Computer Science
COC251
B927949

Creating a knowledge base for The Binding of Isaac

by

Tyler J. Bowcock

Supervisor: Dr. D D Freydenberger

Department of Computer Science
Loughborough University

May 2023

Abstract

The Binding of Isaac with a large number of items and game mechanics, which makes it difficult to find items that leverage those mechanics to the player's advantage. This project aims to ease this issue through the developemt of a web application that represents these interactions in an easy to view manner and allows them to search for interactions. The web application will make use of a graph database to increase the speed and flexibility of database queries.

Contents

List of Figures

List of Acronyms

ACID Atomic, Consistent, Isolated, Durable

AWS Amazon Web Services

CDK Component Dev Kit

CORS Cross-Origin Resource Sharing

HTML Hyper-Text Markup Language

HTTP Hyper-Text Transfer Protocol

JSON JavaScript Object Notation

JSX JavaScript XML

OGM Object Graph Mapper

SQL Structured Query Language

WSGI Web Server Gateway Interface

XML Extensible Markup Language

Chapter 1

Introduction

1.1 Problem Definition

Item interactions are an important mechanic of most modern roguelike/roguelite games, including The Binding of Isaac. However, with hundreds of items, each with a handful of good or bad interactions, it is nearly impossible to effectively remember them all. Graph databases are purpose-built to store and navigate relationships.[1] The output of this project will be a web application that leverages this feature of graph databases to allow users to query item interactions in The Binding of Isaac.

1.2 Aims and Objectives

The goal of this project is to make querying item interactions in The Binding of Isaac quicker and easier by using graph databases. Users will also be able to update the data in the database to ensure it matches any changes in the game.

The aims of the project are to:

1. Create a graph database containing relevant data about The Binding of Isaac.
2. Develop a web application that utilises a graph database to helps users to find item interactions in the game.
3. Explore testing methodologies to aid in producing a stable application with high quality code.
4. Search for possible ways to extend the project with future updates.

1.3 Risks and Constraints

Cost

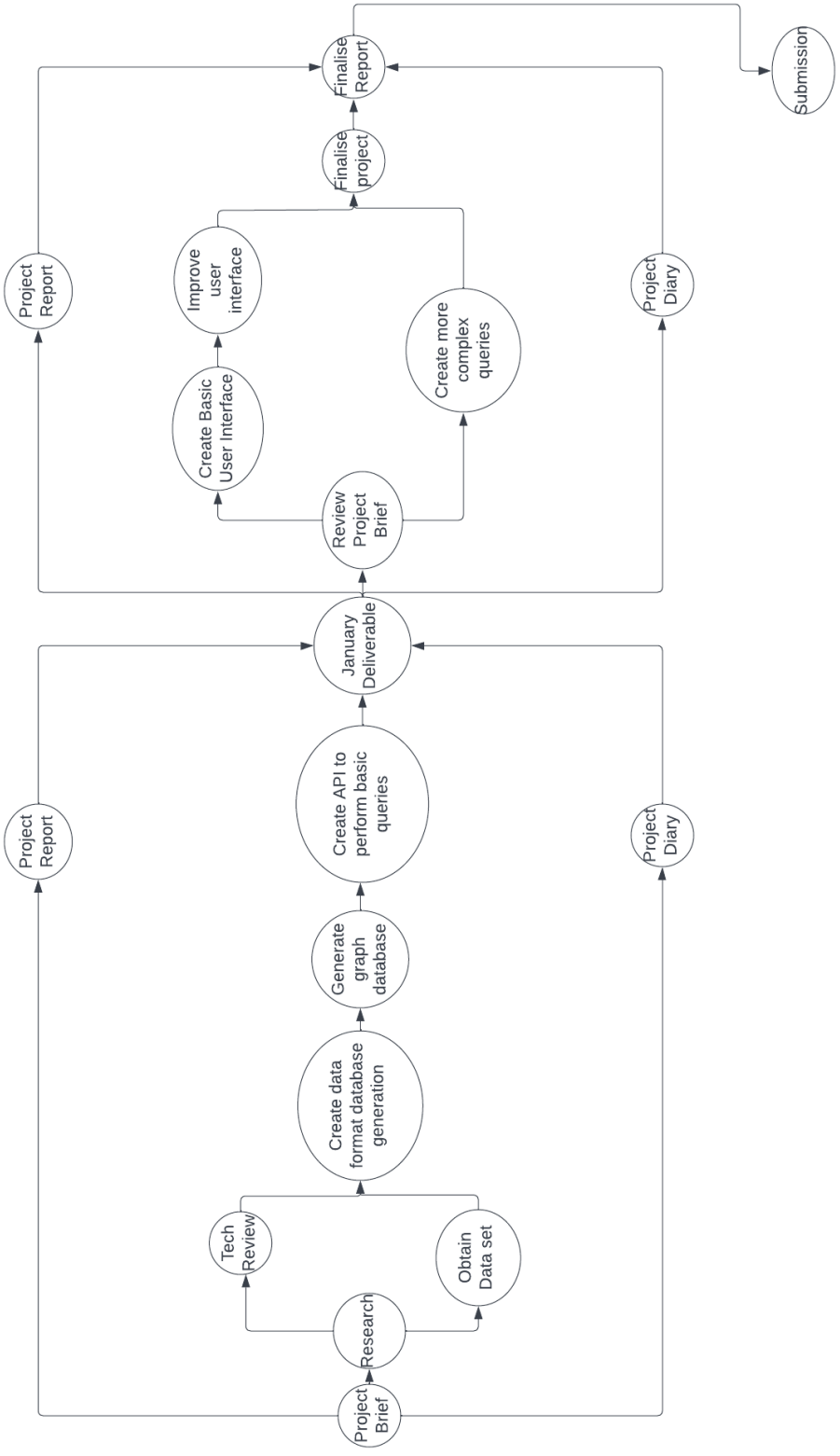
This project has no budget and so any services used in the development of the application will need to be free.

Dataset Availability

The data needed to create the database may become unavailable or unusable.

1.4 Project Plan

Figure 1.1: PERT chart showing project flow



Chapter 2

Background

2.1 Introduction

Through this chapter research will be conducted into varying areas related to the project. The conclusions made from this research will support any design or implementation decisions made in the course of the project.

2.2 The Binding of Isaac

'The Binding of Isaac: Rebirth is a randomly generated action RPG shooter with heavy Rogue-like elements.'[2] The player progresses through 'floors' which are made up of a series of 'rooms', each room can contain a variety of enemies, traps, and items. Each floor has a set of 'special rooms', namely a boss room, item room and a shop. The goal is to use the items found on each floor to defeat each boss, progressing to the next floor until the game is finished. The items in the game often interact, this interaction is usually called a 'synergy' if it benefits the player. In this instance the rogue-like elements are that the game has to be successfully completed many times, these are called 'runs'. Each run is unique and depending on the actions the player takes in the run it can have different outcomes and more parts of the game can be unlocked.

2.3 Existing Solutions

While there is no existing solution that is a direct comparison to this project, there are applications that have a similar purpose. These will be analysed to determine what features should also be implemented in this project and what could be improved by this project.

