



Advanced Databases Report.

Topic - *City Blogging*

PREPARED FOR

Prof. Dr. Barbara Sprick
Prof. Frank Hefter

PREPARED BY - Team Orange

Akshay Gavandi
Akshay Raul
Snehal Ghule
Shashank Upadhyay
Shivam Sharma

TABLE OF CONTENTS -

1.Introduction	3
2.User Stories	4
3.Application & Database Structure	5
3.1 Application Flow Diagram	6
3.2 UML	7
4.Database Selection	8
4.1 MongoDB	8
4.2 Neo4j	9
4.3 Redis	9
5. Use Cases	10
5.1 Use Case 1 - Authentication - LogIn, LogOut *Author- Shashank Upadhyay	10
5.2 Use Case 2 - User Feed *Author - Snehal Ghule	11
5.3 Use Case 3 - Search * Author - Shivam Sharma	12
5.4 Use Case 4 - Selective Sharing of Information *Author - Akshay Raul	12
5.5 Use Case 5 - Ranking of Users *Author - Akshay Gavandi	13
6. Roles and Responsibilities	14
7. Code related to User Stories.	15
7.1 User Registration:	15
7.2 User LogIn	17
7.3 Relevant User Feed	19
7.4 Content Activity - Create Post	20
7.5 Follow and Unfollow	21
7.6. Relevant Search	22
7.7 Ranking and Reputation	24
7.8 Selective Sharing of Information	25
7.9 Commonly Used Queries used by Group	26
8. Implementation	27
9. Evaluation	31
10. Bibliography	32

1. Introduction

The City Blog is a social media application focusing on the particular geographical location i.e. a city. The application focuses on linking the community of a city together. This application is for the community of international students arriving in Deutschland for their education or job. This application focuses on solving their common issues which almost everyone faces and forms a community. This will help the university administrative staff to reduce the load. End-Users can Follow other Users, Create, Comment, and Like Posts and many other things that are present on the application. Another user would be an administrator who can upload some static information/ news and has the right to moderate the content of the post on the application and it has the power to remove and update the user status.

For this project, we are using NodeJS for server, ReactJS for Client. MongoDB, Neo4j and Redis as our polyglot persistence.

We used Docker for the Multicontainer environment where all our depending services are running , Git for version control and VS code as an IDE for developing this application.

Kindly, Check out the Repository for code and queries written for this project on - [GITHUB](#).

2. User Stories

User Story 1 - Registration - As a User, I want to register in the application so that I can see the content feed and see what others of my community are doing.

User Story 2- Authentication - Login, LogOut - As a Registered user I can LogIn do my activities and LogOut when necessary.

User Story 3 - Content Activity - As a Logged In user I want to create content, Like and Comment on other content as well

User Story 4 - Follow/Unfollow- As a Logged In user I want to follow other user so that i can see what they are doing

User Story 5 - Feed - As a Registered User, I Log In for the first time and see my feed is related to the posts which lie in the category which i selected during the registration time. As soon as I follow someone my main feed is changed, to the content created by the users whom I am following.

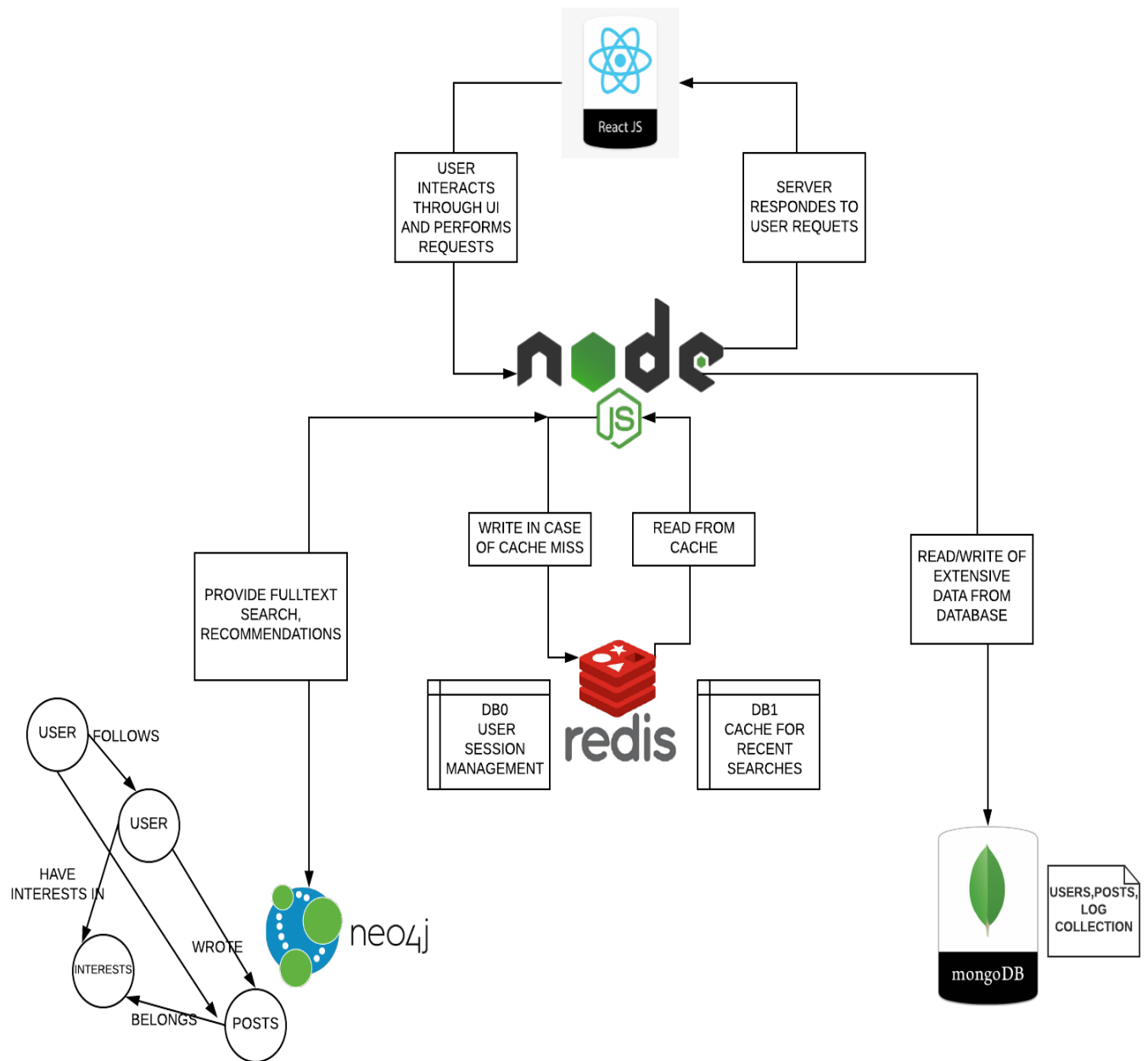
User Story 6 - Relevant Search- As a Logged In User I want to search users and posts. And the search should be relevant to me in some way so that I don't have to crawl over all of the content of the blog.

