strict nature. We decided we needed something that would allow us to more freely use traditional HTML elements alongside the features that a JavaScript framework provides. We ultimately decided on React.js. The change to it was pretty smooth and we were back to where we were prior to the change within a week or so.

## Appendix

Figure A. Concept Sketch........................................................................................4
Table 1: Test Plan...................................................................................................8