Fandom Wiki

Figure 2.1: Item page from Fandom Wiki

The Binding of Isaac has two wiki sites hosted on the Fandom Wiki platform; one for the original flash game[3], and one for the modern version, commonly

referred to as 'Rebirth'[4]. For the purposes of this project we will only be considering the modern version as it is widely considered the 'goto' version within the game's community.

The website contains comprehensive information on all aspects of the game, and it is continually updated by the community. Users can navigate the site using either predefined categories or a powerful search tool.

Advantages

- Contains information on all aspects of the game
- Actively maintained by the community
- Useful search functionality

Disadvantages

- So much information can make it hard to find what is relevant
- Unable to search for interactions, have to go via each item

Platinum God

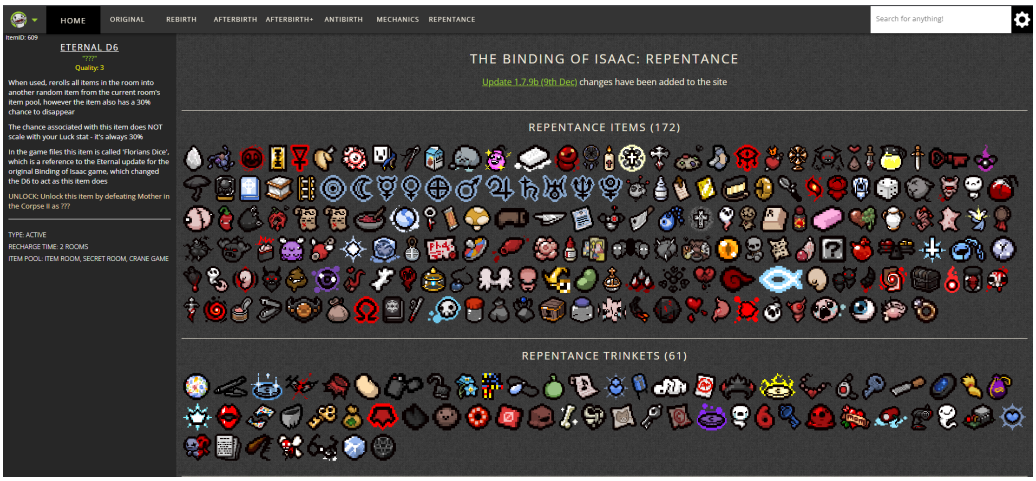


Figure 2.2: Platinum God main page

Platinum God is a self-described 'Isaac Cheat Sheet'[5], and it contains item and key mechanic information for all versions of the game. The site is maintained by one person, and it claims to be more accurate than the community wiki as its updates are 'tested thoroughly in the game using Cheat Engine'[6]. The information is split into pages based on the version of the game; users can navigate this using the item icons which are arrayed on the page, or by using

the search functionality. The search tool has some supported keywords, but will still usually require entering an exact match to an entry in the data. For certain versions of the game there is also a synergy finder tool which lets the user enter two items to see how they interact. However, this is limited to older versions of the game and only a small set of the items are actually included in the tool.

Advantages

- Information is more reliable than the community wiki
- Easier to reference quickly due to there being less information

Disadvantages

- Only one maintainer can mean long update times
- Only contains basic information about each item
- Limited or no synergy information for most items
- Harder to find items without knowing the name or what the item looks like

2.4 Technology Review

This section will research the potential technologies for the project and conclude with decisions about each.

2.4.1 Client Side Framework

Angular

'Angular is an application-design framework and development platform for creating efficient and sophisticated single-page apps.'[7] It was developed by Google to provide a complete framework for simple to complex single-page apps. Due to the size of this framework and the fact that it utilises Typescript makes for a steep learning curve, however this is counteracted by the large number of resources available from it being hugely popular and backed by a large company.

React

'React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called "components".'[8] React allows the developer to use JSX to combine HTML and JavaScript into one file, although this is not required to benefit from

React. Unlike Angular, this is a library and not a full framework. This will reduce the amount of learning required to use it, but it is likely extra libraries will be required to provide all the functionality required. There is also a lot of resources available as React is maintained by Meta and is the most popular of the frameworks considered according the 2022 Stack Overflow Developer Survey[9].

Vue

'An approachable, performant and versatile framework for building web user interfaces.'[10] While not backed by a large company, Vue has grown in popularity and has many large commercial sponsors. This means there is likely to be less documentation available, but this library is relatively simple compared to the options considered above.

Conclusion

After reviewing the three frameworks, the decision was made to use Angular for the project. This is due to prior experience with the framework, and with no background in JavaScript the increase in learning curve from Typescript is negligible. However, any of these choices would have been suitable for the project and the decision is mostly down to personal preference.

2.4.2 Server Side Framework

Due to the previously mentioned lack of JavaScript experience, the decision was made to only consider Python based frameworks to reduce the number of languages to learn.

Django

'Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design.'[11] As with Angular, this can be used as a full framework and so, while it will contain almost everything required, there will also be a lot of unnecessary features included. Django was designed to support rapid development, as shown by the slogan 'The web framework for perfectionists with deadlines.'[11]. It features database abstraction which allows developers to create 'Models' to represent the data stored in a database, handling communication with the database avoiding the need for coding in SQL.

Flask

'Flask is a lightweight WSGI web application framework.'[12] It is a micro-framework that started as a wrapper for Jinja and Werkzeug and has grown into a popular web framework. Flask does not provide much functionality that is already provided by another extension, this keeps Flask small but can introduce complex library requirements.

Conclusion

Django was chosen here, again due to prior experience, particularly in using Django and Angular together. While using two large frameworks may result in the project suffering from an amount of bloat, the prior experience means completing the project and having time to refine it is more likely.

2.4.3 Database

A graph database is a type of NoSQL database that uses nodes, edges, and properties to represent and store data. It is often described as storing data as it would be drawn on a whiteboard. This approach makes querying relationships in the data much faster as they are embedded in the data, rather than using JOIN operations or cross lookups often seen in SQL implementations. The underlying storage mechanism of a graph database can vary, some depending on an abstraction to store the data in a typical table based manner and some opting for a 'native' approach, maintaining the graph structure throughout the system.

Neo4j

'Neo4j is an open-source, NoSQL, native graph database that provides an ACID-compliant transactional backend for your applications'[13] Neo4j provide various tools for developing with graph databases, only the Aura platform will be considered here as it is the direct comparison for the other tools discussed, and a managed service is ideal for this project. A free database instance is provided in AuraDB for each account which allows up to 200000 nodes and 400000 relationships. Aura features a data import tool that allows developers to create a model of the database structure which can then be populated with data from CSV files. It also has an explore tool, Bloom, this allows developers to view the database graphically and queries can be performed to show expected outputs. Neo4j provide a lot of useful documentation and tutorials for all the services

they provide, this includes free e-books and course style content to guide users through the material.

Amazon Neptune

'Amazon Neptune is a purpose-built, high-performance graph database engine optimized for storing billions of relationships and querying the graph with milliseconds latency.'[1] Neptune is a managed graph database service provided by AWS. It offers a high throughput and low latency system that automatically scales with demand. As to be expected from any AWS product it is also highly secure and fault-tolerant. However, this comes at a cost; a free trial is offered for 30 days and after that a monthly fee is incurred with additional costs for data transfer, backups, and storage consumption.

Conclusion

The decision was made to use Neo4j. This is primarily because the Aura platform provides a permanent free database instance which has ample resources for this project. There also exists a Python libraries, neomodel and django-neomodel, for easily integrating database access using Django Models.