User Story 7 - Privacy - As a Logged in User I want to create a content which is not visible to the persons, which I state during the content creation form.

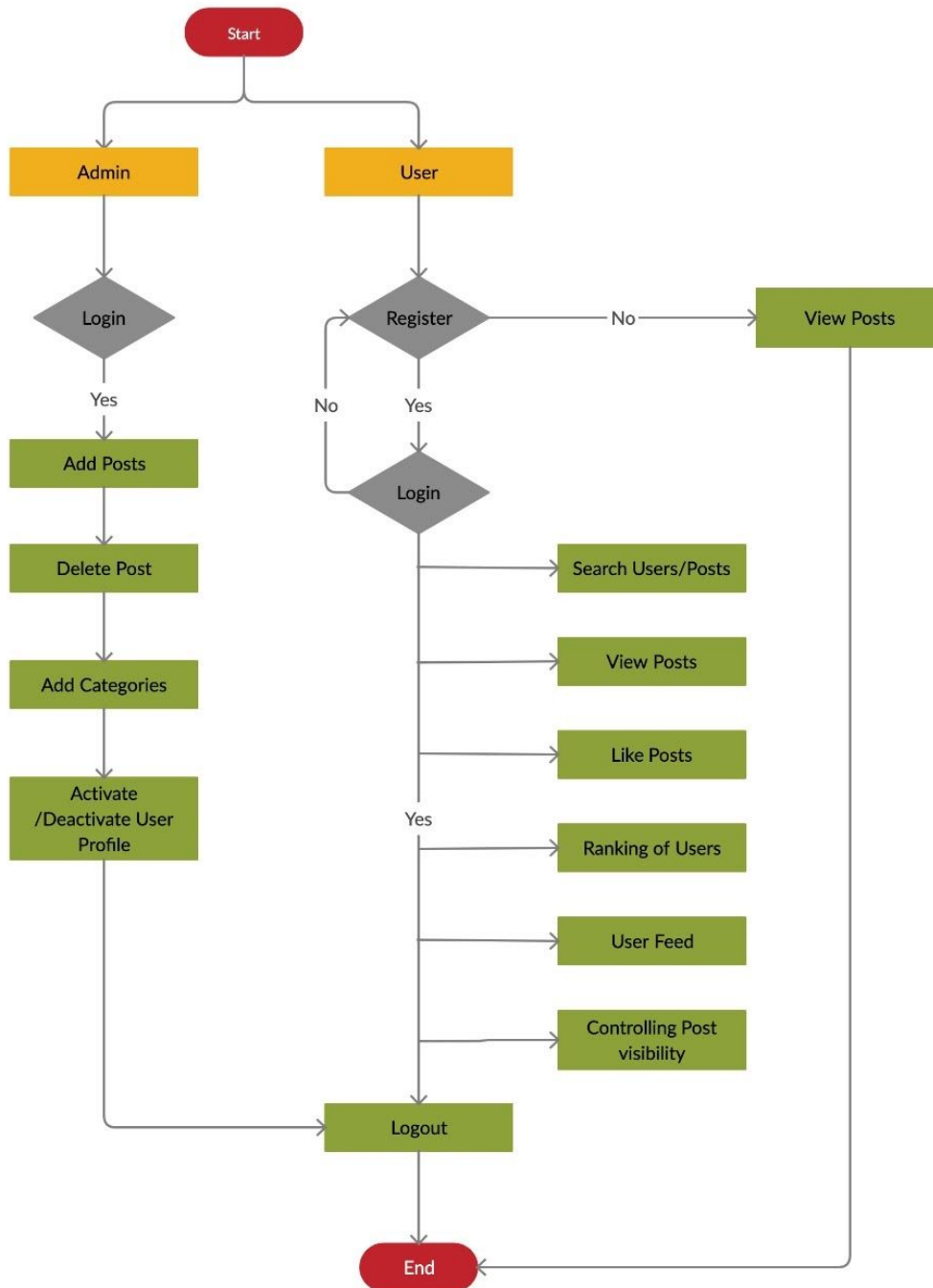
User Story 8 - Ranking and Reputation - As a Logged In user I want to upvote someone's profile if I like all of their content and the content they post really helps me out. And also I want to see the Users who have the highest reputation in the categories so that I can read the quality content.

User Story 9 - Administrator Role - As an Admin I want to activate/ deactivate the user profiles who are not following the community guidelines or their activity seems suspicious.

