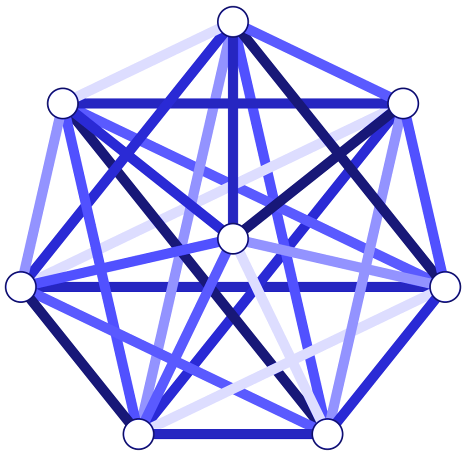
Clik



CSCI 6600

Database Management Systems

Participants

Group 3

Kelle Clark

Gregory Sutton

Section 1: Introduction

Purpose

The purpose of this document is to serve as a resource outlining the life cycle of the application Clik. Important notes from the phases of development are included in the sections that follow from the conception, stakeholder identification, user and use case exploration, requirements analysis, implementation and testing of the project. All code and documentation are available at the GitHub https://github.com/KelleClark/database_project

Intended Audience

This report is intended to be used by Dr. Ding and the members of the team as part of the conceptualization phase for the Database Management Team Project.

Section 2: Project Proposal

Section 2.1 Promoting Real Social Connections

An objective of social media platforms is to connect members with individuals and groups who share similar interests or connections. While FB, Twitter, LI, Reddit and Instagram provide users with an enjoyable and useful service, many believe that these social media sites make users less social. Current social applications available indirectly promote more time behind the screen, encourage individuals to make large and superficial connections, and enable the consumption of curated information that does not reflect reality. This project aims to provide users with some of the functionality of benchmark social media applications but with a focus on a design that improves real interaction between users. The proposed system would include a three tier design implemented as a locally hosted web application for the scope of this semester-long project.

The overarching goal for the system, as viewed by the team, is to bring people together in authentic ways that foster real relationships off screen. The application's use cases will be constructed in early stages of development as the team explores the personas of typical users and the tasks and goals of those users. Possible use cases are to enable the user to create small social circles and to enable the user to engage in face-to-face interactions in real-time.

Section 2.2 The Plan

The development of the application will follow the Agile methodology with frequent and brief updates regarding the project as a whole and the use of sprints with deliverables aimed to generate momentum toward the project's completion. The team will communicate through Microsoft Teams and have set up both a GitHub and Overleaf shared repositories for version control and collaboration.

As the team members have worked well together on a previous project in a Software Engineering course in the Fall of 2020, the team has already established good communication and mutual trust.   Kelle and Greg are also currently taking Web Application and this course, Database Management, providing them with the skills necessary to develop every component of this project.

The team will devise a timeline for the phases and sprints for the project that allows for iterative development and testing with a presentation preferred on April 15th. The first sprint has already begun for the team and entails the choice of DDL/DML and developing environment. Other sprints will involve identifying the users of the system, outlining the functional requirements and use cases, developing context diagrams, workflow diagrams and use case diagram, creating the schema of the required databases, building prototypes of the website, construction of test cases and logging changes for the system, implementing the front-end using html, CSS and bootstrap, jQuery and JavaScript.

Section 2.3 The Deliverables

1. A database to capture members, messages, events, and interactions. This database will also capture relationships between entities.

2. A web based platform that recommends activities to friends and groups of friends including: group chats, events, and the introduction of new friends to existing groups that share similar interests.

3. The documentation for the application that describes the relational schema, a user's guide and the testing performed on the system's functionality.

Section 3 The Stakeholders

This system is being constructed to assist users who desire to connect with local friends by attending events: getting a coffee or playing in a pick-up game of soccer, hiking a trail or browsing a gardening outlet for great deals. Loneliness and isolation are the problem, and this application aims to help a person connect with others by offering the person a way to create small groups of “local” people they know (knowds and pronounced “nodes”), offer up and show interest in activities (events) and get out from behind the screen ...wearing a mask...and get out from behind the devices that isolate people.

It follows that the primary stakeholder of this system is a user who can benefit from broadening their circle of friends. These friends can enjoy simple acts together like a walk in the park, browsing a nursery for plants, throwing a frisbee, or maybe hitting some range balls. The secondary stakeholders are the team members who will learn a whole lot about the design of a database that serves as the backend of a simple application and the life cycle of the application’s development. Other stakeholders include Professor Ding and the other students in CSCI6600 that the team will share the project with.

Section 4 Users and Use Cases

The users of this system will include those individuals who will interact with the system as described in the previous section as well as the components within the three layers of the system’s architecture: the frontend (browser), the business layer (server) and the backend (graphical database management system). hosting the application and the system used to manage the database. We summarize these users in an Actor-Use Case Description Table, Table 4.1.

Table 4.1 Actor-Use Case Description Table for Clik

User/Actor	Description
Friend	Wishes to interact with a system through a web-browser on different devices and receive value from the system by posting activities, adding friends to their network and finding events/friends that align with their interest.
Developer	Wishes to design, implement and test the system including a database that holds tables of records that satisfies all relevant requirements of the system.
Dr Ding	Wishes to inspect the collection of deliverables of the project and judge the package based on a set rubric.
Classmates	Wish to be presented with an interesting database concept and application of concepts learned in the course
server	The system will be hosted on a local server using bolt type protocol (bolt://localhost:7687) and neo4j-driver to consume transactions during a react session.
database	Neo4j Desktop will manage and allow for queries on the

	collection of nodes, friends and events,and edges (relationships connecting a record in the friends table with a record in the events table) as a graphical database.
browser	The browser will render chosen views of the model and relay communications between a Friend and the server in order for the Friend to complete tasks.

The use cases for the actors are presented in Table 4.2, and the team used these use cases to explore the requirements for the system in Section 6.

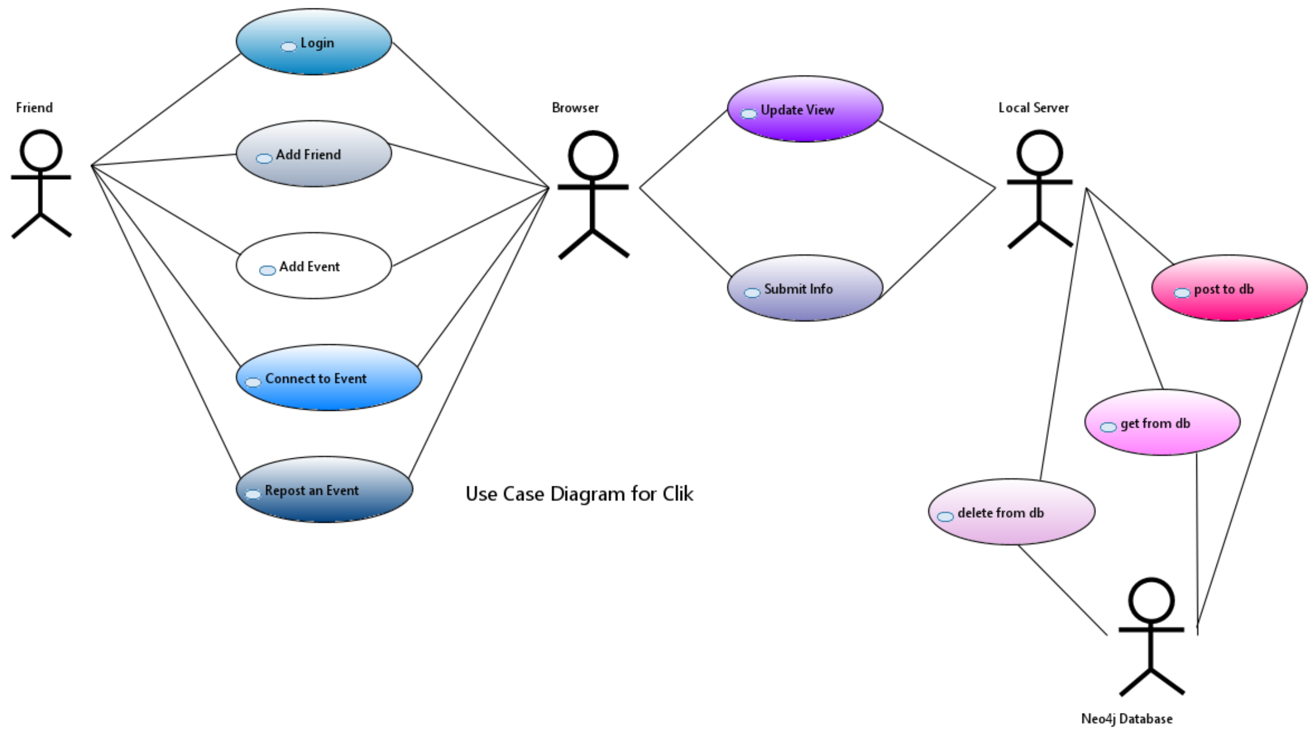
Table 4.2 Use Cases

Use Case	Description
1. Login	A Friend wants to log into the system.
2. Add Friend	A Friend wants to add another Friend user to their network of Friends.
3. Add Event	A Friend wants to add an Event to their network of Events.
4. Connect to an Event	A Friend wants to demonstrate interest in an Event posted by another Friend.
5. Repost an Event	A Friend wants to repost an Event that will increase the network of that Event.
6. Update View	The server will communicate when it is appropriate to update the view based on communication from the database or communication from the Friend.
7. Submit Information	An action has been completed by a Friend through the Browser, information will be communicated to the Server.
8. Post to Database	The Server requests that a new node Friend Event or a new relationship interestIn or rePost be added to the database.
9. Get from Database	The Server will query and manage the database.
10. Delete from Database	The Server will ask for a record to be deleted from the database.

Section 5 Use Case Diagram

After identifying the Use Cases and the Actors for Clik, the team put together a Use Case Diagram Figure 5.1 to illustrate the associations between these two identities using the Papyrus extension within the Eclipse IDE.

Figurer 5.1 Use Case Diagram



Section 6 Requirements Analysis

The functional requirements, to date, as identified by the development team are listed in the following two sections. In List 6.1 the functional requirements for the system Clik where the focus is on what the system should be able to do. The requirement on the measurable performance features of the system is a relatively short list in that the development team and stakeholders in general have not hashed out the details of necessarily metrics in performance.

List 6.1 Functional Requirements to date:

1. The system should allow a Friend to sign up and log in to the system with a username and password
2. The system should keep the username and password information of all Friends in a secure database

3. The system should allow a Friend to view their Clik network of Friends and Events with connection between Friends and Events.
4. The system should allow a Friend to add a Friend to their Clik network.
5. The system should allow a Friend to delete a Friend from their Clik network.
6. The system should allow a Friend to add an Event to their Clik network.
7. The system should allow a Friend to delete an Event from their Clik network.
8. The system should allow a Friend to demonstrate interest in an Event in their Clik network.
9. The system should allow a Friend to rePost an Event of a Friend in their Clik network so that Friends not in a shared network can demonstrate interest in the Event.
10. The system should maintain a database with a node for each Friend, each Event and relationships between Friends and Events as a common edge between a Friend and Event.
11. The system should be documented with all code available, a requirements.txt file and README.md file that assists Dr. Ding and classmates to follow the production of the system.
12. The system should be able to run within Microsoft Edge, Google Chrome or FireFox browsers.
13. The system should be able to run on a Window, Mac or Linux operating system.

List 6.2 Non-Functional Requirements to date:

1. The system should take no more than 1 minute to reload with a new graph of Friends and Events.

Section 7 The Creation of the Database

The Database is the primary focus of this project. The team chose to first explore the attributes and tables that would be needed for Clik using a familiar relational database model approach and also to investigate other types of database designs such as a graphical database system. For this graphical interface we used a Microsoft SQL Server database instance with SQL Server Management Studio GUI tool to render the entity relationship diagram.