2.5 Conclusion

Throughout this chapter research has been conducted that will continue to provide benefit throughout the design and implementation stages. The review of existing solutions has highlighted the need for this application and has provided insight into how existing solutions have addressed the issue. The technology review has made it clear which technologies should be taken forward in the project as well as providing some background on why these technologies are needed and how they work.

Chapter 3

Design

3.1 System Design

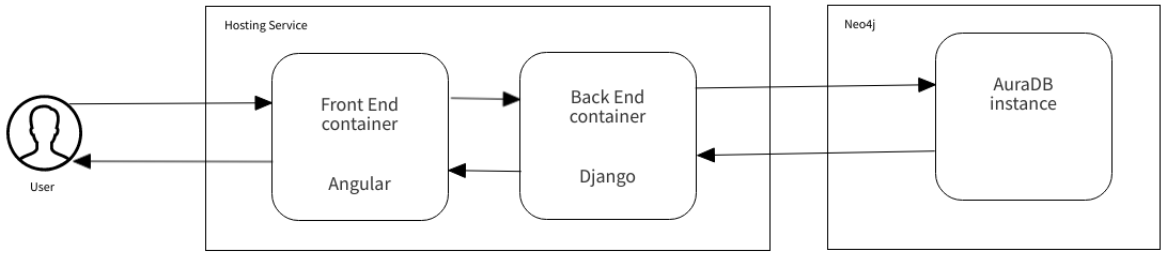


Figure 3.1: System Architecture Diagram

The system architecture for this project is similar to most web based applications as the only real difference is the type of database used. This means the user interacts with the web page created by the front end server, this then communicates with a separate back end server using the defined API. If needed the backend server can then communicate with the database, which in this case is hosted remotely by Neo4j. The front and back end servers would likely be hosted in the cloud via the use of some containerisation system such as Docker.

3.2 UI Design

Given that the data is stored in a graph database and that is the focus of this project, it seemed obvious to display the data to the user as a graph. This idea was further reinforced by looking at the Neo4j tool Bloom. As shown in the screenshots below, Bloom is used to display the data in a dynamic and

responsive graph. The main view area is called the "Scene" and just this alone allows the user to quickly visualise the data and see the relationships formed. In the scene the nodes can be dragged using the mouse and the graph updates using a physics simulation to move the other nodes. This helps ensure the other areas of the graph are still readable while still allowing the user to manipulate the graph.

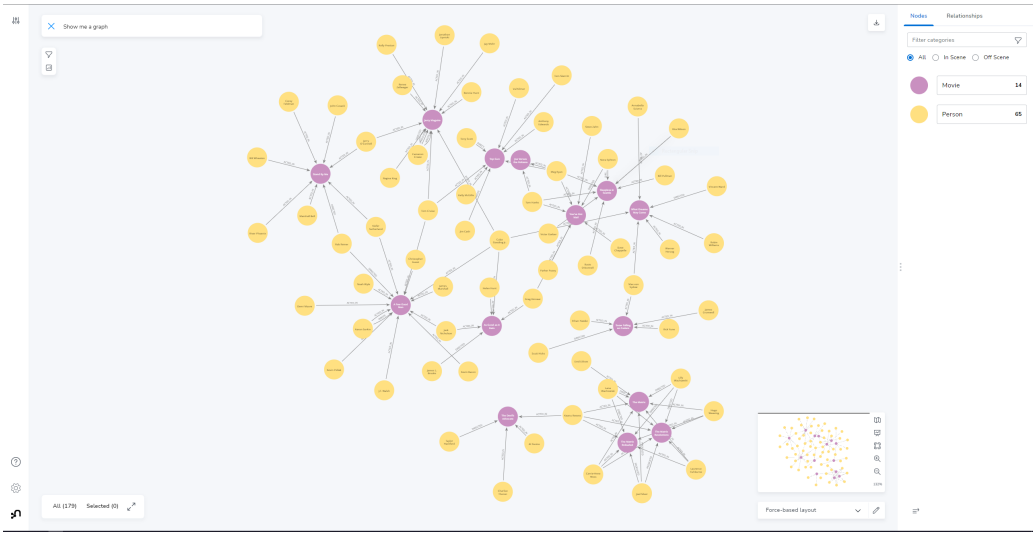
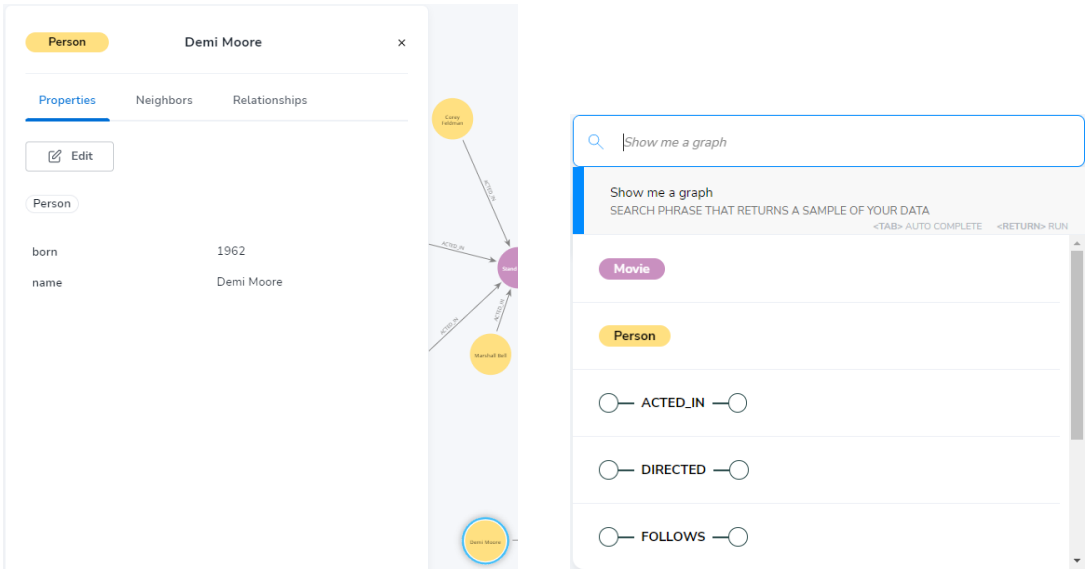


Figure 3.2: Neo4j Bloom using the demo Movies database

Figure 3.3 shows two key features of Bloom. The first is the 'inspect' element which allows the user to view the full properties of a node or relationship by double-clicking them. This means the graph itself is not cluttered with extra information but the user can still easily access that data when required. The other feature is the search bar which has a few unique functions that utilise the graph structure of the database. It uses the types of nodes and relationships in the database to provide some quick pre-filled searches, For example Actor - ACTED_IN - Movie with the demo database. This can be used to quickly filter out unwanted data from the scene without having to know how to write custom query statements. The second functionality follows on from this, in that the user can define their own query statements which become a command available in the search dropdown. This is useful if the user is familiar with Neo4j's own query language CYPHER, and they want to leverage that for more powerful querying.



(a) Bloom Inspect Element (b) Bloom Search Element

Figure 3.3: Neo4j Bloom inspect and search elements

These are all features that should be considered when implementing the user interface, however not all of them are essential for a usable application.