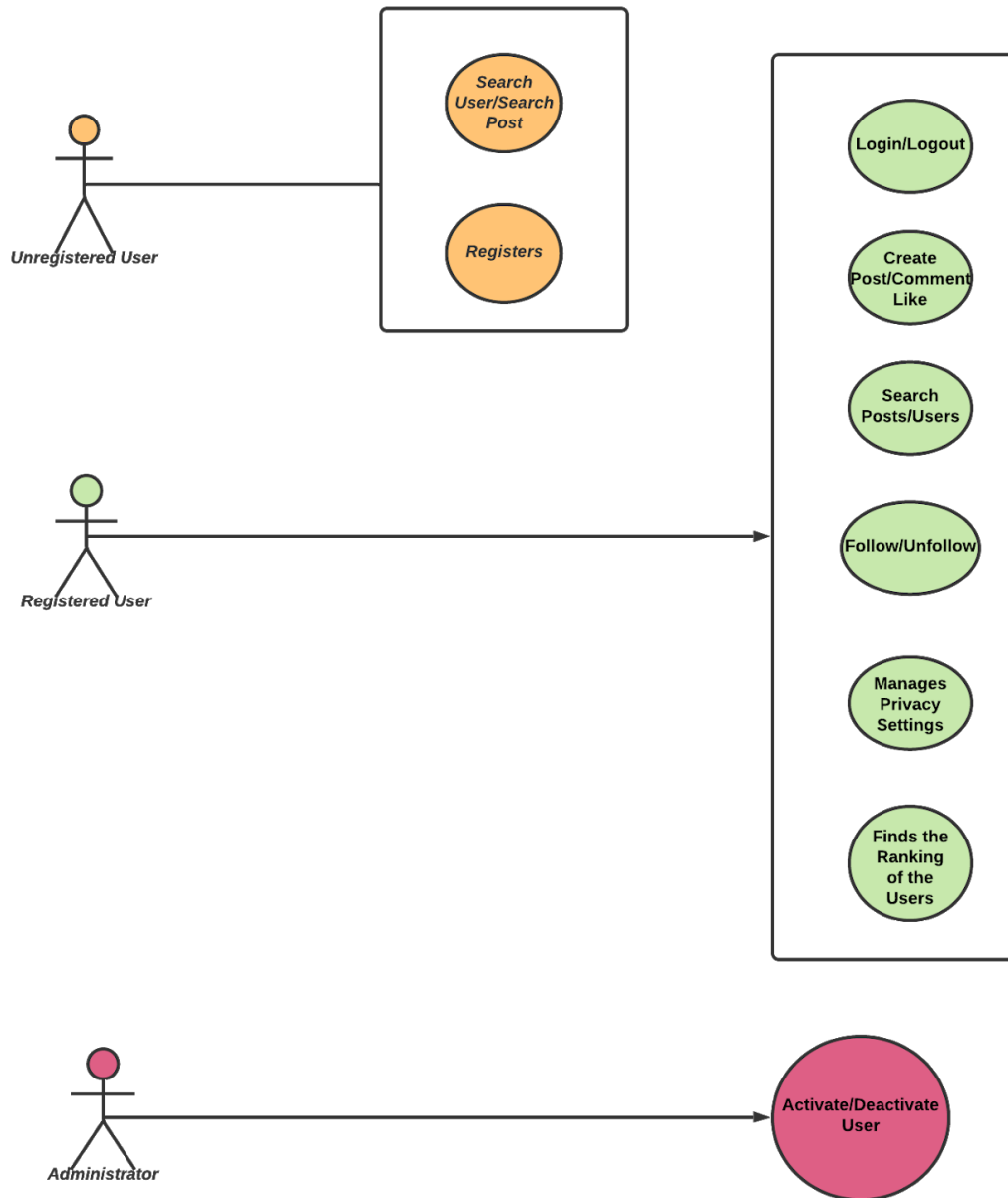
3. Application & Database Structure



3.1 Application Flow Diagram



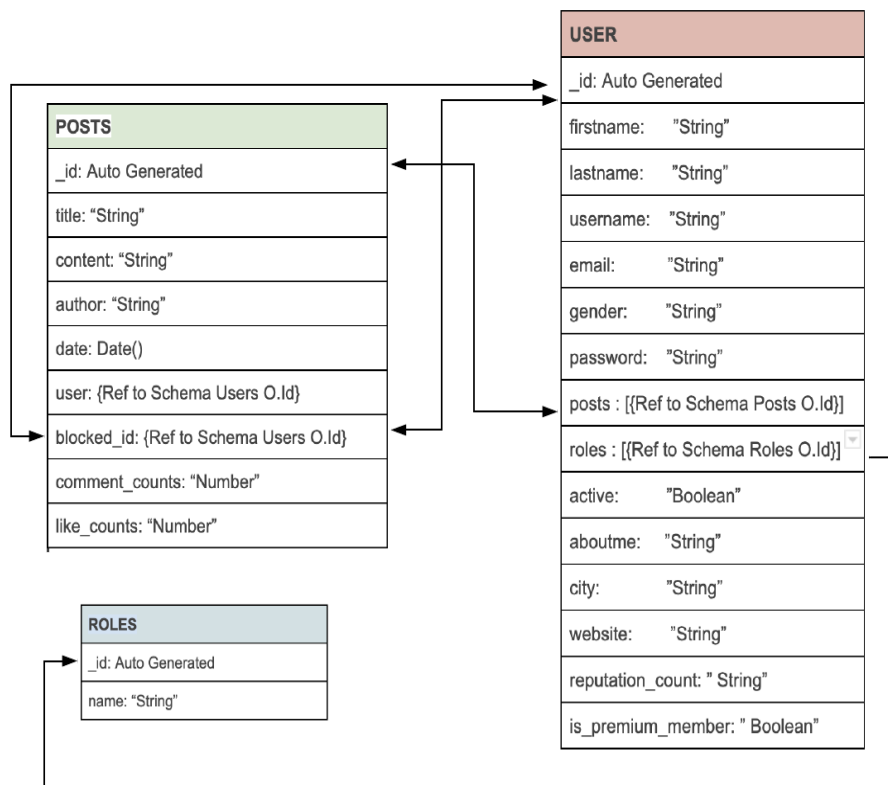
3.2 UML



4. Database Selection

4.1 MongoDB

MongoDB is used in this project for storing extensive data. Images and data of posts and users require a large storage. Since MongoDB supports sharding, that will help to scale, when the application will have more users to handle. Horizontal sharding will help in this case and achieve distributed architecture. MongoDB also has a high availability and fast recovery which ensures Users never lose their data. In addition MongoDB gives the flexibility to have a dynamic schema. So for an update of the data model, previous data is not affected.



4.2 Neo4j

Neo4J reduced the complexity we used to face in joins in the RDBMS. By the implementation of graph (traversing of nodes with relationships) with CYPHER query improved the readability and simplified the relation of data to one another. It also helps in reducing the load by quickly traversing thousands of nodes and reduces the query time compared to any relational database by a huge margin. By having the ACID properties and use of constraints, transactions and indexes for fast read, we were left with this database.