Section 7.1 A Relational Database Approach

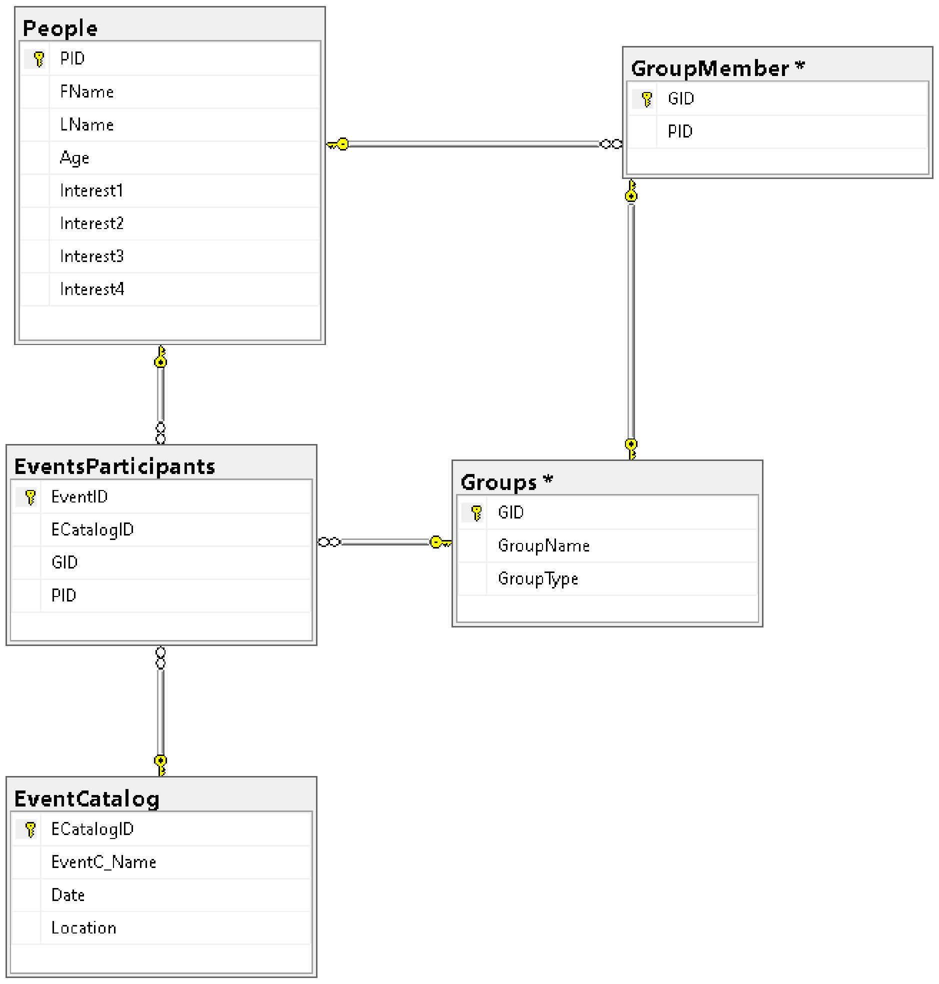
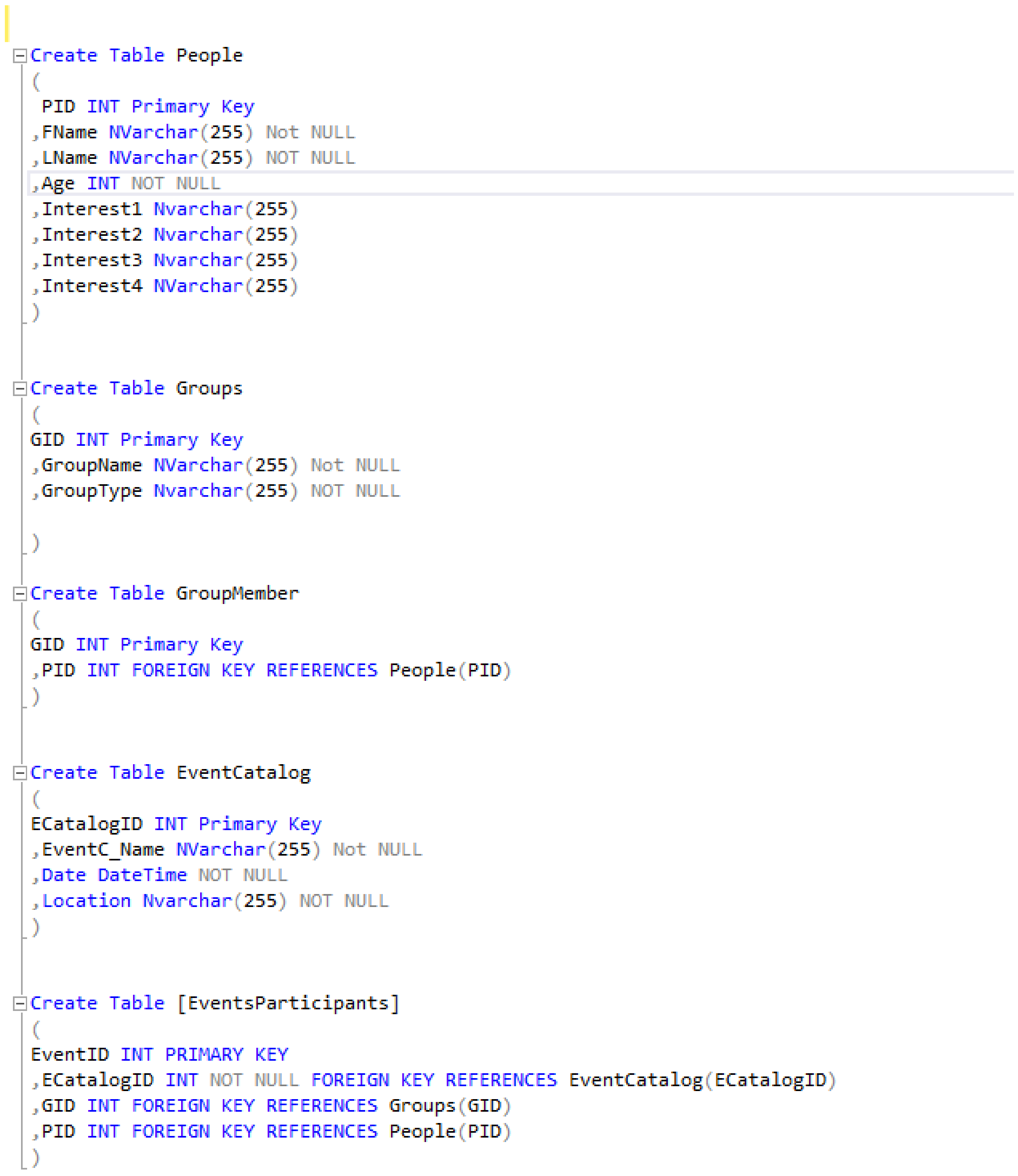


Figure 7.1 The Beginnings of a Relational Database for Clik

Figure 7.2 the DDL used to create a Relational Database for Clik



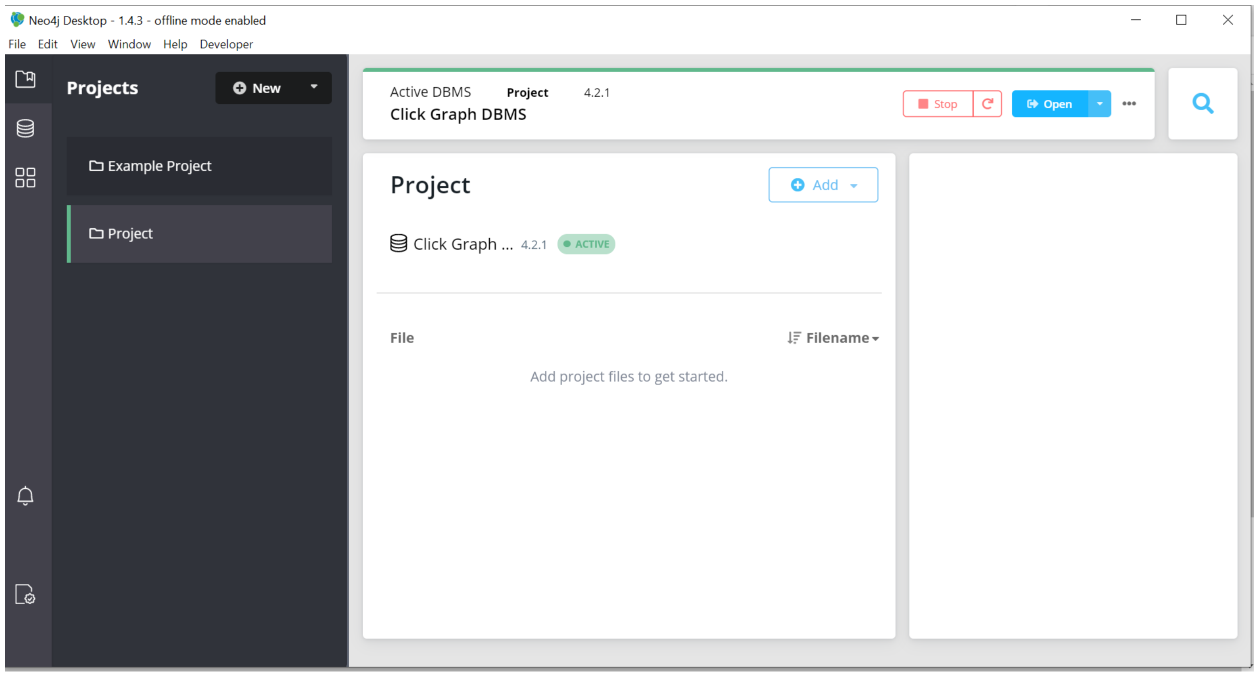
```
Create Table People
(
 PID INT Primary Key
,FName NVarchar(255) Not NULL
,LName NVarchar(255) NOT NULL
,Age INT NOT NULL
,Interest1 Nvarchar(255)
,Interest2 Nvarchar(255)
,Interest3 Nvarchar(255)
,Interest4 NVarchar(255)
)


Create Table Groups
(
GID INT Primary Key
,GroupName NVarchar(255) Not NULL
,GroupType Nvarchar(255) NOT NULL

)

Create Table GroupMember
(
GID INT Primary Key
,PID INT FOREIGN KEY REFERENCES People(PID)
)


Create Table EventCatalog
(
ECatalogID INT Primary Key
,EventC_Name NVarchar(255) Not NULL
,Date DateTime NOT NULL
,Location Nvarchar(255) NOT NULL
)


Create Table [EventsParticipants]
(
EventID INT PRIMARY KEY
,ECatalogID INT NOT NULL FOREIGN KEY REFERENCES EventCatalog(ECatalogID)
,GID INT FOREIGN KEY REFERENCES Groups(GID)
,PID INT FOREIGN KEY REFERENCES People(PID)
)
```

Section 7.2 A Graphical Database Approach

The team decided to explore the possibilities of other types of databases for this application, in particular the team created a graph database for Clik. The concept of allowing each Friend and each Event to be represented as a node within a graph, the relationships between the two records to be embedded and stored within these nodes and to capitalize on the rendering of the database as a graph in the browser was of great interest to the team. Using a graph database allows some queries on the database to run very quickly and there are graphing algorithms built into most graphical database management systems that can provide future features to the application. One such utilization of a path analysis algorithm within a graph allows “influencers” to be identified. In the context of Clik, an influencer could be defined to be a Friend who is associated with a high number of reposted events.

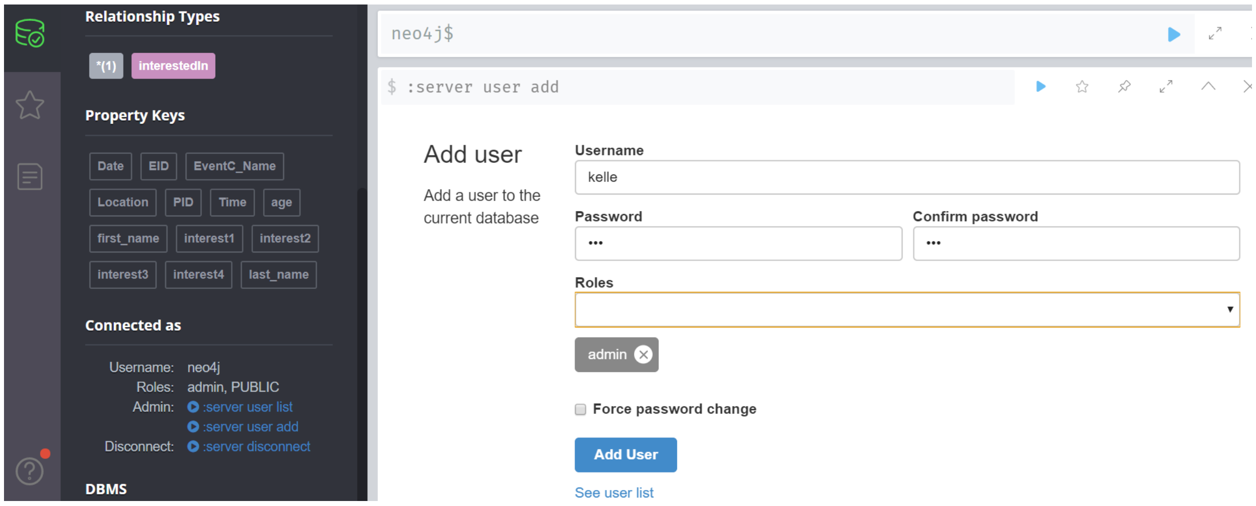
To create the graphical database, the team used the desktop community version of Neo4j available to https://neo4j.com. Within neo4j, a new project “ClickGraph’ was created and then opened with the blue button on the top right of the GUI (Fig 7.3)

Figure 7.3 Neo4j Desktop Software


Neo4j Desktop - 1.4.3 - offline mode enabled
File Edit View Window Help Developer
Projects
New
Example Project
Project
Active DBMS
Project
4.2.1
Click Graph DBMS
Stop
Open
Project
Add
Click Graph ... 4.2.1 ACTIVE
File
Filename
Add project files to get started.

Once the database is active and opened, a user with their role and password can be set. In Figure 7.4, the user is kelle is added with the role of an administrator and the password is set to 1111. These are important parameters used in the configuration of the driver to connect the local server and the application.