Chapter 4

Implementation

4.1 Introduction

4.2 Tools

4.2.1 IDE

Visual Studio Code was used for this project due to prior experience with the IDE. It is also one of the most popular tools in the industry as shown by the Stack Overflow Developer Survey[9] and the TOP IDE Index[14]. VS Code has excellent support for many programming languages, and with a wealth of community made extensions there are many tools to aid with development.

4.2.2 Version Control

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.[15] Using version control with an external hosting provider also makes working over several computers easy which will be useful for this project, and it ensures everything is backed up remotely. Git was chosen for version control as it is the industry standard, with GitHub being used as the hosting provider.

4.2.3 Database Visualisation

Neo4j has two tools for database visualisation as part of the AuraDB web interface; Bloom and Browser. Bloom is used to visualise the data in a graph as has been discussed in the design chapter previously. Browser is used to test CYPHER queries on the database. CYPHER is the query language created by Neo4j for retrieving data from their graph databases. As shown Figure 4.1, the user can enter a query and have the data returned as a graph, table (represented

as a series of JSON objects), raw text and as code (JSON objects). This is useful for quickly testing CYPHER queries and debugging database interactions.

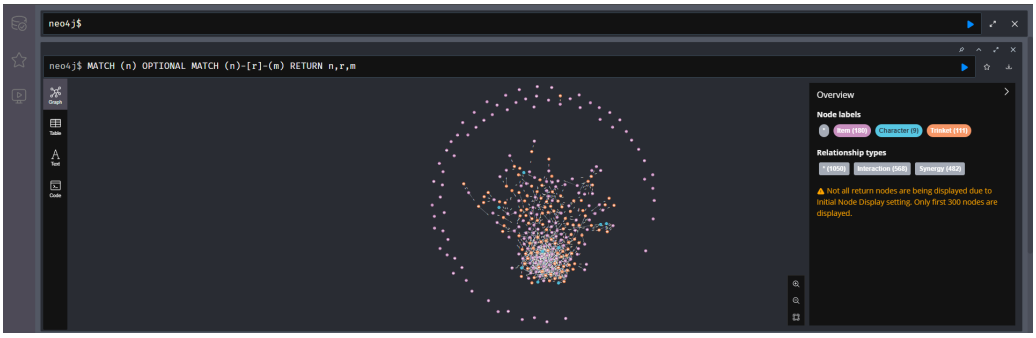


Figure 4.1: AuraDB Browser

4.3 Libraries

4.3.1 Data Processing

Beautiful Soup - A Python library for pulling data out of HTML and XML files[16]. Used to parse the XML dump and extract data for the database.

4.3.2 Client Side

Cytoscape - A JavaScript library that 'allows you to easily display and manipulate rich, interactive graphs'[17]. Used in this project to display the data in graphs to the user.

Material - An Angular specific library containing material design components, used in this project to quickly create UI components.

4.3.3 Server Side

Neomodel - An Object Graph Mapper (OGM) for the Neo4j graph database[18], used to define Django models for access database data.

django-cors-headers - A Django App that adds Cross-Origin Resource Sharing (CORS) headers to responses. This allows in-browser requests to your Django application from other origins.[19] this is required for the front end and back end to communicate properly.

4.4 Data Processing

The first major section of the implementation was setting up the graph database and getting the data required to do so.

4.4.1 Finding Data Source

The initial project brief suggested using the Fandom wiki for the data source as Fandom wikis have the option for downloading an XML dump from the `Special:Statistics` page. As discussed in section 2.3, the Fandom wiki is also the most comprehensive source of data about the game, especially regarding the item interactions. The XML dumps are not always kept up to date, so while a new dump was being requested other options for the data source were investigated. This included investigating the game files, where all the resource files have been packed in to `A` files. In the folder containing the resource files there is a readme which explains that this was done to prevent spoilers and any secrets being found through the files. A resource extractor tool is included with later versions of the game, however the files do not contain any information regarding the interactions of items.

Once received, the updated XML dump presented its own challenges, the first being its size, at just under 500,000 lines long and around 20MB it was too large for most text editors to load with syntax highlighting. This made it difficult to understand the structure of the data as the XML tags became harder to pick out from regular text. Aside from a small preamble, all the data in the file is contained in a series of `page` elements, which unsurprisingly represents a page on the site. Each page element has a similar structure to the below example (Fig. 4.2).

```
        <page>
        <title>Template:</title>
        <ns>10</ns>
        <id>161</id>
        <redirect title="Template:*" />
        <revision>
          <id>161</id>
          <timestamp>2014-09-16T20:27:12Z</timestamp>
          <contributor>
            <username>Maintenance script-gpuser</username>
            <id>41555837</id>
          </contributor>
          <comment>&amp;lt;default import&amp;gt;</comment>
          <origin>161</origin>
          <model>wikitext</model>
          <format>text/x-wiki</format>
          <text bytes="24" sha1="2dwqfi7oey6311xkwxu3dhjasn8gfsi" xml:space="preserve">#
    redirect [[Template:*]]</text>
          <sha1>2dwqfi7oey6311xkwxu3dhjasn8gfsi</sha1>
        </revision>
      </page>

```

Figure 4.2: Example page format

The important parts to note from this is that the `title` element is the web page title and the `text` element contains the data to be displayed (or in this case a redirect to another page). There doesn't appear to be a particular order to the pages and due to the formatting of the file it can be hard to tell where one page ends and another begins.

4.4.2 Data Extraction

Due to the structure of the XML data, the easiest way to extract the data is to use the title tags for each page to find the relevant and pages and then pull the data from the text tag. The XML file contains collections pages which each contain a list of items available in each version of the game, this can be used to get a list of all the item names. Character names are fetched from the `Characters` page. This has a list of links to the character pages, from which the names can be extracted using RegEx. Unfortunately, the same does not exist for trinkets as the page that would contain that list instead uses a template which autofills the data. As a temporary fix, the list of trinkets is fetched from a hardcoded file.

```
        ITEM_REGEX = re.compile("content =(.*?)}}", flags=re.DOTALL)

        def _get_item_names(self):
            collection = self.soup.find("title", text="Collection Page (Repentance)").
    find_parent("page").find("text").text
            return [
                x.strip()
                for x in ITEM_REGEX.search(collection)[1].replace("\n", "").replace("Number
    Two", "No. 2").split(",")
            ]

```

Figure 4.3: Function that returns the list of item names

Figure 4.3 shows the code used for getting the list of item names, similar code has been used for getting character names. `re` is a Python library that provides regular expression matching operations. In the data extraction script a series of compiled RegEx statements are defined as constants, `ITEM_REGEX` being one of them. These are used through the script to pull data from the XML file based on the format of the data. Here BeautifulSoup is used to find the title tag that contains the text 'Collection Page (Repentance)', this is then used to get its parent and to search that for the text element. To get the data from the string of text the RegEx statement is used to search for a string that starts with `content =` and ends with `}}`. The `(.*?)` matches a string of any characters of any length non-greedily and puts it in a match group. This means it will match against any character until the first instance of `}}` and the object returned is sub-scriptable to get the match groups. Accessing group 0 is the whole match which will included the `content =` and `}}`, accessing group 1 is the first (and in this case only) match group. The string returned by accessing the match group then has new line characters removed and and is split into a list using the commas. Each value in this array then has any leading and trailing whitespace removed using `strip()` and a list comprehension. There is an extra step needed here due to the nature of the XML data and that is manually replacing the item name for 'Number Two'. This had to be done as the name appears in several formats throughout the data, but the only one that has a page with data in is 'No. 2'. The rest of the pages just redirect back to that page and so instead of handling the redirect it is easier to catch and replace the different formats when they appear.