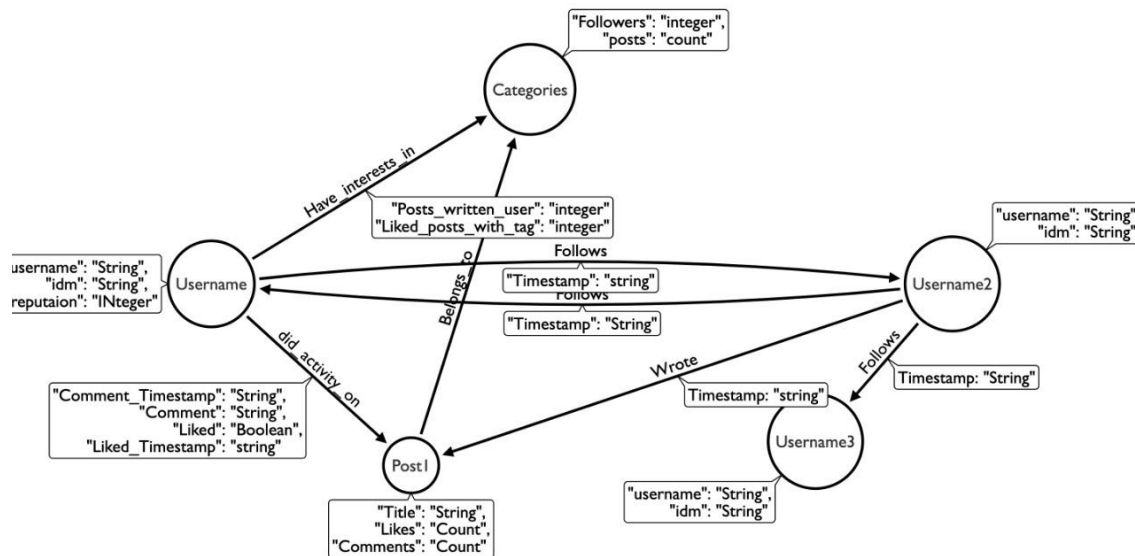


Figure - Arrow Diagram of Nodes and Relationships.

4.3 Redis

We have used two databases of redis for the entirety of two separate purposes.

DB0 is used for user session management when the user Logs In, SessionId created by our server is stored on Redis which helps us to save the memory of the server and at the same time maintains a fast access for Login. And out of the box functionality like 'Expiry' for TTL of cookies helps to maintain it.

```
"set" "sess:62xWlnU8PpBm1vhKd16YzeWUWWfo9LnI"
{"cookie":{"originalMaxAge":7200000,"expires":"2020-06-11T14:11:17.105Z","secure":true,"httpOnly":true,"path":"/","sameSite":true},"user":{"blocked_id":[],"posts":["5edfdbc2d96ffad738f013f"],"roles":{"_id":"5ecf84df1a3ea800133bd67f","name":"user"},"_id":"5edf4411f327d8954b13fb6a","firstname":"test","lastname":"testing","username":"Richard","email":"Richard@gmail.com","gender":"male"}} "EX" "7200"
```

DB1 is used for saving searches and providing a fast access of user queries to the client. A cache miss would fetch the request from Mongo or Neo and save that result in Redis and display it to the client, where Key is user query and Value is the result.

Storing Key Value for below mentioned lead us to choose Redis -

- To have User sessions maintained on a fast read,write store.
- To store user searches temporarily.
- A database where durability is not the highest priority

5. Use Cases

5.1 Use Case 1 - Authentication - Login, Logout *Author- Shashank

Upadhyay

Description - Authentication(Register, log in, and logout) functionality using session cache by Redis. Redis will enable fast access to session data as it keeps the key-value pair in memory by building a caching layer which will also increase the speed of page load. This data exists until the session expires which helps the respective user to quickly access the basic information.

Data Flow: After a successful Registration, the user will be redirected to the Login page at UI in which login credential (Username and password) is required to login in a validated form.

1. After Login, the credential entered by the user is checked by a validation on the server where it finds the user, using a findOne query on mongo and then a function written for a session generates a session Id and value which contains the user data which is then set to redis with an expiry for that session.

After a session is created, state is managed by an `isLoggedIn` function that holds the session details which provide the user details for further actions performed by the user in that session.

2. Logout: Once user hits the signout route with an `Access-Control-Allow-Origin` header which matches with the existing credentials and a logout function at backend destroys the generated session and removes the cookie from the client storage. New session Id will be generated for a new login.

User - Registered user.

Data Sources and interactions- Interaction is done with the frontend `LogIn` page which communicates with NodeJS server where data sources (Neo4j and MongoDB) are integrated.

5.2 Use Case 2 - User Feed *Author - Snehal Ghule

Description - When the user will register his /her interests will also be captured from the categories he/she selects. The feed initially will be the trending posts of that category. As soon as a user follows another user, the feed will change to the connections posts on the application.

Data Flow – After login the User relation (like follow/unfollow) with other Users will be checked if User do not follow the other Users then, with respect to the User interest the filter query is executed and the related posts will be filtered, then the `Post_id` in Neo4j is compared with the `Post_id` in MongoDB and the data stored in MongoDB (like Title and Content) will be displayed as node Properties and if the User follows the other Users then first the followed Users list will be filtered and then the Posts of those Users will be displayed.

User – Registered Users.

Data Sources and interactions- Interaction is done with the frontend feed page which communicates with NodeJS server where data sources (Neo4j and MongoDB) are integrated.

5.3 Use Case 3 - Search * Author - Shivam Sharma

Description- A search functionality where users can search posts or users. The list of search results for users/posts is sorted by relevance to the logged in user. Using Neo4j, we can find out the relation of a logged in user to another user and posts. Full-text Schema Index powered by Apache Lucene is used to achieve this use case.

Data Flow - A search box in the UI will capture the keyword and Search user/post button will trigger the route associated with it.

1. For search of users, a higher score is set in the index if the user is following the searched keyword, relative to normal score which is for other users.
2. For search of posts, a higher score is set in the index if the user has written a post matching the searched keyword, relative to normal score which is for other posts.
3. The index is ordered by score, the score is boosted/set higher according to the conditions. The result is then formatted in the form of list and is displayed to the frontend, where the list item is clickable and redirects to a page which is more descriptive. That click event on lists then again fetches the descriptive data from the MongoDB.

User - Registered user

Data Sources and Interactions - Interaction is done with the Frontend search available on the sidebar of all the pages which communicates with NodeJS server where data sources (Neo4j and MongoDB) are integrated.

5.4 Use Case 4 - Selective Sharing of Information

*Author - Akshay Raul

Description -The registered user before posting content selects usernames which are blocked from viewing this particular post.