Figure 7.4 The Neo4j Browser User Interface



Relationship Types

*(1) interestedIn

Property Keys

Date EID EventC_Name Location PID Time age first_name interest1 interest2 interest3 interest4 last_name

Connected as

Username: neo4j
Roles: admin, PUBLIC
Admin: :server user list
:server user add
Disconnect: :server disconnect

DBMS

```
neo4j$
$ :server user add
```

Add user
Add a user to the current database

Username: kelle
Password: •••
Confirm password: •••
Roles: admin

Force password change

Add User
See user list

Next, the database is built one node at a time using the Cypher DDL/DML. To create a Person record from the People Table of the original relational database described above, one uses the sequence of key words and strings or integers followed by the "play" button in the desktop or browser edition of neo4j.

```
create (p:Person {PID:0003,  first_name: "greg", last_name:"sutton", age: "null", interest1: "gardening", interest2: 'travel', interest3:'pets', interest4:'baking'})
```

Here there is no way to set the constraint that the PID is unique, but we will allow a user to enter in the other attribute values and create the node with care to use distinct PID so that PID can be the Primary Key for these nodes. However, when you create any node or edge in neo4j, the system assigns that element a unique identifier...an id. The <id> can be used as a way to uniquely access a node, but it will not prevent multiple records from being created with the same value of all attributes. I accidentally created another Person node with no attribute values, and the neo4j system created a node with <id> = 4 (the fourth node created). To remove this node, it must not have any edges, and so I needed to use the command below to remove the entry from the database.

```
MATCH (n:Person) where id(n)=4
DETACH DELETE n
```

To view a node, you need to find the node in the graph database and return the node. The code to see the Person node with PID 003, use:

```
match (n:Person) where n.PID=003 return n
```

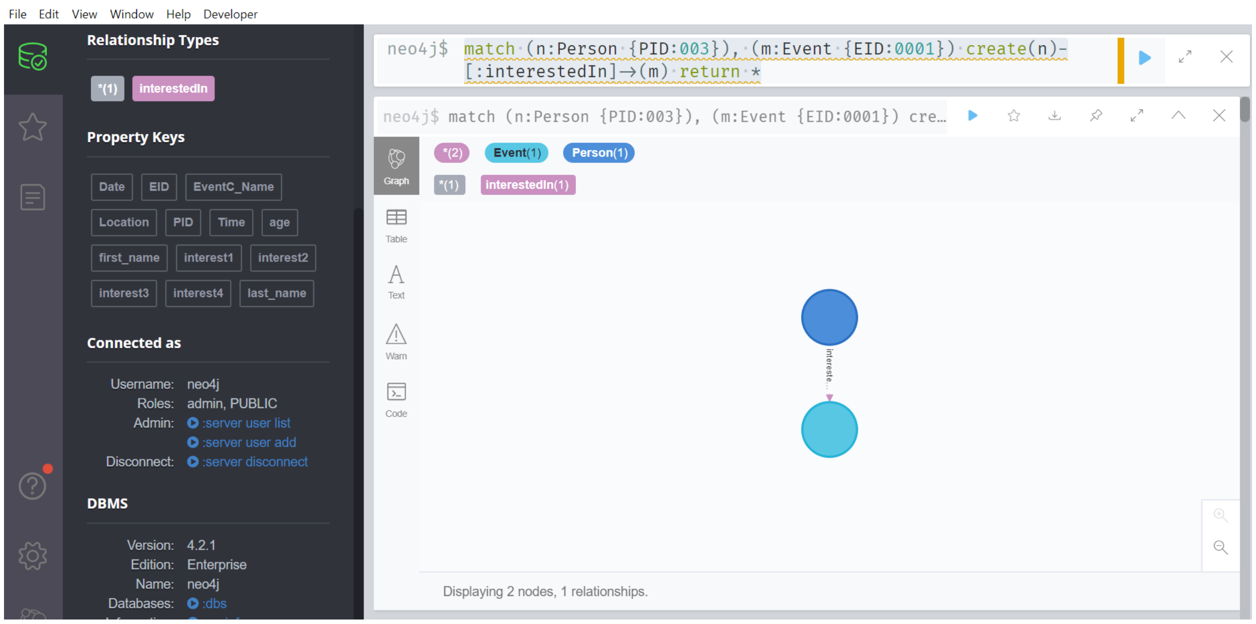
To generate a beginning database for testing the application, the team created a number of Persons and Events. An example cypher line to create the Event with PID 0001 is:

```
create (e:Event {EID:0001,EventC_Name:"",Date:'04/01/2021',Location:'OMally_Bar',Time:'2100'})
```

And to establish an edge demonstrating that Person PID 003 is interestedIn the Event EID 0001, use the cypher command below. The results can be viewed as a graph, a table or as a list of json objects as shown in Figure 7.5

```
match (n:Person {PID:003}), (m:Event {EID:0001}) create(n)-[:interestedIn]->(m) return *
```

Figure 7.5 The views of the database



Relationship Types

*(1) interestedIn

Property Keys

Date EID EventC_Name Location PID Time age first_name interest1 interest2 interest3 interest4 last_name

Connected as

Username:	neo4j
Roles:	admin, PUBLIC
Admin:	:server user list
	:server user add
Disconnect:	:server disconnect

DBMS

Version:	4.2.1
Edition:	Enterprise
Name:	neo4j
Databases:	:dbs

```
neo4j$ match (n:Person {PID:003}), (m:Event {EID:0001}) create(n)-
[:interestedIn]→(m) return *
```

```
neo4j$ match (n:Person {PID:003}), (m:Event {EID:0001}) cre…
```

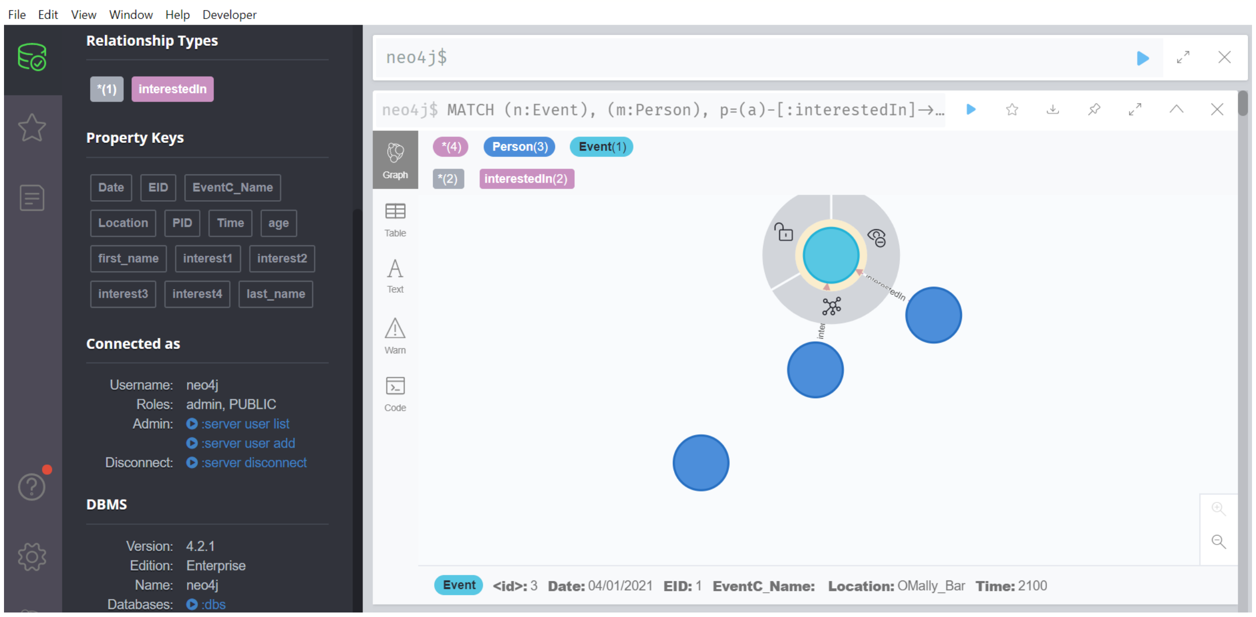
interestedIn(1)

Displaying 2 nodes, 1 relationships.

The entire collection of Events and Persons is the Clik network that would be seen by kelle if this person was the logged in user interacting with Clik. To gain access the three different views of the data (as a table, as code and as a graph) use the command below and the network illustrates the friend Kelle and Greg are both interested in attending the event on 4/01/2021 at 2100 hours at O'Mally Bar (note the attributes in Figure 7.6 below).

```
match (n:Person), (m:Event), p=(a)-[:interestedIn]->(b) return *
```

Figure 7.6 The attributes and Neo4j system defined id for each node is unique for every node.



Relationship Types

*(1) interestedIn

Property Keys

Date EID EventC_Name Location PID Time age first_name interest1 interest2 interest3 interest4 last_name

Connected as

Username: neo4j
Roles: admin, PUBLIC
Admin: :server user list
:server user add
Disconnect: :server disconnect

DBMS

Version: 4.2.1
Edition: Enterprise
Name: neo4j
Databases: :dbs

```
neo4j$
neo4j$ MATCH (n:Event), (m:Person), p=(a)-[:interestedIn]->…
```

*(4) Person(3) Event(1)
*(2) interestedIn(2)

Event <id>: 3 Date: 04/01/2021 EID: 1 EventC_Name: Location: OMally_Bar Time: 2100

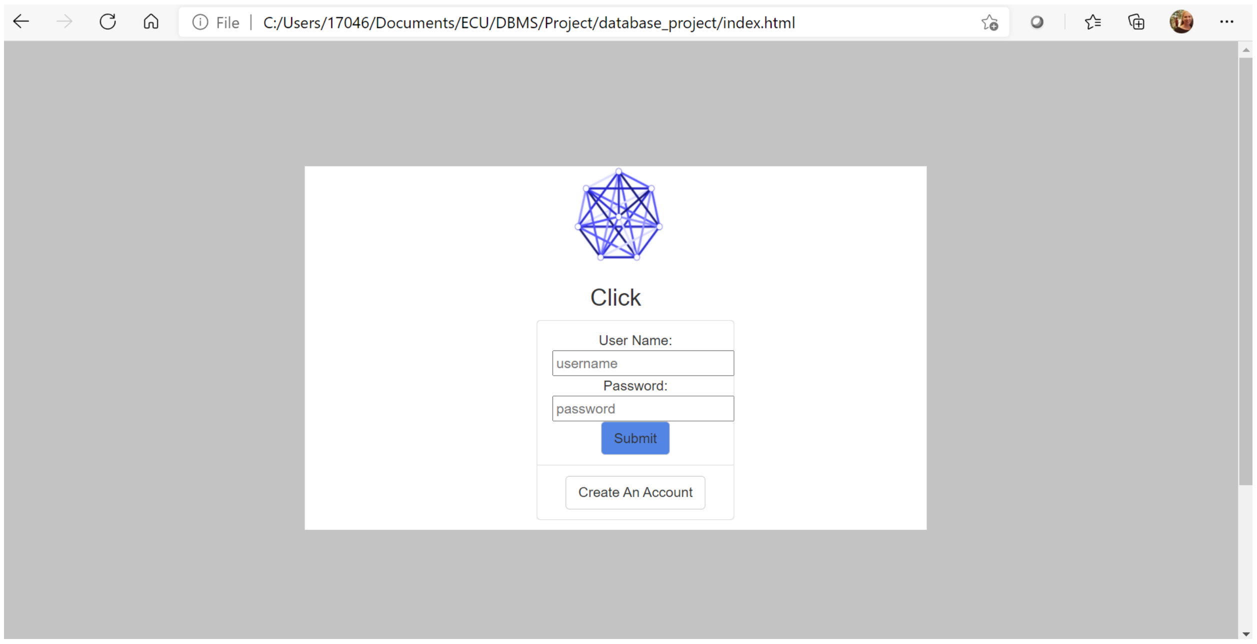
Section 8 The User Interface and Happy Flow

The team is developing the User Interface (UI) incrementally. Currently, the team is using the neovis driver to render the graph. Another option for the system would be to use D3js and the simple neo4j driver for javascript which would potentially make options for color choice and the labeling of the nodes available to the user.

The “happy flow” for a user of the system will consist of:

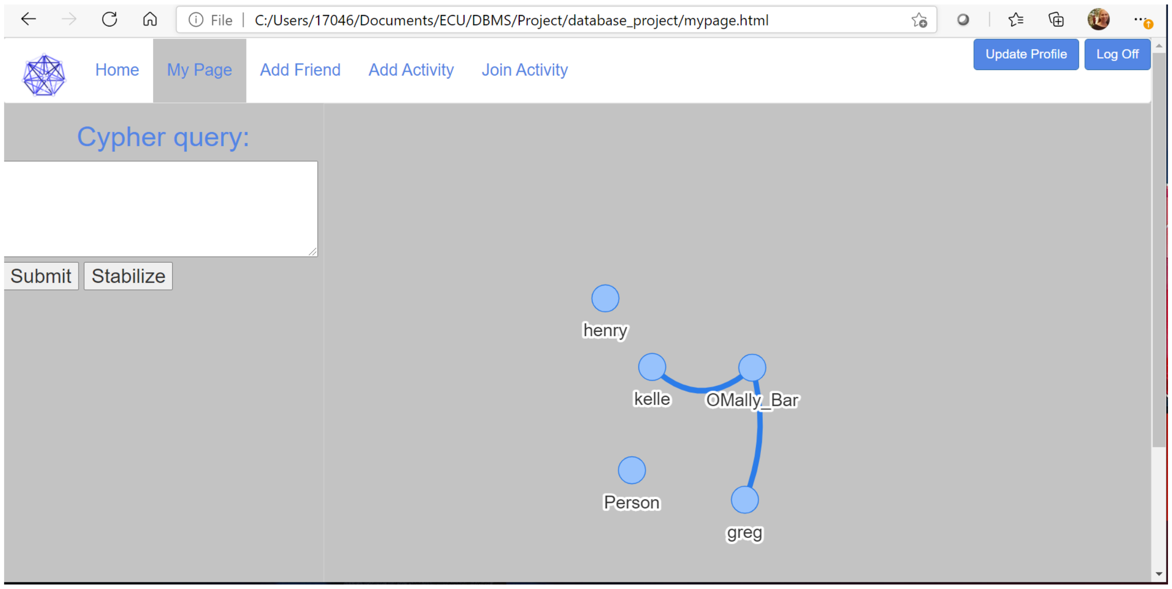
1. Person will sign up with a username and password that is unique in the database (the system will check this). A node will be added to the database with the username and password.