Functions could now be written for getting item, trinket and character data respectively. These functions all follow the same format and the differences are

in catching special exceptions and the output created.

```
        def get_all_items(self) -> list:
        item_names = self._get_item_names()
        tags = self.soup.find_all("title")
        item_data = []
        for tag in tags:
            if tag.text not in item_names:
                continue

            item_text = tag.find_parent("page").find("text").text

            item_id = self._infobox_get(item_text, ID_REGEX)
            if item_id is None:
                continue

            item_data.append(
                [
                    tag.text,
                    f"I{item_id}",
                    self._infobox_get(item_text, QUOTE_REGEX),
                    self._infobox_get(item_text, DESCRIPTION_REGEX),
                    self._infobox_get(item_text, ITEM_QUALITY_REGEX),
                    self._infobox_get(item_text, UNLOCK_REGEX),
                    self._list_get(item_text, EFFECTS_REGEX, True),
                    self._list_get(item_text, NOTES_REGEX, True),
                ]
            )
            self.id_lookup[tag.text.lower()] = f"I{item_id}"
            self.synergies[tag.text] = self._list_get(item_text, SYNERGIES_REGEX)
            self.interactions[tag.text] = self._list_get(item_text, INTERACTIONS_REGEX)
        return item_data

```

Figure 4.4: Function that extracts the item data

Each function iterates over the title tags in the XML file and checks if it is in the list of tags being looked for. If it is the text element belonging to that title tag is fetched and an attempt at getting the item ID is made. This is done by passing the relevant compiled RegEx statement and the text to process to a function which will execute the RegEx and return `None` if no match is found and the ID string if a match is found. If no ID is found that means the page found isn't actually an item page and so can be ignored. The rest of the data can then be extracted using the same method as for getting the ID, the only difference being for the effects and notes data another function is used that has extra processing for handling lists in the text. The synergy and interaction data is added to a separate dictionary as it requires extra processing to create usable data. The item name and ID is added to a lookup dictionary which will be used later when processing the synergies/interactions. Importantly, for each ID a letter prefix is added to indicate whether the ID belongs to an item, trinket or character; this is needed to ensure the IDs are unique across the entire

database.

The function for getting character data is much simpler than for items and trinkets, this is because it only gets the character name and ID. The decision was made to ignore character data because it is much more complicated than any of the other data types. The page for each character has a unique structure, especially for characters that are a combination of two characters such as Jacob and Esau or The Forgotten. Combined with the fact that most of the character data does not contribute to the relationships in the database which is the main focus, it was decided that processing the character data was outside the scope of this project.

Once all the item, character, and trinket data has been extracted the lookup dictionaries can be used to process the synergies and interactions. The function to process the data is fairly simple, it iterates over each entry in the dictionary, each entry contains a list of strings. That list is iterated over and the RegEx show in figure 4.5 is applied to create a list of item names which represent the relationship destination. The RegEx looks for tags that link to other items/characters/trinkets, for example i—Libra. However, these tags don't seem to always follow a set structure which is why the RegEx has some extra match groups either side of the `(.+?)` group to catch and ignore extra characters that are not relevant. These irregularities also mean an if block is needed to catch links where the link text does not match the text in the title tag.

```
1         destinations = re.findall(r"{{[ict]\|(1=)?(.+?)(\|.+?)?}}", relationship[0], re.
      IGNORECASE)
2
```

Figure 4.5: Synergy/Interaction RegEx

The ID lookups are then used to find the ID for the source entity and the destination entity found via the RegEx. This is then added to the output along with the string containing the relationship data.

4.4.3 Cleaning the Data

The data extracted above still contains special characters and formatting that is used by the XML to generate the website correctly. This includes `<br>` HTML tags, tags using square brackets to denote alternative text/images and the curly brace tags seen above which are used for a variety of purposes. One of the uses

of curly brace tags is to show which version of the game the information is relevant to; on the webpage this is show via a small image but in the XML that image is represent by a tag in the following format `{{dlc|<tag>}}` where `<tag>` is replaced by a code that represent the game version. These tags are replaced with a relevant string using RegEx, e.g. `{{dlc|anr}}` becomes '(Added in Afterbirth, Removed in Repentance)'. RegEx is then used to extract the useful text from the rest of the tags and remove the brackets and any other formatting characters.

The lists used when processing synergies and interactions have to be generated from the string return by the data extraction. In the XML the bullet pointed lists are represented using '*' characters where the number of characters indicates the level of indentation. The string is split using the bullet characters and then a recursive function is used to create a nested list that represents the list indentation. Optionally, the function can also take the nested list and create a formatted string with tab spacing to create the indentation.

4.4.4 Importing into Database

The AuraDB platform provided by Neo4j has a data importer tool which uses CSV files to populate a predefined model. Creating CSV files with Python is very simple, the data needs to be in a 2D array where each inner array is a row in the CSV file, the CSV writer also takes a list of headers and from the two it can create a CSV file. The CSV files needed to be created so that each file represents a node/edge in the model, this meant 5 CSV files were needed. With the files uploaded to the import tool the model shown in figure 4.6 could be created.

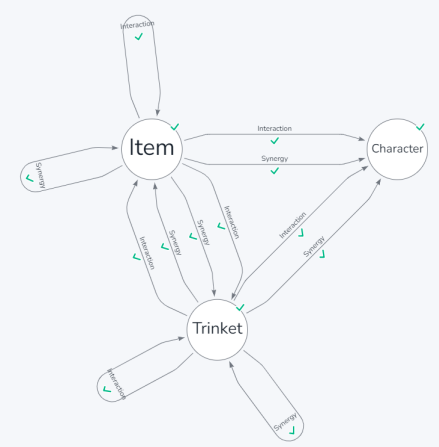


Figure 4.6: Database Model in Neo4j Data Importer

As the model shows items and trinkets can have interactions and relationships between each other and themselves. The relationships to character entities are only towards the character nodes because the character data is not processed as discussed above. After importing the data Neo4j Bloom can be used to check the database structure which is show in figure 4.