Data Flow:

- 1) User initiates create post API
- 2) Post's data which include blocked ids are inserted in the posts collection
- 3) Blocked User accesses posts using API to fetch posts
- 4) Data of posts that have the currently logged in user's id mapped to it as blocked id are not fetched and not accessible.

User:Registered User

Data Sources and Interactions - Blocked Id can be mentioned which communicates with NodeJS server where data source of (MongoDB).

5.5 Use Case 5 - Ranking of Users *Author - Akshay Gavandi

Description -The reputation of a user can be upvoted by another user and credits/points of that user will increase and be displayed under profile details. The higher the number of upvotes the more probability it is that the user is a valuable content writer. The reputation will be saved in Neo4j. For example a user has 57% of posts on topic 1, 35% on topic 2 and 8% on topic 3. A ranking page of users will be available. All top users will be displayed with the topic.

Data Flow-

1. Search user ranking.
2. Select a category of posts.
3. Get the posts by users in the selected category from Neo4j.
4. Fetch user reputation from Neo4j users collection.
5. List the ranking of users.
6. Listing first ordered by reputation and second ordered by the number of likes on the Post.

User - Registered User

Data Sources and Interactions - The reputation and the number of likes on the Post stored in Neo4j will be calculated.

6. Roles and Responsibilities

ACTIVITIES	AKSHAY GAVANDI	AKSHAY RAUL	SNEHAL GHULE	SHASHANK UPADHYAY	SHIVAM SHARMA
DATABASE DESIGN	R	R	R	R	R
USE CASE DESIGN	R	R	R	R	R
User Story 1 - Registration	I	I	I	R	R
User Story 2 - Login/Logout	I	I	I	R	C
User Story 3 - Content Activity	I	I	I	R	R
User Story 4: Follow/Unfollow	I	I	I	R	R
User Story 5: Feed of content	I	I	R	I	R
User Story 6: Relevant Search	I	I	I	I	R
User Story 7: Privacy	I	R	I	I	I
User Story 8: Ranking and Reputation	R	I	I	I	I
User Story 9: Administration	I	I	I	I	R
Use Case 1: Authentication Login, Logout	I	I	I	R	I
Use Case 2: User Feed	I	I	R	I	C
Use Case 3: Search	I	I	I	I	R
Use Case 4: Selective sharing of information.	I	R	I	I	I
Use Case 5: Ranking and Reputation	R	I	I	I	I
MOCK DATASET	I	I	I	I	R
INFORMATION GATHERING	R	R	R	R	R
DOCKER SETUP	I	I	I	I	R
DOCUMENTATION	C	C	C	C	R

RESPONSIBLE	R
INFORMED	I
CONSULTED	C

7. Code related to User Stories.

7.1 User Registration:

```
exports.signup = async (req, res) => {
  const user = new User({
    firstname: req.body.firstname,
    lastname: req.body.lastname,
    username: req.body.username,
    email: req.body.email,
    gender: req.body.gender,
    password: bcrypt.hashSync(req.body.password, 8),
    active: true,
  });

  user.save((err, user) => {
    if (err) {
      res.status(500).send({ message: err });
      return;
    }

    if (req.body.roles) {
      Role.find(
        {
          name: { $in: req.body.roles },
        },
        (err, roles) => {
          if (err) {
            res.status(500).send({ message: err });
            return;
          }

          user.roles = roles.map((role) => role._id);
          user.save((err) => {
            if (err) {
              res.status(500).send({ message: err });
              return;
            }

            res.send({ message: "User was registered successfully!" });
          });
        }
      );
    } else {
      Role.findOne({ name: "user" }, (err, role) => {
        if (err) {
          res.status(500).send({ message: err });
          return;
        }
      });
    }
  });
}
```

```
    }
    user.roles = [role._id];
    user.save((err) => {
      if (err) {
        res.status(500).send({ message: err });
        return;
      }
      res.send({ message: "User was registered successfully!" });
    });
  });
}
});
const session = driver.session();
idm = user.id;
const fullname = user.firstname + " " + user.lastname;
console.log(fullname);

await session
  .run("CREATE (n:Person {name: $username, pidm:$idm,
fullname:$fullname})", {
    username: req.body.username,
    idm: idm,
    fullname: fullname,
  })
  .then(() => {
    session.close()
  });
const cgt = req.body.cgt;

cgt.forEach(createrel);
async function createrel(categoryName) {
  const session2 = driver.session();
  console.log(categoryName);
  const username = req.body.username;
  await session2.run(
    "MATCH (a:Person), (b:Category) WHERE a.pidm = $idm AND b.name =
$categoryName CREATE (a)-[: Have_interests_in {created_at:
TIMESTAMP ()}]>(b) ",
    {categoryName: categoryName, idm: idm, });
  await session2.close();
}
};
```


7.2 User Login

```
exports.signin = async (req, res) => {
  User.findOne({
    username: req.body.username,
  })
    .populate("roles", "-_v")
    .exec((err, user) => {
      if (err) {
        res.status(500).send({ message: err });
        return;
      }

      if (!user) {
        return res.status(404).send({ message: "User Not found." });
      }

      var passwordIsValid = bcrypt.compareSync(
        req.body.password,
        user.password
      );

      if (!passwordIsValid) {
        return res.status(401).send({ message: "Invalid Password!", });
      }

      if (!user.active === true) {
        return res.status(401).send({
          message:
            "Your account has been deactivated. Kindly contact administrator.",
        });
      }

      req.session.user = user;

      var authorities = [];

      for (let i = 0; i < user.roles.length; i++) {
        authorities.push("ROLE_" + user.roles[i].name.toUpperCase());
      }

      const session4 = driver.session();
      const pidm = user._id;
      session4
        .run(
          `MATCH (p1:Person)
           WHERE p1.pidm= "${pidm}"
           MATCH (p1:Person)<-[:Follows]-(p3:Person)
           RETURN p3.pidm , p3.fullname`,
          {
            pidm: pidm,
          }
        )
        .then((result) => {
          const followers = [];
          result.records.forEach((record) => {
```

```

        followers.push({
            id: record._fields[0],
            fullname: record._fields[1],
        });
    });
    follower = followers;
})

.catch((error) => {
    console.log(error);
})
.then(() => {
    session4.close(() => {});
    const session5 = driver.session();
    session5
        .run(
            `MATCH (p1:Person)
WHERE p1.pidm= "${pidm}"
MATCH (p1:Person)-[:Follows]->(p3:Person)
RETURN p3.pidm , p3.fullname ` ,
            {pidm: pidm,})
        .then((result) => {
            const followings = [];
            result.records.forEach((record) => {
                followings.push({
                    id: record._fields[0],
                    fullname: record._fields[1],
                });
            });
            following = followings;
            res.status(200).send({
                id: user._id,
                firstname: user.firstname,
                lastname: user.lastname,
                username: user.username,
                email: user.email,
                roles: authorities,
                aboutme: user.aboutme,
                city: user.city,
                website: user.website,
                posts: user.posts,
                follower, following,
            });
        })

        .catch((error) => {
            console.log(error);
        })
        .then(() => {
            session5.close(() => {});
        });
    });
});
};