Figure 8.1 The home page of Clik (Click)


File | C:/Users/17046/Documents/ECU/DBMS/Project/database_project/index.html
Click
User Name:
username
Password:
password
Submit
Create An Account

2. Person will sign in with their credentials. When authenticated, the system will update the user view to mypage.html that displays only the friend and events in the user's node. The current mypage.html (Figure 8.2) shows that the team has gained control over color scheme including the KNOWDS and Events.
3. The Person will add a Friend, add an event, update their profile.

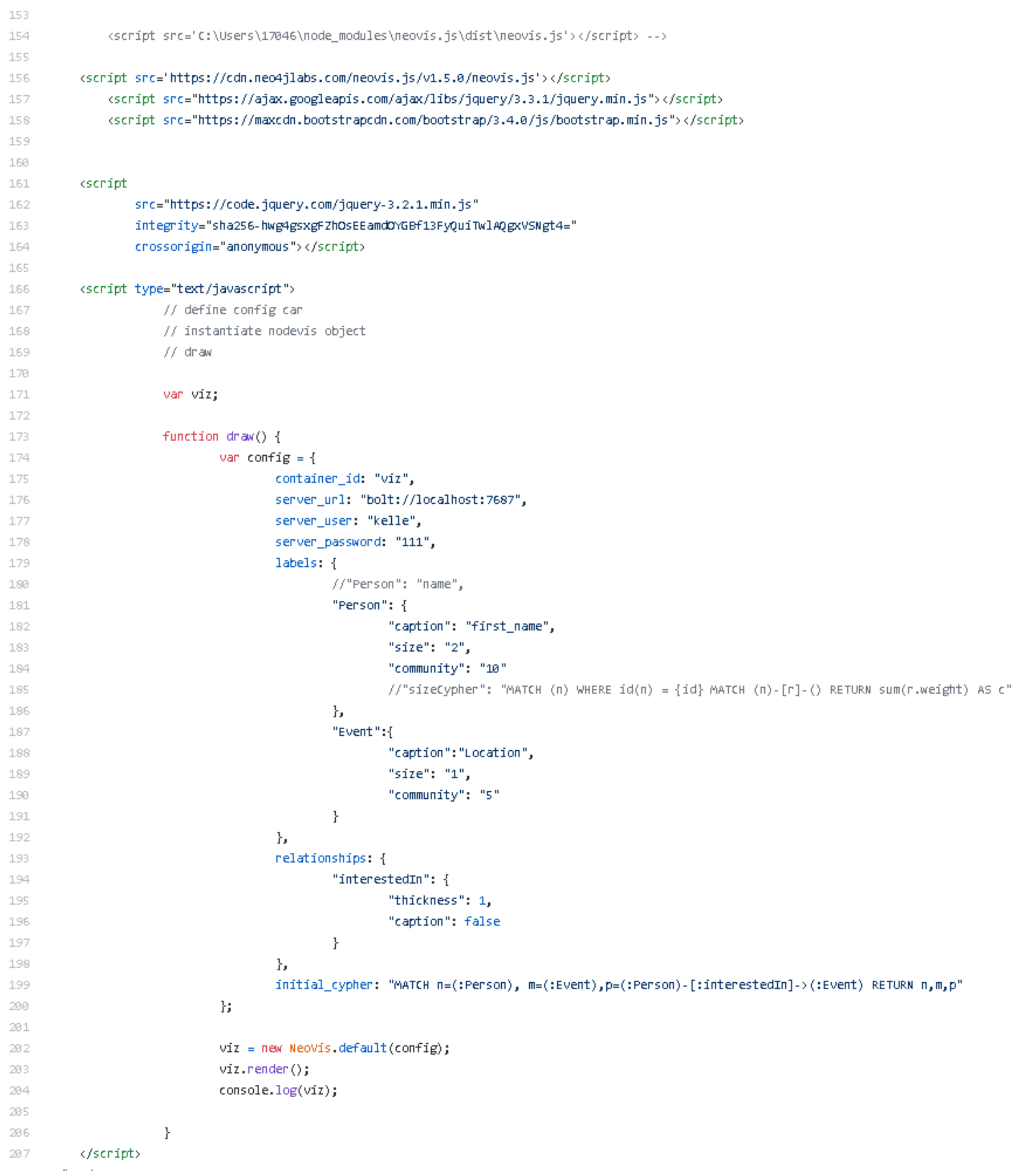
Figure 8.2 The Person's view of their KNOWD Network



Section 9 The Database

To connect the Clik website to the Neo4j Clik database, the team utilized the open source javascript code libraries Neovis.js (https://github.com/neo4j-contrib/neovis.js/) and npm (https://docs.npmjs.com/) for their ability to connect to Neo4j instances, create visualization features for the website and to compile our code. The team also imported ajax and bootstrap libraries for current and future feature development. The mypage.html code is included in Figure 9.1 to demonstrate the configuration embedded.

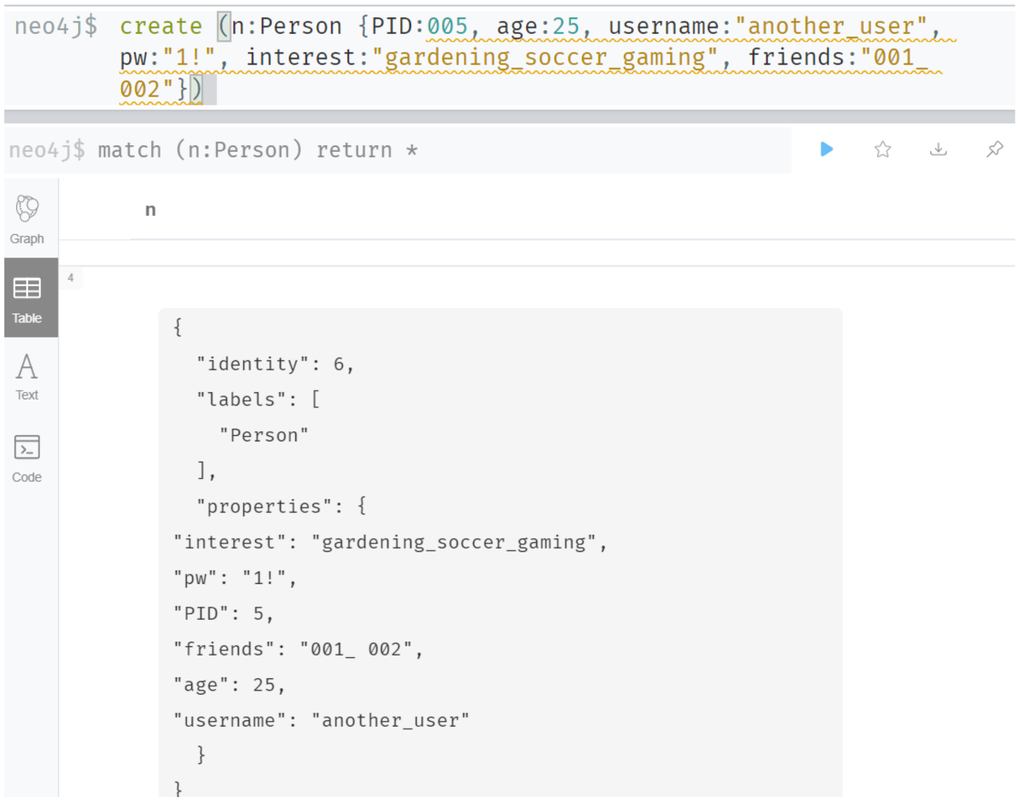
9.1 The html document for the “mypage.html” rendered by the browser when a user is logged in.



```
    <script src='C:\Users\17046\node_modules\neovis.js\dist\neovis.js'></script> -->

<script src='https://cdn.neo4jlabs.com/neovis.js/v1.5.0/neovis.js'></script>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/js/bootstrap.min.js"></script>


<script
        src="https://code.jquery.com/jquery-3.2.1.min.js"
        integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
        crossorigin="anonymous"></script>

<script type="text/javascript">
            // define config car
            // instantiate nodevis object
            // draw

            var viz;

            function draw() {
                    var config = {
                            container_id: "viz",
                            server_url: "bolt://localhost:7687",
                            server_user: "kelle",
                            server_password: "111",
                            labels: {
                                    //"Person": "name",
                                    "Person": {
                                            "caption": "first_name",
                                            "size": "2",
                                            "community": "10"
                                            //"sizeCypher": "MATCH (n) WHERE id(n) = {id} MATCH (n)-[r]-() RETURN sum(r.weight) AS c"
                                    },
                                    "Event":{
                                            "caption":"Location",
                                            "size": "1",
                                            "community": "5"
                                    }
                            },
                            relationships: {
                                    "interestedIn": {
                                            "thickness": 1,
                                            "caption": false
                                    }
                            },
                            initial_cypher: "MATCH n=(:Person), m=(:Event),p=(:Person)-[:interestedIn]->(:Event) RETURN n,m,p"
                    };

                    viz = new NeoVis.default(config);
                    viz.render();
                    console.log(viz);

            }
</script>
```

An update of the database node was required to accommodate a design change allowing for a Person to SignUp, SignIn and only see Persons in their KNOWD network in the view. The new database for a Person is shown in Figure 9.2.

Figure 9.2 Person



```
neo4j$ create (n:Person {PID:005, age:25, username:"another_user",
       pw:"1!", interest:"gardening_soccer_gaming", friends:"001_
       002"})
```

```
neo4j$ match (n:Person) return *
```

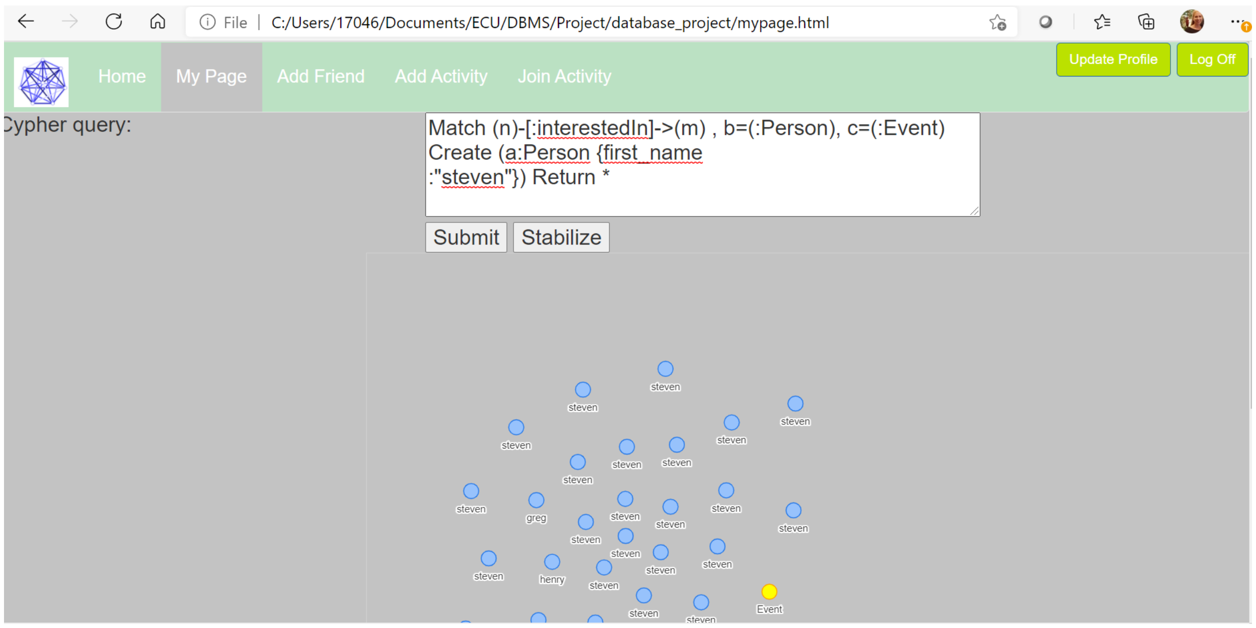
```
{
  "identity": 6,
  "labels": [
    "Person"
  ],
  "properties": {
"interest": "gardening_soccer_gaming",
"pw": "1!",
"PID": 5,
"friends": "001_ 002",
"age": 25,
"username": "another_user"
  }
}
```

The change of attributes to allow a user (Person) to sign in will use a username instead of first_name and last_name and creates a password string attribute. The Persons' in the user's network will be indicated by a string of PID (Person Identification) where each unique PID is separated by an underscore delimiter allowing the system to parse the value of this attribute, match Persons in the database on this attribute and to return all KNOWDS (Friends of the user) and Events posted by these KNOWDS.

Other information about the user's age and interest can be updated when the user selects the Update Profile feature of the application. When the user signs in, the application will be able to authenticate the credentials using the database using the unique username.

The system currently implements many features through a Cypher Query textbox. This input allows the user to enter in cypher code to add a Friend, add an Event, or Search for Persons with similar interest. This textbox will be updated into “stored procedures” that accept only the value from a set form and enable the user to interact with the database without having to know cypher. Note, this is also how the team discovered the need for a delete option...way too many Friends named Steven AND the need for distinct record control through this (earlier) UI.

Figure:



```
Match (n)-[:interestedIn]->(m) , b=(:Person), c=(:Event)
Create (a:Person {first_name
:"steven"}) Return *
```

Section 10 The Continued Plan

While our presentation is quickly approaching, the team feels confident that they will be able to provide the following features within the application before they present.