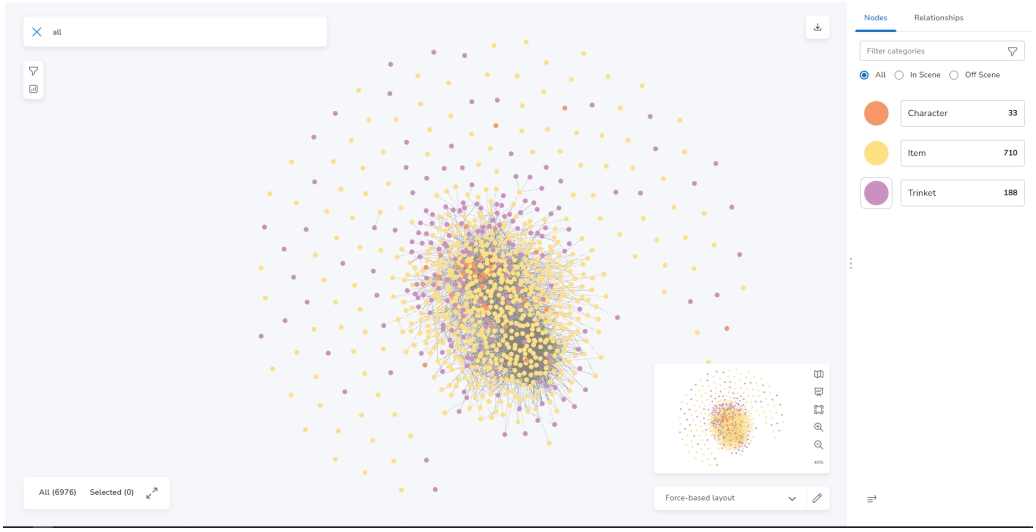


Figure 4.7: Bloom showing complete database

4.5 Web Stack

Before the database can be utilised the Angular and Django projects need to be set up and connected to form the web stack. The first step is to create a Django project, this is achieved using a single command provided by the Django Python library. This command creates the base folder structure for the project

which includes the management file, an SQL database and several configuration files. The `manage.py` file is used to perform most admin actions, commonly this is keeping the database migrations up to date and running the web server; in this project only the latter is needed due to using a different database. Out of the configuration files generated, `settings.py` is the only one that needs configuring at this stage. Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources.[20] This needs enabling on this project so that the Angular app can send requests to the Django web server from another domain. Enabling CORS headers in Django requires the installation of the django-cors-headers Python library and some additional some entries in the `settings.py` file. Now a Django app can be created in the project using a command similar to the one that created the project. The app folder contains the models, views and URLs files that will contain all the functionality of the backend. `models.py` is the where the database interaction is defined, usually this is done via model classes and serialisers supplied by Django. However, due to using a database hosted by Neo4j this will be accomplished using the neomodel Python library. The `views.py` file sits on top of the models file, it contains the functions that take in HTTP requests, perform some action which may involve using the models to interact with the database, and then generate a response. These functions are called by the paths declared in the `urls.py` file, each path defines a URL pattern and the view function that should be called to handle it. Some final changes need to be made to the configuration files to finishing setting up the Django app and then the backend is ready for development. The first change is to add some paths to the `urls.py` in the main project folder, this is to set which URLs should be directed to the app that has just been defined. The other changes are in the `settings.py` file and simply involve including the new app name in the `INSTALLED_APPS` section.

```
DjangoAPI
├── DjangoAPI
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── QueryApp
│   ├── admin.py
│   ├── apps.py
│   ├── models.py
│   ├── tests.py
│   ├── urls.py
│   └── views.py
└── manage.py
```

Figure 4.8: Folder structure after creating Django project

Creating the Angular project follows a similar set of steps to creating the Django project. First a new project is creating using a command provided by the Angular library, this creates quite a large folder structure which can be very overwhelming. However, only a handful of these files are used over the course of the project. Before configuring any of these, the app components and services need to be created, this is again done using commands provided by Angular which generates the following folder structure for each component.

```
example
├── example.component.css
├── example.component.html
└── example.component.spec.ts
    ├── example.component.ts .3 example.service.spec.ts
    └── exmaple.service.ts
```

Figure 4.9: Example component folder structure

Each component has a structure similar to a normal HTML/CSS/JavaScript application, the exception being TypeScript is used instead of JavaScript. The additional component file (example.component.spec.ts) is where unit tests should be written, and it includes some basic test structs as part of the file generation. `service.ts` files contain classes and functions dedicated to interacting with the back end, for this project there is a shared services file at the app level and then each component has a service which inherits from the shared one. Services also have `spec.ts` file for API unit tests. A final commonly used file type is an interfaces file, this is a regular TypeScript file that contains many interface objects which are exported. These are used to define custom data

types/objects which are needed as TypeScript is a typed language and the data the front end receives can be in many formats. Once the components have been created it's important to update `app.module.ts`, this file defines the root module and its contents. If a component, service or imported library is not correctly defined in this file, then the project will fail to compile as it will not be able to find all the files required. Next `app-routing.module.ts` will need configuring, similar to the `urls.py` files in Django, this defines routes as a path and which component should be called for that path. This information is then used in the `app.component.html` via the `<router-outlet>` element, which will contain the HTML of the corresponding component for the current URL. The HTML file can also contain HTML elements that are to be displayed across all pages, such as a navigation bar. The final file of interest is `tsconfig.json`, which is a configuration file that contains compiler options. This often will not require any changes, however sometimes it is useful to disable compiler errors, especially when working with third party libraries.

Finally, to connect the Angular and Django project to allow requests and responses to be sent between them, a read only constant is defined in the shared service of the Angular component. This constant contains the IP address of the web server created by the Django project, it can then be used later when performing HTTP requests to direct them to the back end.

4.5.1 Database Interaction

Neomodel was used to allow Django to connect to the remote database. The username, password and database URI are stored as constants which are passed into the neomodel configuration. Then classes can be defined which represent each of the node and edge types. For node classes, class attributes are defined for each property of the node and for each relationship type the node can have. The relationships only need to be defined in one direction and so most of the relationships are defined in the `Item` class. The edge classes only need to define the properties of the relationship. These classes contain built-in methods for performing simple queries on the data, for more advanced queries neomodel has a method for querying with user defined CYPHER queries.

4.5.2 Displaying Data in Tables

To help ensure the database was functioning properly and to provide an opportunity to learn the intricacies of neomodel in a simpler environment, the

decision was made to implement displaying the data in tables first. This would require fetching all the data for each node and edge type and formatting it correctly. Getting all the data for a set of nodes is easy with neomodel due to the `nodes.all()` method, which returns a list of all the nodes for the given node type. However, this data was not in a format accepted by the front end. The list contained instances of the model class which have string representations which meant it could be printed in the correct format, but that would not the format returned in the HTTP request. Instead, a format function was written which extracts the data into a Python dictionary which can then be returned in a JSON response. This method was suitable for node data, however relationships do not have a method equivalent to `nodes.all()`. To get the data for all the relationships of one type a custom CYPHER query has to be used, as shown in figure 4.8 for synergy data.

```
1         neomodel.db.cypher_query("MATCH (n)-[r:Synergy]->(m) RETURN n, r, m")
2
```

Figure 4.10: Initial CYPHER query for getting synergy data

Formatting the data returned by this query would take a long time, requests would take approximately 8 minutes to return. This was interesting as the formatting simply iterated over the list of objects and extracted the relevant to into a dictionary, something that isn't usually a computationally heavy task. After some preliminary troubleshooting, the issue was left in the code as it was likely to change when displaying the data in graphs as intended and the tables were only intended as a tool for learning and debugging.