```

7.3 Relevant User Feed

```
exports.getPost = async (req, res) => {
  var data = [];
  const userid = req.params.id;
  const session4 = driver.session();
  const session5 = driver.session();
  const session6 = driver.session();
  session4
    .run(
      `MATCH (p:Person)
      WHERE p.pidm= $id
      MATCH (p)-[:Follows]->(p2) WITH collect(p2.pidm) AS followers RETURN
followers`,
      {
        id: userid,
      }
    )
    .then((result) => {
      const data3 = [];
      result.records.forEach((record) => {
        data3.push(record._fields[0]);
      });
      c = data3[0].length;

      if (c <= 1) {
        session5
          .run(
            `MATCH
(p1:Person(pidm:$id))-[:Have_interests_in]->(c:Category)-[:Belongs_to]-(p3:Post)
RETURN p3.title, p3.pidm `,
            {
              id: userid,
            }
          )
          .then((result) => {
            const data1 = [];
            result.records.forEach((record) => {
              data1.push({
                title: record._fields[0],
                id: record._fields[1],
              });
            });
            const page = parseInt(req.query.page) || 1;
            const pageSize = 5;
            const pager = paginate(data1.length, page, pageSize);
            const pageOfItems = data1.slice(
              pager.startIndex,
              pager.endIndex + 1
            );
            res.status(200).send({ pager, pageOfItems });
          })
          .catch((error) => {
            res.send(error);
          })
          .then(() => {
            session5.close(() => {});
          });
      }
    });
}
```

```

    });
  } else {
    session6
      .run(
        `MATCH (p:Person)
        WHERE p.pidm= $id
        MATCH (p)-[:Follows]->(p2)
        MATCH (p2)-[:Wrote]->(p3:Post)  RETURN p3.pidm ,p3.title`,
        {
          id: userid,
        }
      )
    .then((result) => {
      const data2 = [];
      result.records.forEach((record) => {
        data2.push({
          title: record._fields[1],
          id: record._fields[0],
        });
      });
      console.log("inhere");
      const page = parseInt(req.query.page) || 1;
      const pageSize = 5;
      const pager = paginate(data2.length, page, pageSize);
      const pageOfItems = data2.slice(
        pager.startIndex,
        pager.endIndex + 1
      );
      res.status(200).send({ pager, pageOfItems });
    })
    .catch((error) => {
      res.send(error);
    })
    .then(() => {
      session6.close(() => {
        console.log(` Followed`);
      });
    });
  }
})
.catch((error) => {
  res.send(error);
})
.then(() => {
  session4.close(() => {
    console.log(` Followed`);
  });
});
});
};

```

7.4 Content Activity - Create Post

```
exports.submitPost = async (req, res) => {
  const session = driver.session();
  const session2 = driver.session();
  user = req.params;
  id = user.id;
  const { title, content, selectedCgt } = req.body;
  const userById = await User.findById(id);
  const username = userById.username;
  console.log(selectedCgt);
  const post = await Post({
    title: title,
    content: content,
    user: id,
    author: username,
  });
  await post.save();

  userById.posts.push(post);
  await userById.save();
  const id1 = post.id;
  const writeTxPromise = session.writeTransaction((tx) =>
    tx.run(
      "MATCH ( a:Person { name: $username }) MERGE ( b: Post { title:$title, likes: 0, pidm: $id1, comment: 0}) MERGE (a)-[: Wrote {created_at: TIMESTAMP()}]->(b)",
      { username: username, id1: id1, title: title }
    )
  );
  writeTxPromise.then(() => {
    console.log("in the write");
    const write1TxPromise = session.writeTransaction((tx) =>
      tx.run(
        " MATCH ( a:Post {idm: $id1}), (b:Category{name : $selectedCgt}) CREATE (a)-[: Belongs_to ]->(b)",
        {
          id1: id1,
          selectedCgt: selectedCgt,
        }
      )
    );
  });
  write1TxPromise.then(() => {
    session.close();
    console.log("Matched created node with id ");
  });
};

return res.status(200).send("added");
};
```

7.5 Follow and Unfollow

```
//FOLLOW
exports.follow = async (req, res) => {
  const session = driver.session();
  const id1 = req.params.id;
  const id2 = req.params.id2;
  session
    .run(
      "MATCH (a:Person), (b:Person) WHERE a.pdm = $username1 AND b.pdm = $username2 MERGE (a)-[: Follows {created_at: TIMESTAMP()}]->(b) ",
      {
        pdm: id1,
        pdm2: id2,
      }
    )
    .then(() => {
      session.close(() => {
        console.log(` Followed`);
      });
    });
  res.status(200).send(`${username1} followed ${username2}`);
};

//UNFOLLOW
exports.unfollow = async (req, res) => {
  const session = driver.session();
  const u = req.params;
  const id1 = u.id;
  const id2 = req.params.id2;

  session
    .run(
      "MATCH (a:Person)-[r: Follows]->(b:Person) WHERE a.pdm = $id1 AND b.pdm = $id2 DELETE r ",
      {
        id1: id1,
        id2: id2,
      }
    )
    .then(() => {
      session.close(() => {
        console.log(` Unfollowed`);
      });
    });
  res.status(200).send(`${id1} unfollowed ${id2}`);
};
```