1. Implement constraints on a Person username and password to that they are unique.
2. Allow a user to add a friend, add an event without using cypher code, rather use a form or select an entry from a table.
3. Develop a test database
4. Make some progress to using available graph algorithms to demonstrate influencer analysis in the database.
5. Continue to improve the quality of the UI and design for appeal and ease of use.

Section 11 Clik's Implementation

Section 11.1 The Requirements

The Package for Click can be found at [KelleClark/database_project (github.com)](https://github.com/KelleClark/database_project) . One way to run the system locally on your computer:

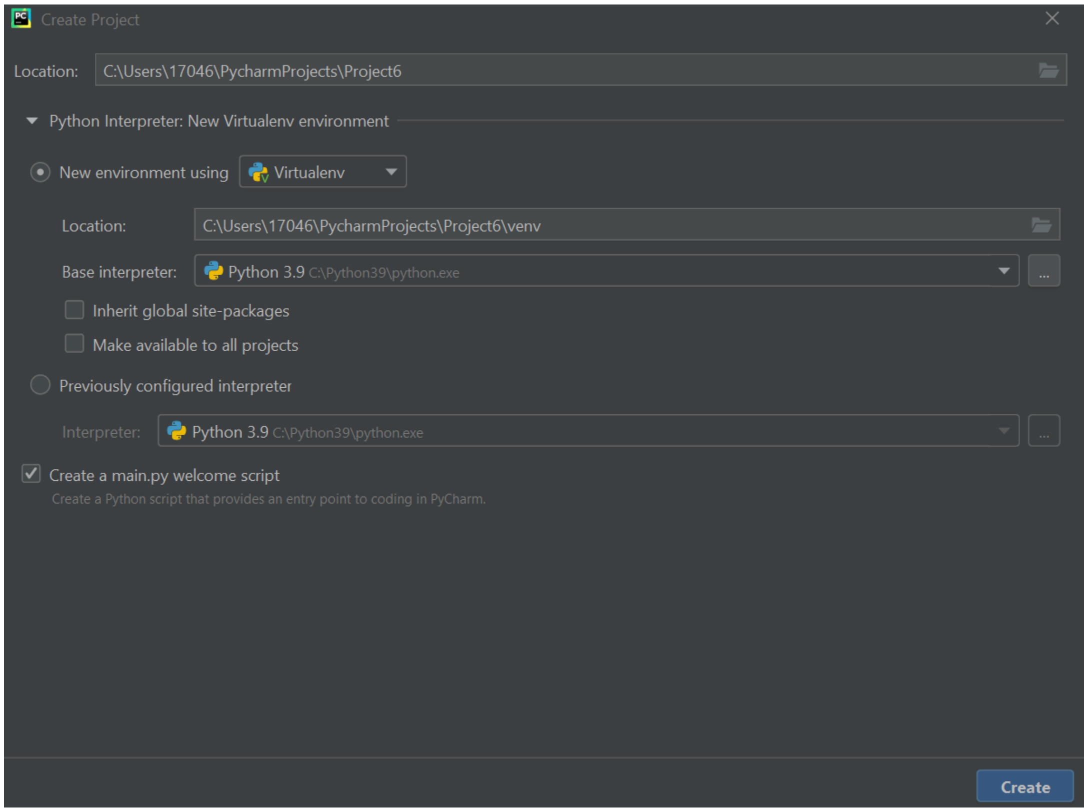
1. You will need to have the following on your computer:

Python
PyCharm 2019 or later
Neo4j Desktop v.4 or later
Git

Note where you store Python and Git within your File System, you will need to set the location of the .exe files when setting up your project.

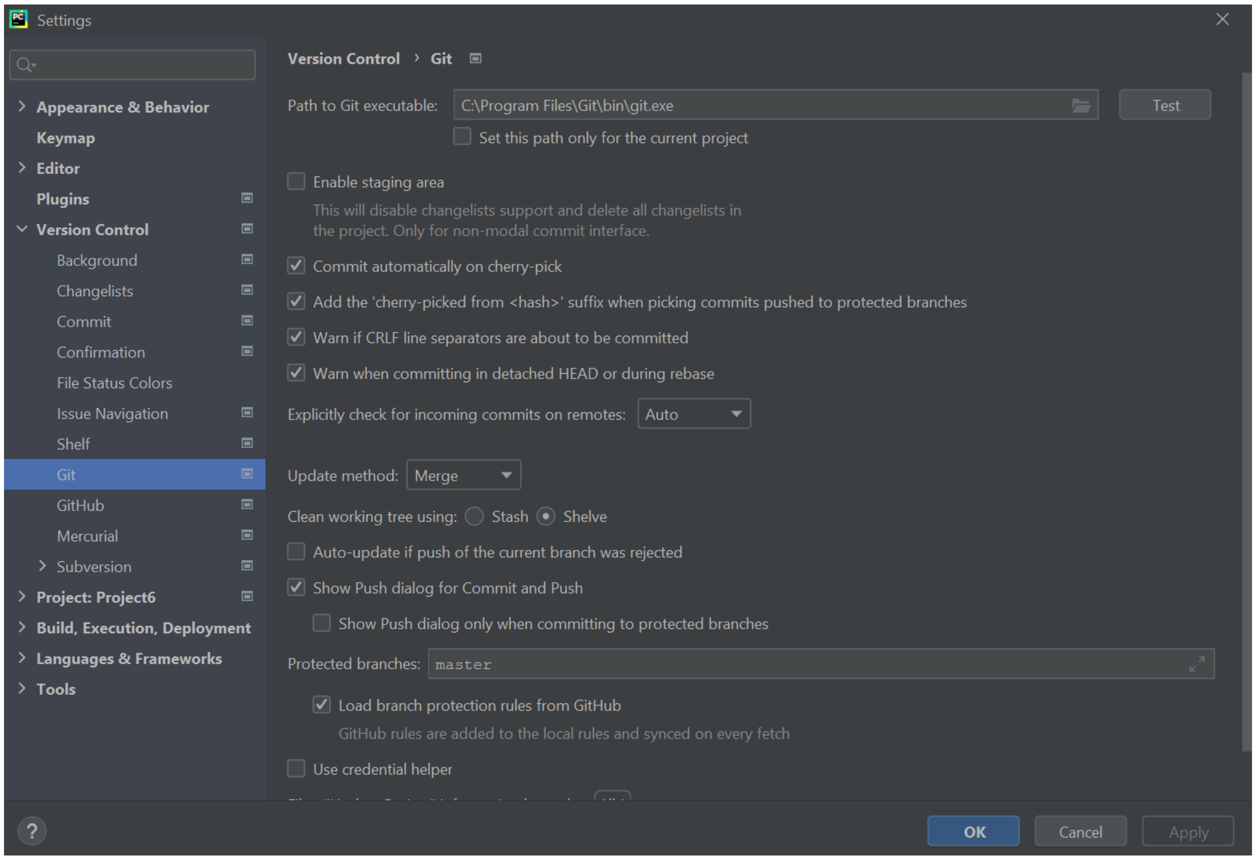
2. Open PyCharm and select NewProject:
   choose a location for your new project on your computer
   set the location of your Python Interpreter: New Virtual Environment to your Python.exe
   create

Figure : set up new project


Create Project
Location: C:\Users\17046\PycharmProjects\Project6
Python Interpreter: New Virtualenv environment
(●) New environment using: Virtualenv
Location: C:\Users\17046\PycharmProjects\Project6\venv
Base interpreter: Python 3.9 C:\Python39\python.exe
[ ] Inherit global site-packages
[ ] Make available to all projects
( ) Previously configured interpreter
Interpreter: Python 3.9 C:\Python39\python.exe
[✓] Create a main.py welcome script
Create a Python script that provides an entry point to coding in PyCharm.

3. Under Files/Settings/Version Control select Git and fill in the location on your computer of your Git.exe file

Figure: set up the Git for the project



Version Control › Git

Path to Git executable: C:\Program Files\Git\bin\git.exe

[ ] Set this path only for the current project

[ ] Enable staging area
This will disable changelists support and delete all changelists in the project. Only for non-modal commit interface.

[✓] Commit automatically on cherry-pick

[✓] Add the 'cherry-picked from <hash>' suffix when picking commits pushed to protected branches

[✓] Warn if CRLF line separators are about to be committed

[✓] Warn when committing in detached HEAD or during rebase

Explicitly check for incoming commits on remotes: Auto

Update method: Merge

Clean working tree using: Stash [✓ Shelve]

[ ] Auto-update if push of the current branch was rejected

[✓] Show Push dialog for Commit and Push

[ ] Show Push dialog only when committing to protected branches

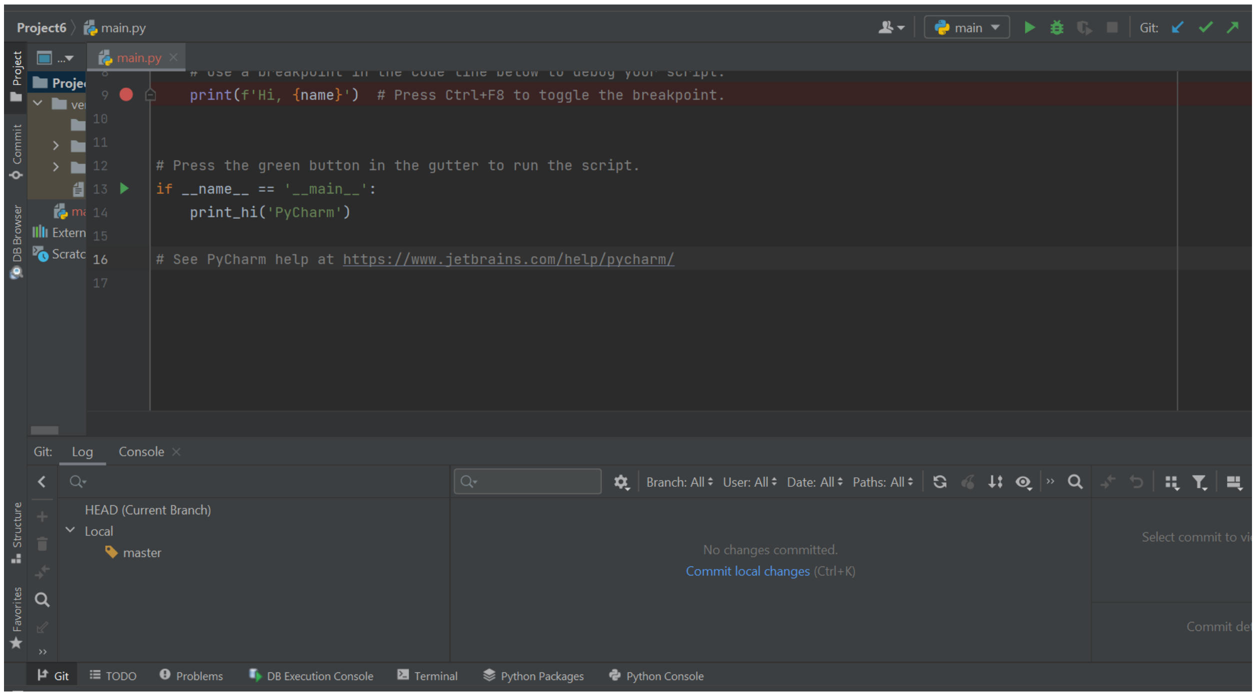
Protected branches: master

[✓] Load branch protection rules from GitHub
GitHub rules are added to the local rules and synced on every fetch

[ ] Use credential helper

4. In the top navigation bar, select the Version Control System VCS and choose Git. From the Github address above, grab the url to clone the repository and paste in the url of the git repository. It is a good idea to use the Test button to see if Git is connected.
You should now be able to use the Git features like commit (green check) , commit/push (green up arrow) and pull (blue arrow) on the top right and to checkout and create branches on the bottom left.