Displaying the data in the front end required writing the functions that send requests to the URLs defined in the back end. These functions are in the shared services file described previously, each function sends a get request to a URL and returns an `Observable` object. Then in the TypeScript file for the component this object can be subscribed to so when the data is received it is stored in a class attribute. This subscription first happens when the page is first initialised, this implemented via the `ngOnInit()`. To get this data into a Material table it needs to stored in a `MatTableDataSource` which can then be set as the data source for the table in HTML as shown below.

```
<table mat-table [dataSource]="dataSource">
    <ng-container matColumnDef="name">
        <th mat-header-cell *matHeaderCellDef> Name </th>
        <td mat-cell *matCellDef="let item"> {{item.name}} </td>
    </ng-container>

```

Figure 4.11: Item table excerpt

As show in figure 4.11 the table is defined using a series of `ng-containers`, each container describes a column. In each container the usual table header and table data tags are used, however now `*matCellDef` can be used to get the data to be displayed in the table using the data source. The tables for each data type are defined in the HTML, and they are all wrapped in Material tab groups. This allows tabs to be displayed at the top of the page to switch between tables, using Material to do this means this can be done using just HTML which highlights the main benefit of the library. The data for each table is only fetched on the first time the tables tab is selected, this reduces loading time when loading the page. The data is stored in a variable after the first request, so that subsequent selections do not have to re-request the data. It does still take the 8 minutes to load synergy and interaction data as mentioned in the previous section.

After implementing these tables it was discovered that the ID being returned was incorrect, instead of being the ID property of the node the element ID from the database was being returned. These are different in Neo4j, the property being the ID extracted from the XML data and the element ID is the ID automatically assigned to every entity in database on creation. After some investigation it was found this was caused by the use of the models defined using neomodel. To fix this the queries all had to be rewritten using CYPHER queries instead to avoid the data being passed through a neomodel object, instead the database query now returns a list of `neo4j.graph.node` objects. The query response is a list regardless of the number of objects in the response and each object itself is wrapped in a list, while annoying this does not add any real complexity to processing the data. The object itself acts like a normal Python dictionary, so values can be extracted using their IDs which match the field names in the database.

4.5.3 Displaying Data in Graph

There are a variety of JavaScript libraries for representing data in a graph, the most notable is D3.js. D3 is a comprehensive library for creating dynamic data visualisations and allows for the creation of force simulated graphs that update in real time. However, this is quite complex and with the additional complexity of using a JavaScript library in TypeScript it quickly became unfeasible to use. Instead, the decision was made to use Cytoscape.js which can create graphs using similar force simulations as D3, but they are not dynamic by default. Cytoscape has a number of pre-made layouts that can be imported to quickly create basic graphs without the need for extensive configuration. Before a graph can be generated the back end had to be updated so that data could be returned in the format required by Cytoscape, which is shown in figure 4.12.

```
{
    nodes: [
        {data: {id: <id>, name: <name>}},
        {data: {id: <id>, name: <name>}},
        ...
    ],
    edges: [
        {data: {id: <id>, source: <source>, target: <target>, name: <name>}},
        {data: {id: <id>, source: <source>, target: <target>, name: <name>}},
        ...
    ]
}

```

Figure 4.12: Data format expected by Cytoscape

As all the data needed to generate the graph is needed at one time, it would be efficient to use a CYPHER query that gets all that data at once rather than combining several smaller requests. Initially, a CYPHER query was used that fetched all the data in the database, however this includes many nodes that have no relationships which clutter the rendered graph. So instead of fetching all the data, a CYPHER query was used that fetches all the data about related nodes, this is the query statement in figure 4.13.

```
MATCH (n)-[r]-(m) RETURN n, r, m;

```

Figure 4.13: CYPHER query to get graph data

Once the data being sent by the back end is correctly formatted, rendering the

graph requires very little code. Inside the subscription that gets the data a variable is defined that contains a Cytoscape object which takes 4 parameters: container, elements, style, and layout. Container is the HTML element the graph should be rendered in, elements is the data for the graph, style is optional and contains a CSS-like style sheet, and layout specifies which imported layout to use. The style sheet for the graph is stored in a separate file and imported as it can get very large which reduces code readability. Unfortunately, getting the data from the back end still suffers from long waiting times as before, with this larger query often taking close to 20 minutes to return data.

As the loading issue still persists with the queries needed for the graph some further debugging was needed to find the issue. Upon initial investigation it became clear that the database query returned almost instantly and so was not the issue, rather it was the processing of the returned data in Python. However, from surface level inspection this was also confusing as the Python code simply looped over the list of returned objects and extracts the relevant data into a dictionary. This is not something that would usually be computationally demanding, which raised suspicion that accessed the neomodel objects created by the query was the source of the issue. To test this initially, the ID values that are fetched multiple times per loop were stored in a variable, so they only had to be fetched once. This noticeably reduced the loading time, but it was still taking too long to be usable. The CYPHER query was returning a lot of data with just over 6000 objects, so to reduce the amount of data that needed to be processed the CYPHER query was re-written to only return the properties needed for the graph.

```
1 MATCH (n)-[r]-(m) RETURN n.id,n.name,labels(n),m.id,m.name,labels(m);
2
```

Figure 4.14: Refined CYPHER query to get graph data

This change had the unexpected added benefit of stopping neomodel resolving the data into objects and instead returning the data as a nested list of items where each inner list is the values specified by the RETURN in order. This meant that instead of accessing an object every time data is extracted, it is a simply accessing an array via index. While this does make the code less readable, it means the request returns almost instantly. At this stage the `labels()` were added to the query, this returns the type of the node, i.e. Item. This is

sent as part of the graph data for a couple of purposes, but at this stage it is just used as a way of specifying different styles for different node types.

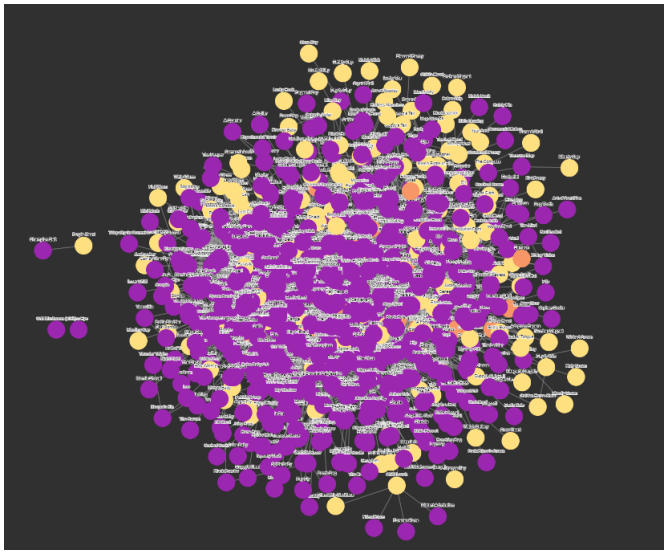


Figure 4.15: The final rendered graph

Now the graph is complete and loading data is much faster, it was decided that the table view created previously should be included as part of the final site. To do this a toolbar was added using the Material library which contained a menu button which opens a side navigation menu. This is added in the `app.component.html` file so that it is visible on both the graph and table pages.