7.6. Relevant Search

7.6.1 For User-

```
exports.searchuser = (req, res) => {
  const session = driver.session();
  const userid = req.params.id;
  const kkeyword = req.params.q;
  session
    .run(
      ` MATCH (p:Person)
        WHERE p.pidm= $id
        MATCH (p)-[:Follows]->(following)
        WITH collect(following.pidm) AS followers
        WITH "(" + apoc.text.join( followers, " OR " ) + ")^3" AS queryPart
        CALL db.index.fulltext.queryNodes('persons', 'fullname: ${kkeyword} pidm: '
+ queryPart)
        YIELD node, score
        RETURN node.pidm, node.fullname,  score `,
      {
        id: userid,
        kkeyword: kkeyword,
      }
    )
    .then((result) => {
      console.log(result);
      const data = [];
      result.records.forEach((record) => {
        data.push({
          id: record._fields[0],
          name: record._fields[1],
          score: record._fields[2],
        });
      });

      res.send(data).status(200);
    })
    .catch((error) => {
      console.log(error);
    })
    .then(() => {
      session.close(() => {});
    });
};
```

7.6.2 Search For Post-

```
exports.searchpost = (req, res) => {
  const session = driver.session();
  const id = req.params.id;
  const kkeyword = req.params.q;
  session
    .run(
      ` MATCH (p:Person)
        WHERE p.pidm= "${id}"
        MATCH (p)-[:Wrote]->(post)
        WITH collect(post.pidm) AS mypost
        WITH "(" + apoc.text.join( mypost, " OR " ) + ")^3" AS queryPart
CALL db.index.fulltext.queryNodes('myposts', 'title: ${kkeyword} pidm: ' +
queryPart)

        YIELD node, score
        RETURN node.pidm, node.title, score `,
      {
        id: id,
        keyword: kkeyword,
      }
    )
    .then((result) => {
      console.log(result);
      const post = [];
      result.records.forEach((record) => {
        post.push({
          id: record._fields[0],
          title: record._fields[1],
          score: record._fields[2],
        });
      });

      res.send(post);
    })
    .catch((error) => {
      console.log(error);
    })
    .then(() => {
      session.close(() => {
        console.log(` Followed`);
      });
    });
};
```


7.7 Ranking and Reputation

```
// Query that sets a like property on Did_activity_on relationship.If the User
likes the Post the Likes count is set to 1 and if dislikes then it is set to
0.Initially the likes will be set to 0.

MATCH (a:Person),(b:Post)

WHERE a.name = 'John' AND b.idm = "5edf61d6f327d8954b13fb75"

CREATE (a)-[r:Did_activity_on { name: a.name + '->' + b.name }]->(b)

set r.likes="0"

RETURN type(r), r.name

match
(n:Person{name:'John'})-[k:Did_activity_on]->(m:Post{idm:"5edf61d6f327d8954b13fb75"
}) set k.likes="1" return n,m

// Now for reputation :- The likes are counted by all incoming Did_activity_on
where likes property value is 1.

match(p:Person) -[r:Wrote]->(t:Post)-[r1:Belongs_to]-> (c:Category)

match (u:Person) -[e:Did_activity_on{likes:"1"}]-> (t)

return p.name, c.name,t.name,count(e) as count

// Last step get all the posts which user has written along with the Category and
total likes on that Posts and the Reputation of the User Profile. The User
reputation will be sorted on the post likes and Profile Reputation.

match(p:Person) - [r:Wrote] -> (t:Post) - [r1:Belongs_to] -> (c:Category) where
c.name='Universities'

match (p1:Person)-[r2:Did_activity_on{likes:"1"}]->(t)

return p.name,c.name,t.name, count(r2),p.reputation

ORDER BY count(r2) DESC , p.reputation DESC
```

7.8 Selective Sharing of Information

```
Post.find({
  $and:[
    { $or: [
      { title: { $regex: search, $options: "i" } },
      { content: { $regex: search, $options: "i" } },
      { author: { $regex: search, $options: "i" } },
    ]},
    {
      "blocked_ids": {"$nin": [id]}
    }
  ]
});
```

7.9 Commonly Used Queries used by Group

```
CREATE CONSTRAINT ON (p: Person) ASSERT p.idm IS UNIQUE
CREATE CONSTRAINT ON (a: Post) ASSERT a.idm IS UNIQUE

CALL apoc.import.csv([{"fileName": 'file:/adcityblog.csv', labels:
['Person','Posts','Categories'] ,type:['Belongs_to','Did_activity_on', 'Follows',
'Wrote', 'Have_interests_in']}], [], {})

MATCH (p:Person {username: "Dinesh"})-[:Follows]->(following)
WITH "(" + apoc.text.join(following(following.idm), 'OR') + ")"^1 AS queryPart
CALL db.index.fulltext.queryNodes('persons', 'name: Monic id: ' + queryPart)
YIELD node, score
RETURN node, score

CALL db.index.fulltext.createNodeIndex('persons', ['Person'], ['fullname', 'idm'])
// get the users searches.
MATCH (p:Person {name: "Dinesh"})<-[:Follows]-(following)
WITH "(" + apoc.text.join( collect( following.idm), 'OR') + ")"^2 AS queryPart
CALL db.index.fulltext.queryNodes('persons', 'name: mon id: ' + queryPart)
YIELD node, score
RETURN node, score
// to change the names of all properties of nodes
MATCH (p:Post)
WITH collect(p) AS posts
```

```
call apoc.refactor.rename.nodeProperty("name", "title", posts)
YIELD committedOperations
RETURN committedOperations

// change properties all
MATCH (post:post)
WITH collect(post) AS post
CALL apoc.refactor.rename.nodeProperty("idm", "pidm", post)
YIELD committedOperations
RETURN committedOperations

// Match and update:
MATCH (n:Person {name: 'Dinesh'})
SET n.reputation = 1
RETURN n

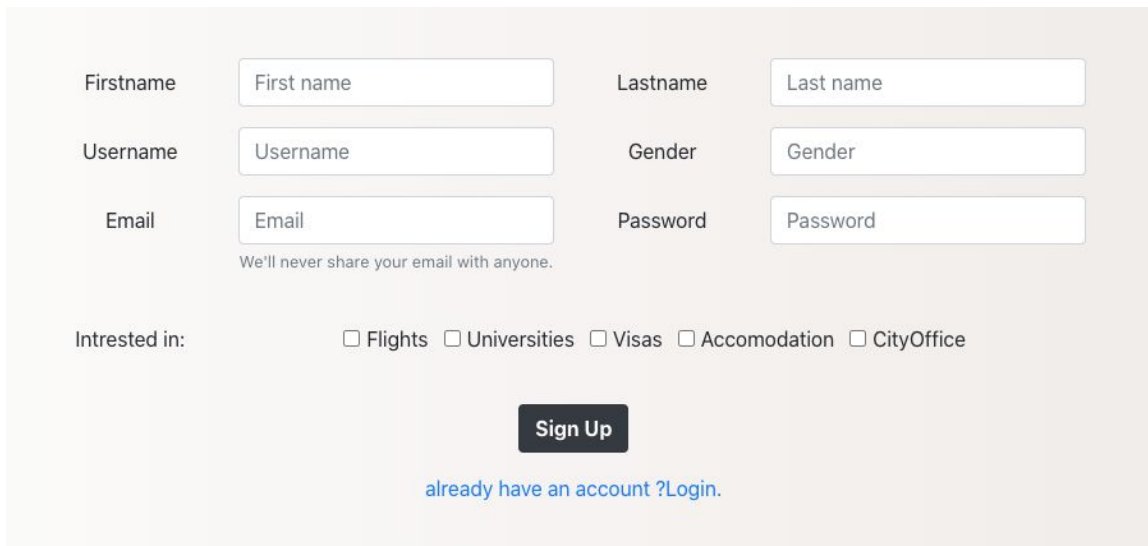
// get the posts searched.
CALL db.index.fulltext.createNodeIndex('searchposts', ['Posts'], ['Person'],
['name', 'pidm', 'title'])

LOAD CSV WITH HEADERS FROM "file:///adcitblog2.csv" AS line
MERGE (p:Person{name:line.name, fullname:line.fullname, idm:line.idm})
MERGE (b:Post{title:line.title, pidm:line.pidm})
MERGE (c:Category{name:line.name})
MERGE (p)-[:Wrote {}]->(b)
MERGE (b)-[:Belongs_To]->(c)
MERGE (p)-[:Have_ineterests_in]->(c)
MERGE (p)-[:Follows]->(p)
MERGE (p)-[:Did_activity_on]->(b)
```