Figure The Git features in PyCharm

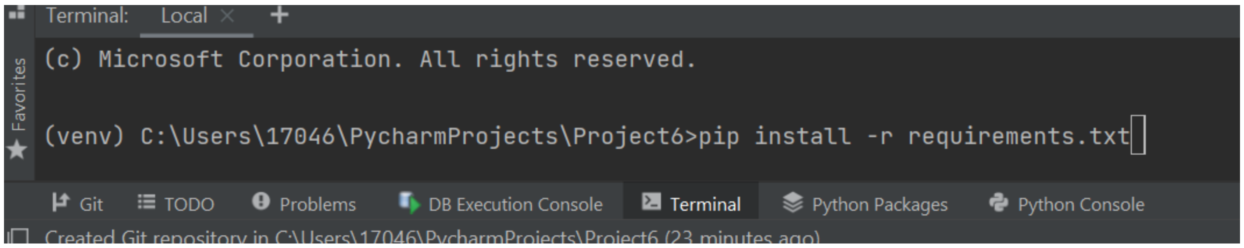


```
    print(f'Hi, {name}')  # Press Ctrl+F8 to toggle the breakpoint.


# Press the green button in the gutter to run the script.
if __name__ == '__main__':
    print_hi('PyCharm')

# See PyCharm help at https://www.jetbrains.com/help/pycharm/
```

```
HEAD (Current Branch)
Local
    master

No changes committed.
Commit local changes (Ctrl+K)
```

5. Now we need to make sure our python virtual environment that you set up in step 2 has all of the required packages and libraries. Go to the python terminal on the bottom of the interface and use the cli $ (venv) somelocation > pip install -r requirements.txt

Figure To install required packages and libraries



```
(c) Microsoft Corporation. All rights reserved.

(venv) C:\Users\17046\PycharmProjects\Project6>pip install -r requirements.txt
```

```
Created Git repository in C:\Users\17046\PycharmProjects\Project6 (23 minutes ago)
```

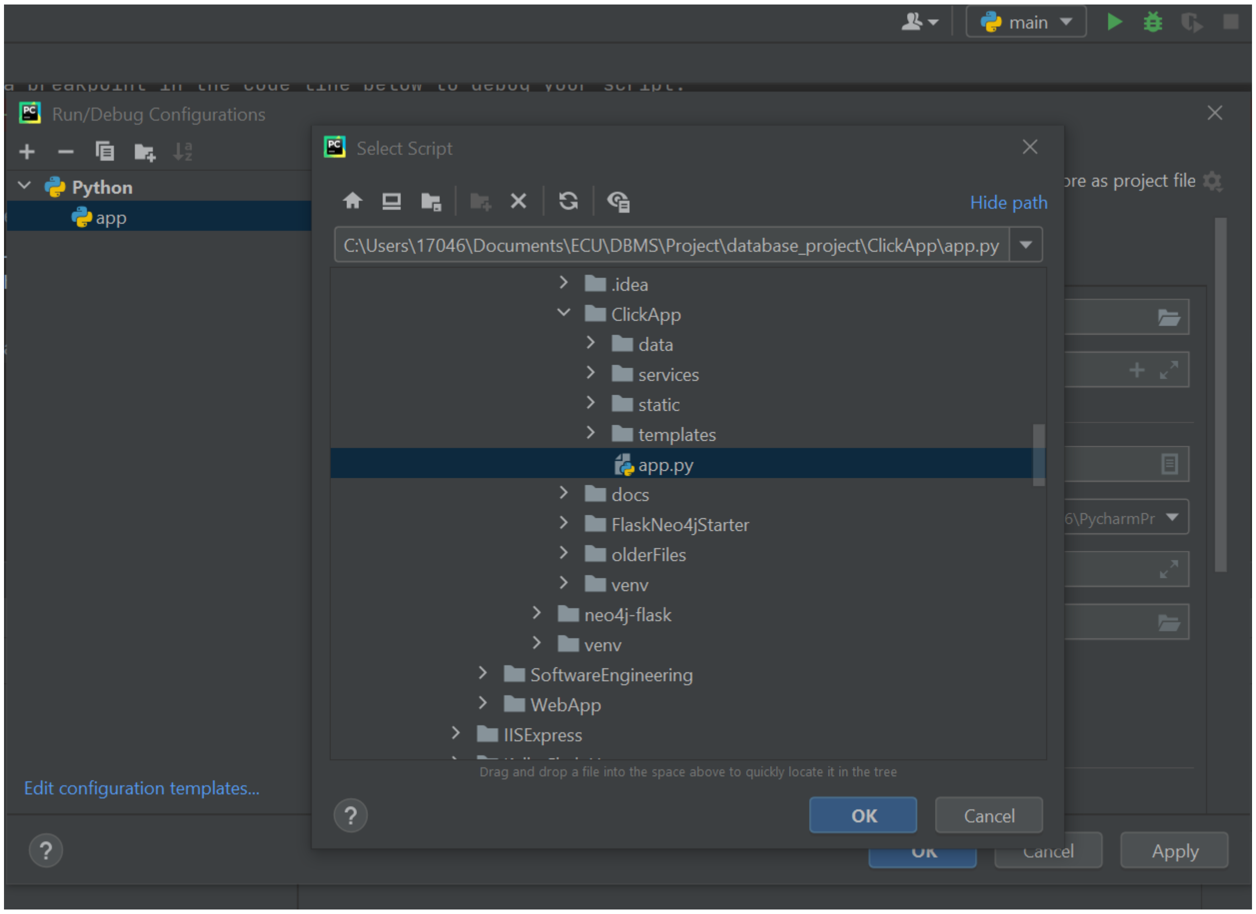
Now minimize PyCharm for a second and open Neo4j Desktop.

1. Go to Project and create a New Project...and then add a Database. Start the Database and Open the Neo4j Browser. Connect to the database by using the cli cypher statement on the command line at the top $:server connect

2. Open the DB menu from the left and add find :user server add. Add yourself as a user ...for me username is kelle and password is 111. We will need that in our database set up.

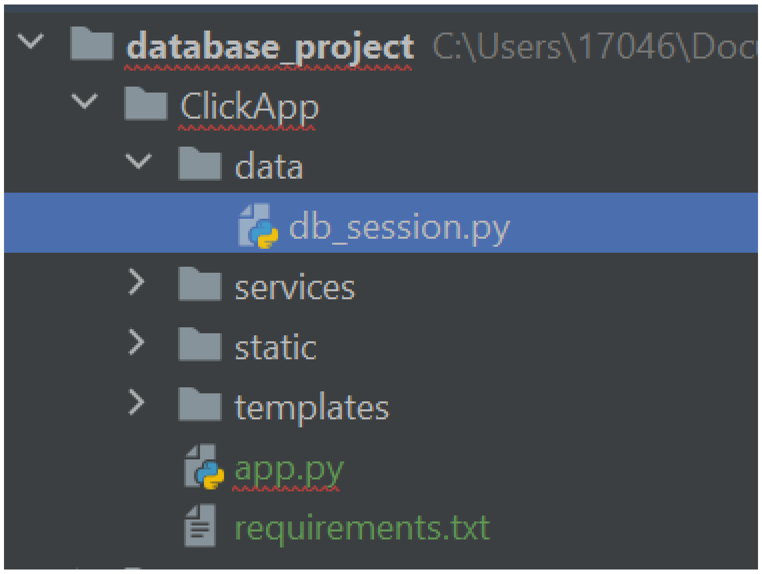
Minimize Neo4j Desktop and Neo4j Browser. Open PyCharm back up and set the Configuration near the top in the navigation bar. Then name the configuration...I named mine app and set the location to the file you want to run the application...app.py. Make sure you have the path of the Python venv set to the one for your project… and apply.

Figure Set up the PyCharm configuration to run the ClickApp



Before we run the system, find the file for our configuration of the graph database…

Figure: Find the configuration file for Graph neo4j object



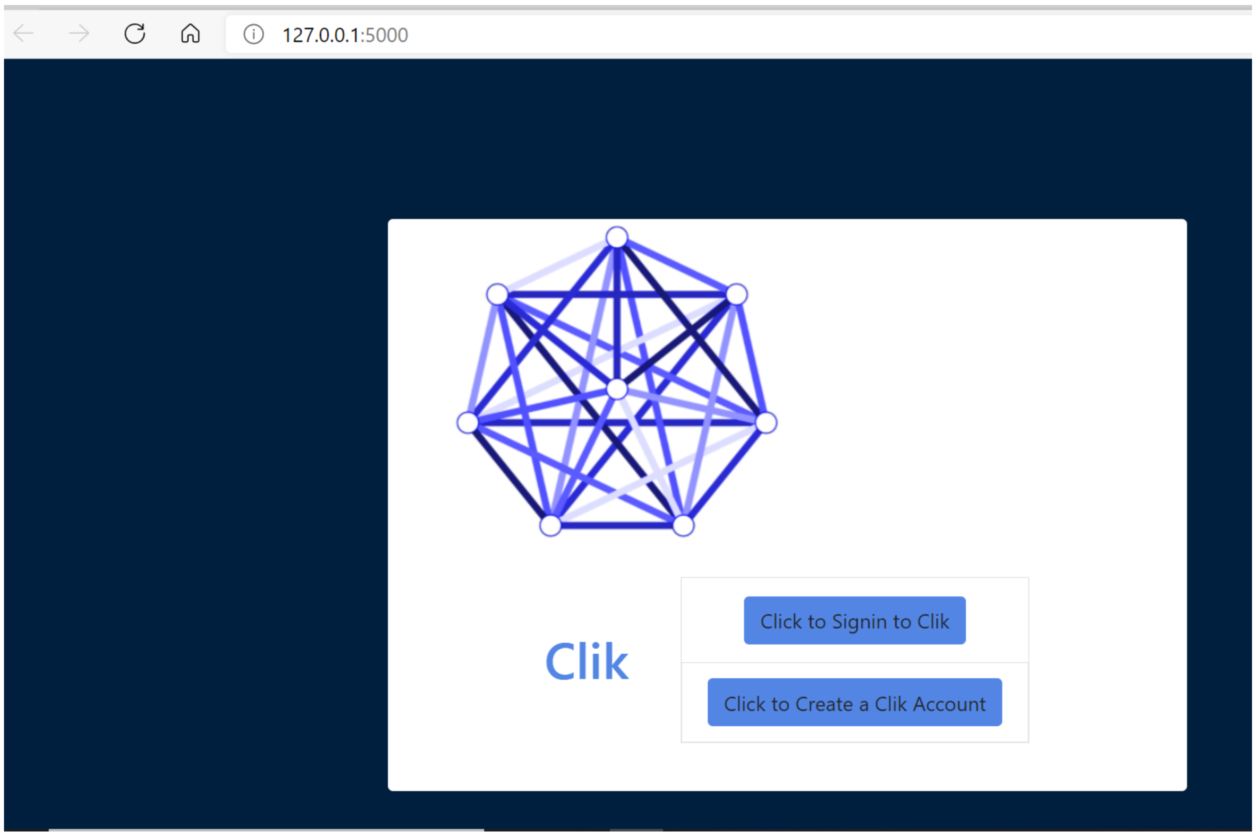
Update your username and password that you just added to the Neo4j Database within your new project.

Figure : Make sure to change user and password to your information from your Neo4j Project Database

```
from py2neo import Graph


def db_auth():
    user = 'kelle'
    pword = '111'
    #graph = Graph("bolt://127.0.0.1:7687/db/data/", username=user,
password=pword)
    graph = Graph("http://127.0.0.1:7474/db/data/", username=user,
password=pword)
    return graph
```

You should now be able to press the green “play” button and run the system or use the run button on the bottom. Errors will be shown in the run terminal at the bottom, but you should be able to open your web browser (Edge is good) and open 127.0.0.1:5000

Figure The rendering of home/index.html rendered by @app.route("/") within the app.py file using flask



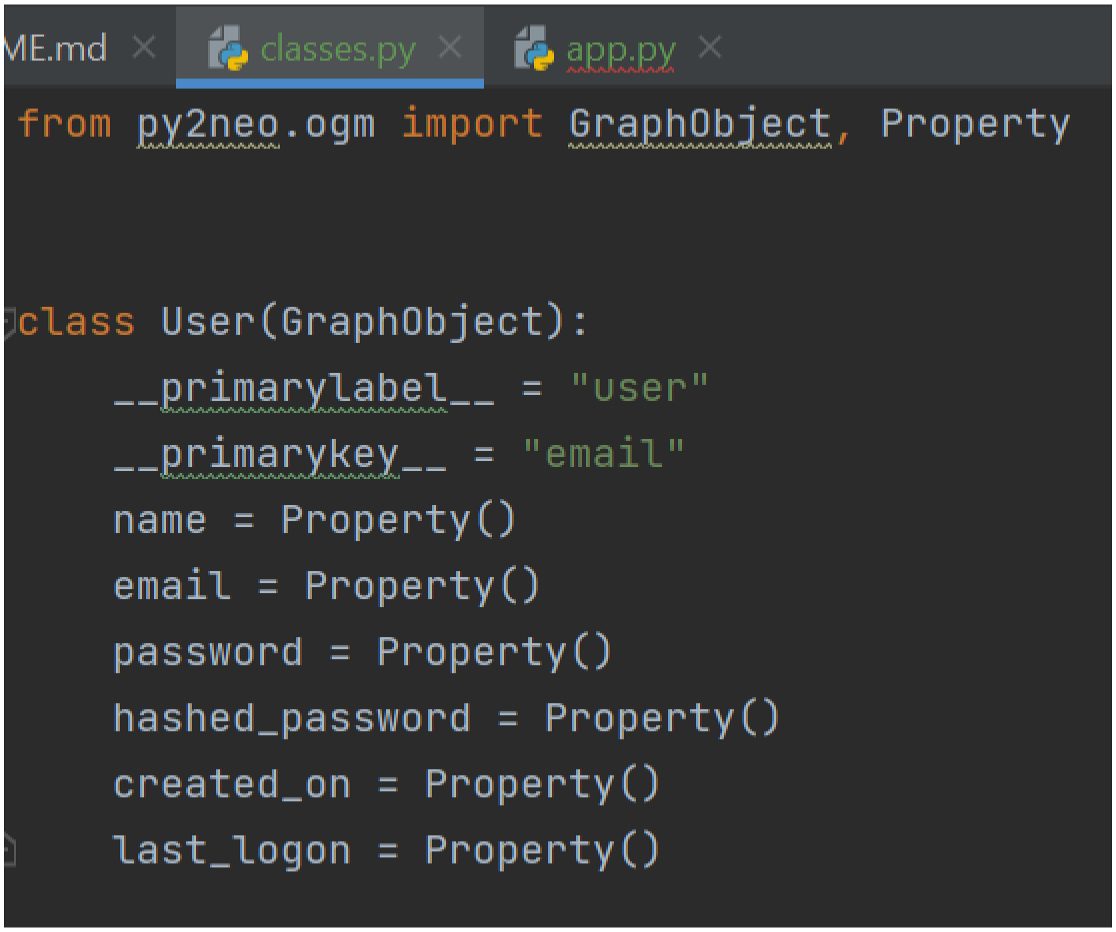
Section 11.2 The Model

We used two different graph databases, one to authenticate the user and issue a token so that a request to the server would not return information unless the token could be decoded and determined to be signed. The other graph database is for the Person and Event node objects and the interestedIn and postedEvent relationships between two Persons or a Person and an Event.