4.5.4 Inspect Element

The first feature to be implemented for getting more detailed data from the graph is the inspect element. This should display a card on screen when the user clicks a node or an edge which contains all the data for the corresponding entity in the database. On click events can be added to cytoscape elements using the `on()` function which takes the event type, element type and a function that should be called. After attempting to implement some simple on click events that would just output the fetched data to the console, a major issue became apparant. Currently, the variable for the graph is defined inside the subscription that fetches the data, which means the on click events would need to be defined in the same scope. However, these functions also need to call a subscription which causes an error and is just poor coding practice. One solution would be to declare the variable outside the subscription so that it can be accessed by the on clicks, but that results in the on click functions attempting be defined

on a variable that isn't defined yet. This is because there is nothing telling the script to wait for the subscription to return data and finish before moving to the next command. The correct solution to this is to rewrite the service functions so that the `async` and `await` keywords can be used. To do this the functions in the service file were updated to return a `Promise` which contains a resolve callback with the data from the HTTP request. Then async functions in the components TypeScript file were defined that call the service function with the `await` keyword. This means the elements definition for the graph can be replaced with the async function call and the subscription wrapping the graph definition is no longer needed. It also means the `then()` function can be used on the async function call, where the code in the brackets is only executed once the promise is resolved. This second feature is used in the on click functions so that the data is available before the code attempts to use it.

Now the data is correctly retrieved it needs to be displayed in a UI element; the Material library has card components that are designed for displaying this kind of information. However, just using a normal card component will mean a portion of the screen would have to be left blank for the card to be rendered in, this is space that could be used to display the graph. Luckily, the Material CDK (Component Dev Kit) has the overlay component which is designed for rendering elements over existing ones. Unfortunately, the documentation provided by the library is very sparse, and the code examples given were often incomplete or did not cover the functionality needed. However, it is clear that there are 2 main ways of using overlay components; the first is to create a `ng-template`, HTML element which would contain the card component HTML. This template is tied to a boolean variable that toggles whether the element should be rendered or not. This is the simplest of the methods, but it cannot be used in this instance as the data isn't available until an entity is clicked and so the template errors even though it isn't being rendered. The second method uses TypeScript to generate the overlay component and custom components to handle the content. The code in figure 4.16 shows the code that creates an overlay component when an Item node is clicked.

```
        this.getItem(evt.target.id()).then(
                (data) => {
                    if (this.currentOverlay) {
                        this.currentOverlay.destroy()
                    }
                    const overlayRef = this.overlay.create(this.overlayConfig);
                    const portal = new ComponentPortal(ItemComponent);
                    this.currentOverlay = overlayRef.attach(portal);
                    this.currentOverlay.instance.itemData = data;
                }
            );

```

Figure 4.16: Code inside the function called when an Item node is clicked

First it checks if an overlay already exists, if it does then it is removed so that there is only one overlay on the screen at a time. Then an `OverlayRef` instance is created using a predefined overlay config. The overlay config is just a dictionary like object that can contain various settings for the overlay, here it used to set the `PositionStrategy` and some CSS which controls where the overlay is rendered and it's maximum size. A `PositionStrategy` is another part of the Material CDK which is poorly documented; as the name suggests it defines where the component is positioned on the screen. In this instance a global position strategy is used which allows the overlay to be positioned anywhere on the screen regardless of other components. There are other position strategies for positioning the element in relation to another component, which is often useful for creating a button drop down for example. Overlays display content using `ComponentPortal` objects, so a new component needs to be created containing the content to be shown in the overlay. The `ItemComponent` being passed into the component portal is a custom component that is defined in the same way as the graph and table components have been defined. The HTML file contains the card element that will display the data, the TypeScript file defines the component and the variable that will contain the data. These components do not have their own CSS files and instead are directed to use the CSS file for the graph component, this is so the same CSS definitions can be used across all the elements. Now the portal can be attached to the `OverlayRef` object which means the overlay is actually displayed on the screen. However, it doesn't have any data as the data sent by the back end has not been passed from the graph component to the child component. The final line in the code snippet sets the variable in the child component mentioned before, this is possible because the variable was defined with the `@Input()` tag, which allows data to be passed

from a parent component to a view child.

Figure 4.17: An example of the overlay element

4.5.5 Searching

explain the different types of search redraws the graph

4.6 Conclusion

Chapter 5

Evaluation

5.1 Introduction

5.2 Project Evaluation

5.3 Future Work

5.4 Lessons Learned

5.5 Conclusion

Appendix A

Appendix

References

[1] *What Is a Graph Database?* Amazon Web Services, Inc. URL: https://aws.amazon.com/nosql/graph/ (visited on 01/06/2023).

[2] *The Binding of Isaac: Rebirth on Steam.* URL: https://store.steampowered.com/app/250900/The_Binding_of_Isaac_Rebirth/ (visited on 01/31/2023).

[3] *The Binding of Isaac Wiki.* URL: https://bindingofisaac.fandom.com/wiki/The_Binding_of_Isaac_Wiki (visited on 01/09/2023).

[4] *Binding of Isaac: Rebirth Wiki.* URL: https://bindingofisaacrebirth.fandom.com/wiki/Binding_of_Isaac:_Rebirth_Wiki (visited on 01/09/2023).

[5] *Isaac Cheat Sheet - Platinum God.* URL: https://platinumgod.co.uk/ (visited on 01/09/2023).

[6] *Frequently Asked Questions - Isaac Cheat Sheet - Platinum God.* URL: https://platinumgod.co.uk/faq (visited on 01/09/2023).

[7] *Angular - Introduction to the Angular Docs.* URL: https://angular.io/docs (visited on 01/09/2023).

[8] *Tutorial: Intro to React – React.* URL: https://reactjs.org/tutorial/tutorial.html (visited on 01/09/2023).

[9] *Stack Overflow Developer Survey 2022.* Stack Overflow. URL: https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022 (visited on 01/10/2023).

[10] *Vue.Js - The Progressive JavaScript Framework — Vue.Js.* URL: https://vuejs.org/ (visited on 01/09/2023).

[11] *Django.* Django Project. URL: https://www.djangoproject.com/ (visited on 01/09/2023).

[12] Armin Ronacher. *Flask: A Simple Framework for Building Complex Web Applications.* Version 2.2.2. URL: https://palletsprojects.com/p/flask (visited on 01/09/2023).

[13] *What Is a Graph Database? - Developer Guides.* Neo4j Graph Data Platform. URL: https://neo4j.com/developer/graph-database/ (visited on 01/09/2023).

[14] *TOP IDE Top Integrated Development Environment Index.* URL: https://pypl.github.io/IDE.html (visited on 04/23/2023).

[15] *Git - About Version Control.* URL: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control (visited on 04/23/2023).

[16] *Beautiful Soup Documentation — Beautiful Soup 4.4.0 Documentation.* URL: https://beautiful-soup-4.readthedocs.io/en/latest/ (visited on 04/25/2023).

[17] Max Franz et al. "Cytoscape.Js: A Graph Theory Library for Visualisation and Analysis". In: *Bioinformatics* 32.2 (Jan. 15, 2016), pp. 309–311. ISSN: 1367-4811, 1367-4803. DOI: 10.1093/bioinformatics/btv557. URL: https://academic.oup.com/bioinformatics/article/32/2/309/1744007 (visited on 04/24/2023).

[18] *Neomodel Documentation — Neomodel 5.0.0 Documentation.* URL: https://neomodel.readthedocs.io/en/latest/ (visited on 04/24/2023).

[19] Otto Yiu. *Django-Cors-Headers: Django-Cors-Headers Is a Django Application for Handling the Server Headers Required for Cross-Origin Resource Sharing (CORS).* Version 3.14.0. URL: https://github.com/adamchainz/django-cors-headers (visited on 04/24/2023).

[20] *Cross-Origin Resource Sharing (CORS) - HTTP — MDN.* May 10, 2023. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS (visited on 05/11/2023).