The schema for User, defined as a python class and used within the Graph object, is defined in the classes.py file of the Model directory.

The email attribute of User will serve as the primary key (and is thus required). The label for the node in the Graph associated with an instance of a User will be the name attribute. When a user registers, their password is encrypted and stored within their user node. This is the method used to ensure that the user's information is only available to them during their session and could also be used to set views and permissions within the system. If the system were to have used a token for a user, then a token would be generated with each login of the user, and would be encoded (not encrypted). The token is included with all requests to the server and checked to see if when decoded returns "signed." If signed, then the get request continues; otherwise, an error of invalid user is returned in the terminal.   We chose to use a dictionary datatype and the idea of a session. The user's email is included in the session and if the user's email is a key of the dictionary...then the system uses the values of the dictionary.

Figure   The User Model for session and authenticating login

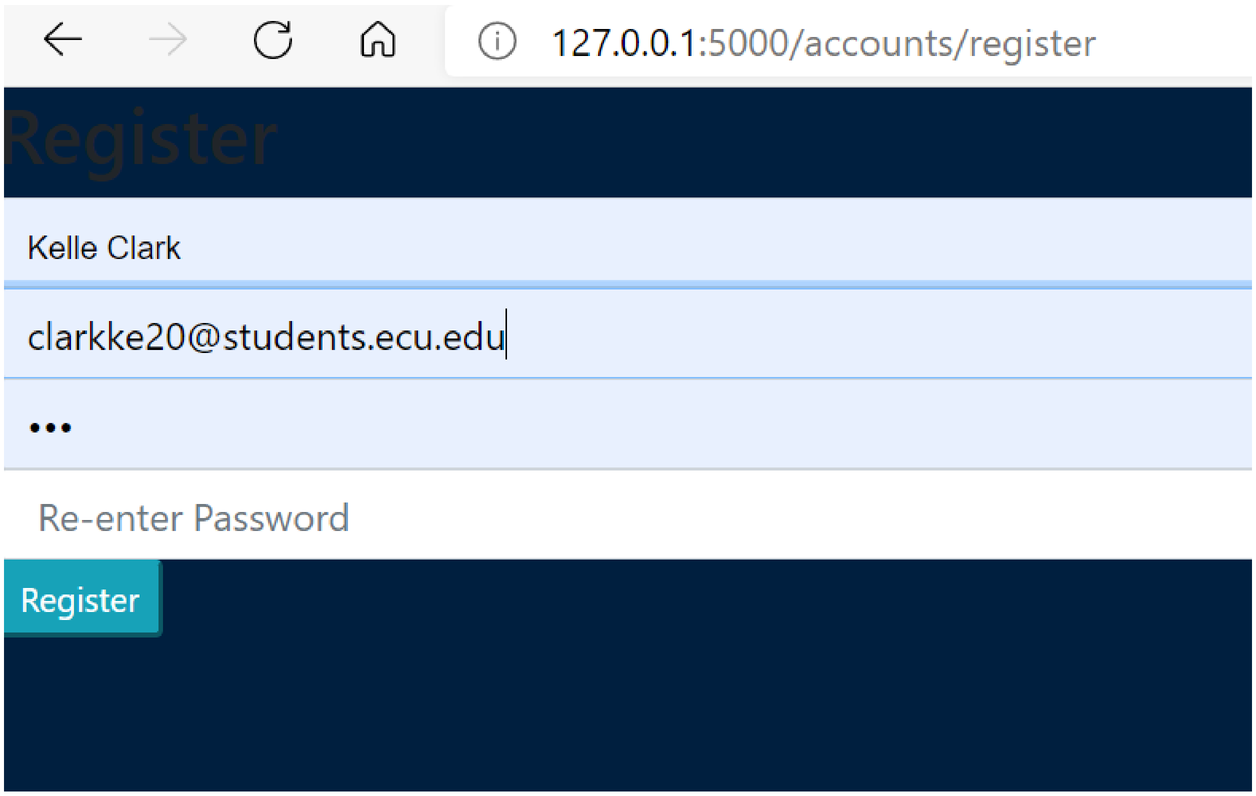


```
from py2neo.ogm import GraphObject, Property


class User(GraphObject):
    __primarylabel__ = "user"
    __primarykey__ = "email"
    name = Property()
    email = Property()
    password = Property()
    hashed_password = Property()
    created_on = Property()
    last_logon = Property()
```

The attributes name, email and password are input from a form that requires these fields to be filled in with type string, email and password on the client side using form control. A warning is issued to the user if the input is not valid. There should be no user in the system with the same email. The attributes created_on and last_logged in are not required and are not currently used.

If the user clicks the sign up button, the system takes in the information and checks to see if the email input is already in the system and if not uses the create_user in account_services.py to add the user to the system. Along with the passed in information, the user node also stores a hashed value of the password using the `sha512_crypt` function within the `passlib.handlers.sha2_crypt` library.

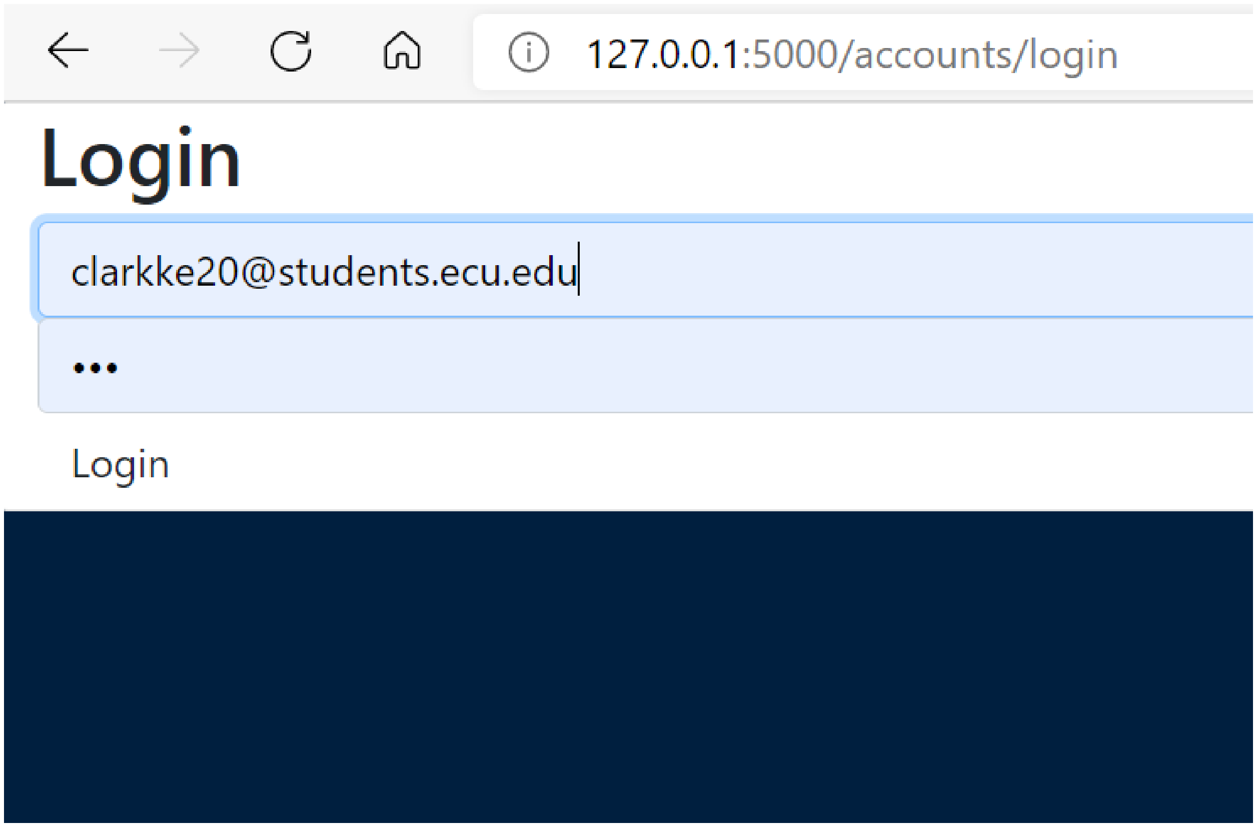
Figure The register page



In the event that the email is already among the users, then a warning is flashed letting the user know that they must use a different set of values to create their account. This check on the collection of nodes is done via a method of the User/Graph object `user = User.match(graph, f"{email}")` and implements a cypher query of the graph database.

If the user clicks the login button, then the system again performs a cypher query using the same method. If the user is found then the session is created for the user in that the email: information pair is added to the session dictionary.

Figure The login page


127.0.0.1:5000/accounts/login

Login

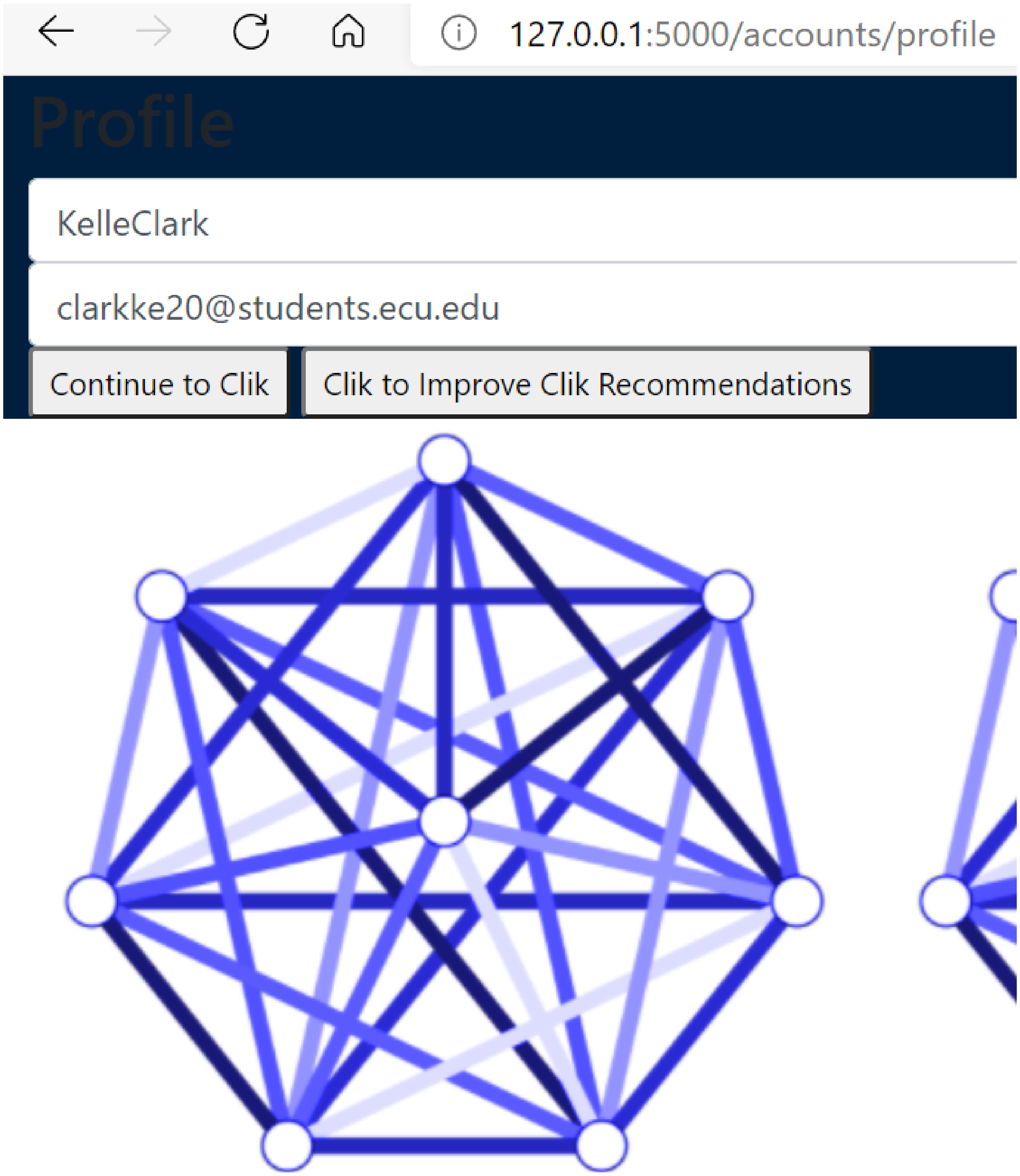
clarkke20@students.ecu.edu

•••

Login

The flask framework uses the provided route to send the browser the user's profile page. On this page, the user can see the information that is stored in the user node for them. Future features would allow a user to update their profile information and to answer questions to help the system to recommend events and friends through Machine Learning algorithms. For now, the user should select to continue to Clik. As a team, the current database that is being accessed through the py2neo driver needs to be stopped and then a new project/database started with the same credentials but that will be connected with the neovis driver to provide utility and visualize the graph database of friends and events.

Figure The profile page



127.0.0.1:5000/accounts/profile

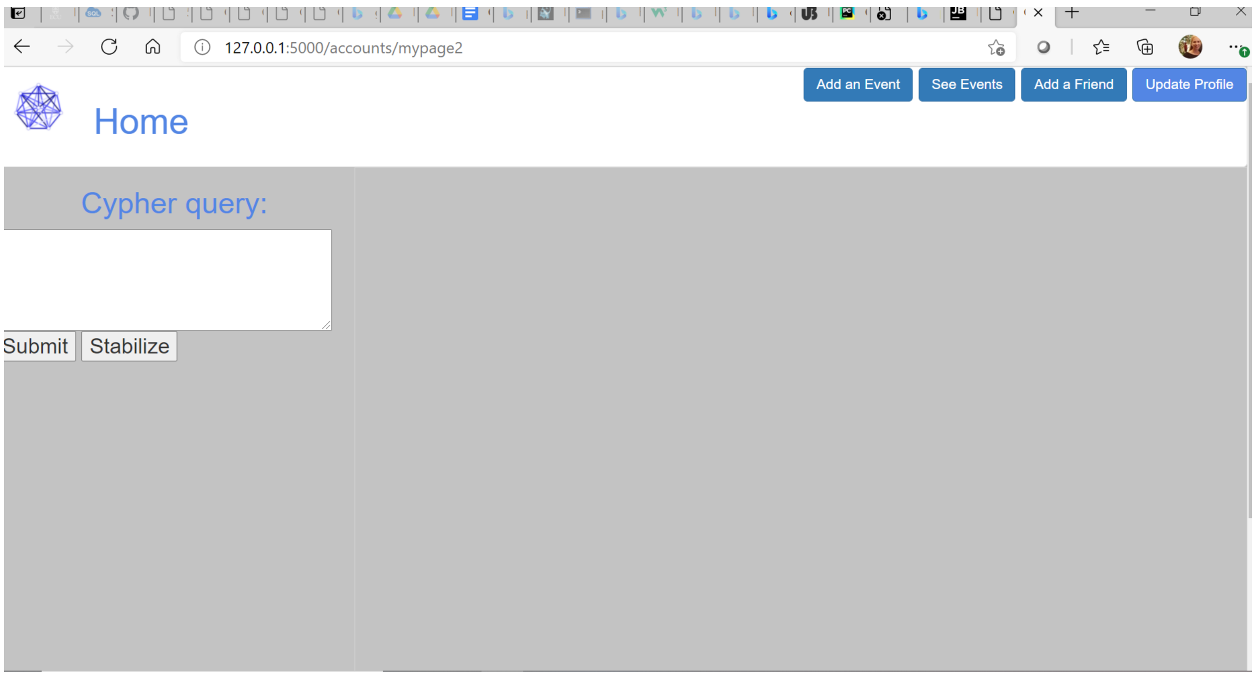
Profile

KelleClark

clarkke20@students.ecu.edu

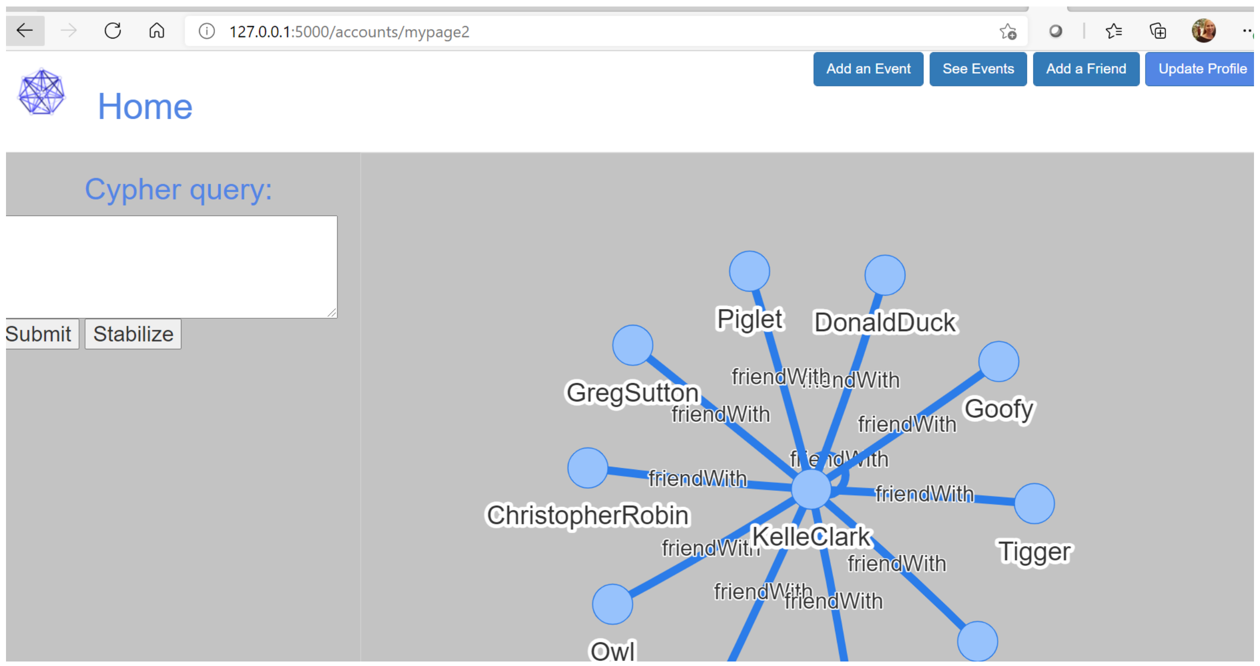
When the continue to clik button is pressed before the other db is stopped and the new database is created, our page looks like this:

Figure Mypage for Clik



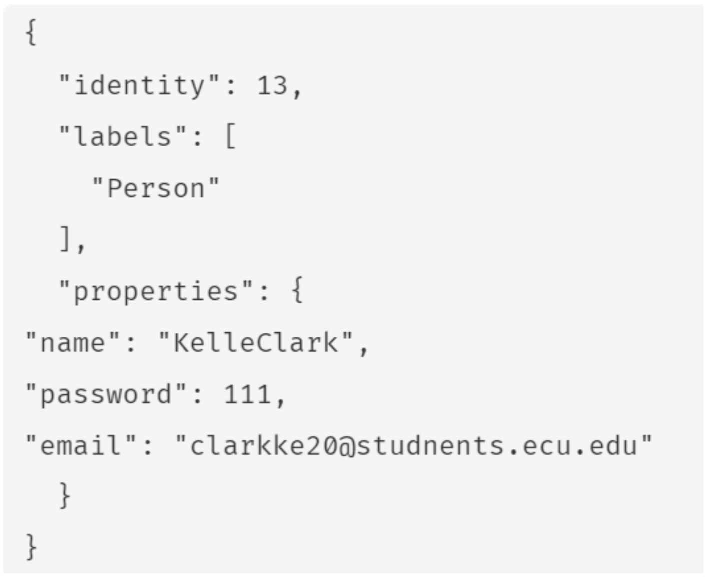
After the new database is started on the desktop and the neo4j browser is opened for the server to connect, the div on the right side will contain the user's friends and events using the initial cypher query. The py2neo connection does use a http:127.0.0.1:7474 url for the graph object, while the neovis connection uses a bolt:127.0.0.1:7687 url. Within the settings file of each db are options for using different ports and protocols...but making it work seamlessly was beyond the time frame and ability currently.

Figure Mypage for Clik when the Friend and Event database is available

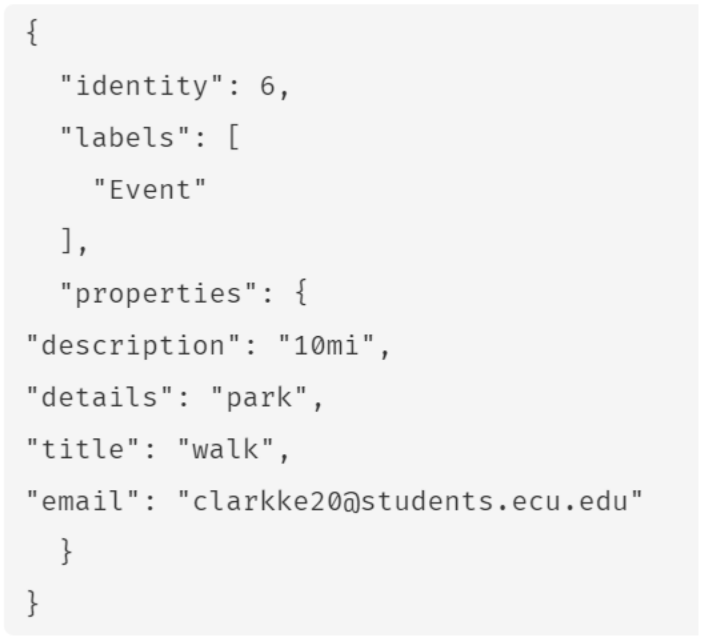


The Nodes and relationships represented in the small clik network of users have similar schema and are best viewed using the Neo4j browser (neo4j@bolt://localhost:7687)

Figure The schema for a Person node and for an Event Node



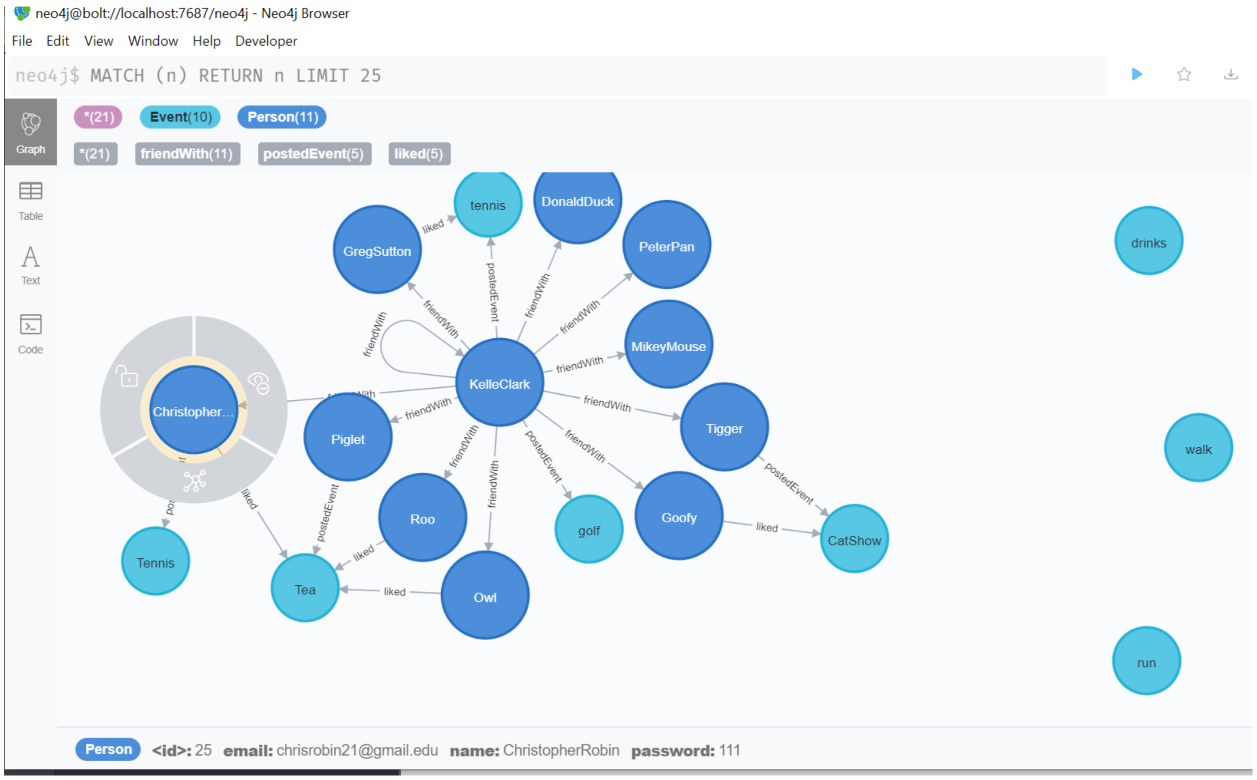
```
{
  "identity": 13,
  "labels": [
    "Person"
  ],
  "properties": {
"name": "KelleClark",
"password": 111,
"email": "clarkke20@studnents.ecu.edu"
  }
}
```



```
{
  "identity": 6,
  "labels": [
    "Event"
  ],
  "properties": {
"description": "10mi",
"details": "park",
"title": "walk",
"email": "clarkke20@students.ecu.edu"
  }
}
```

The fact that information is kept in a graphical database allows many queries to be performed with only a few lines of cypher text and we do not need joins which makes queries faster.

Figure The relationships between Nodes and Events



neo4j@bolt://localhost:7687/neo4j - Neo4j Browser

```
neo4j$ MATCH (n) RETURN n LIMIT 25
```

Person <id>: 25 **email:** chrisrobin21@gmail.edu **name:** ChristopherRobin **password:** 111

From Neo4j, we see that the email for a Person and the node id can both be used as a primary key...the choice of the primary key being the email is based on the context of many use cases. A user would not find it convenient to rely on knowing what a friend's node id is that is assigned to them by neo4j, id(n) where n is the friends' node in the graphical database. Using an email to perform tasks such as add a friend is more appropriate in this use case. In the use case where we would like to return all friends of a user, we know the user's email address from their session key value. Using a cypher query with the match on email address and has a 'friendsWith" relationship allows the system to generate a view of all friends of the user. This is also the case when a user would like to view all events they have posted with the relationship "postedEvent" replacing "friendsWith."

Section 11.3 The View and The Controller

We have seen that the flask framework allows the client (and browser) to request different views for the user like the html pages with css/bootstrap/js/jquery in the templates folder of the application: login page, the create account page, the profile page, and mypage. These requests are controlled using @app.route methods of our flask application within app.py.

The needed libraries for the pages of the application are utilized by the inclusion of scripts and links to url within content delivery networks with as much of this header information included in a _layout.html file and use jinja to extend this file.

Section 11.4 Other Features and Future Features

Section 12 Conclusion