8. Implementation

Common User Interfaces which seems simple but implements use cases that we build behind the curtains.

- ★ Registration - Here the User details and interests are captured and saved in respective databases.



Registration form UI showing fields for Firstname, Lastname, Username, Gender, Email, and Password. It includes a "Sign Up" button and a link for existing users.

Firstname: Lastname:

Username: Gender:

Email: Password:

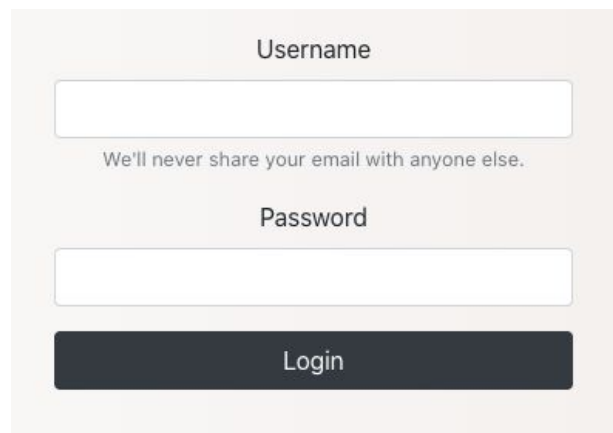
We'll never share your email with anyone.

Interested in: ☐ Flights ☐ Universities ☐ Visas ☐ Accomodation ☐ CityOffice

[Sign Up](#)

[already have an account ?Login.](#)

- ★ Login - After Login User is Redirected to Login, where the credentials, roles assigned to that users and encrypted password matching is done.



Login form UI showing fields for Username and Password, and a "Login" button.

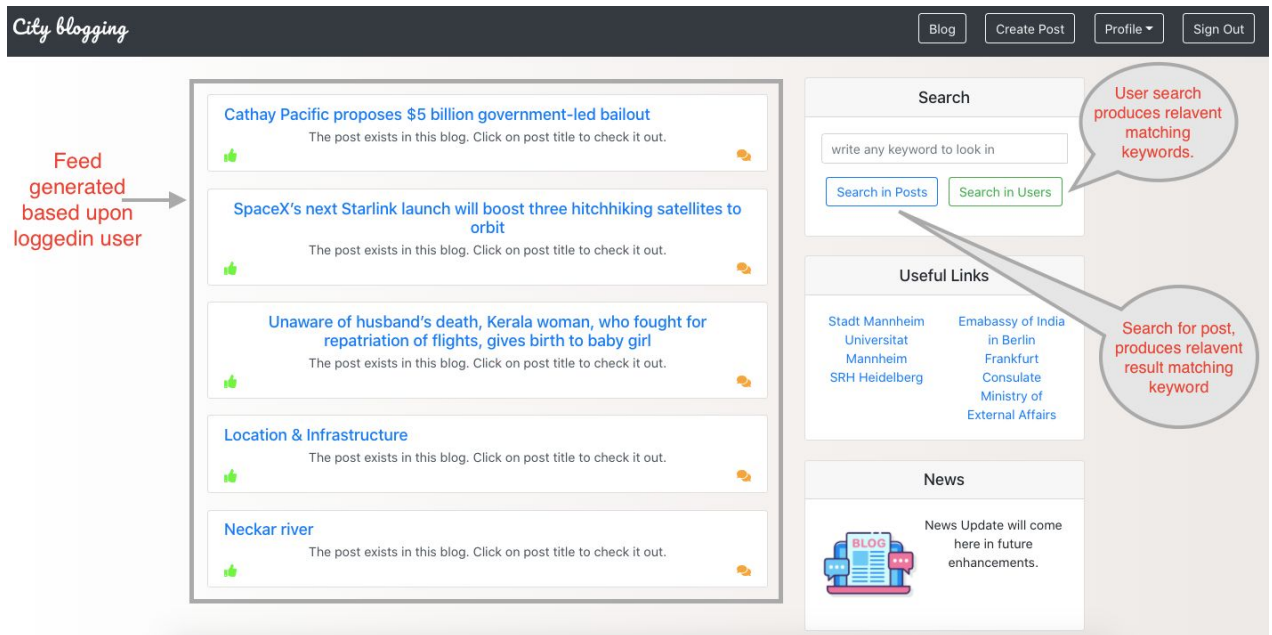
Username:

We'll never share your email with anyone else.

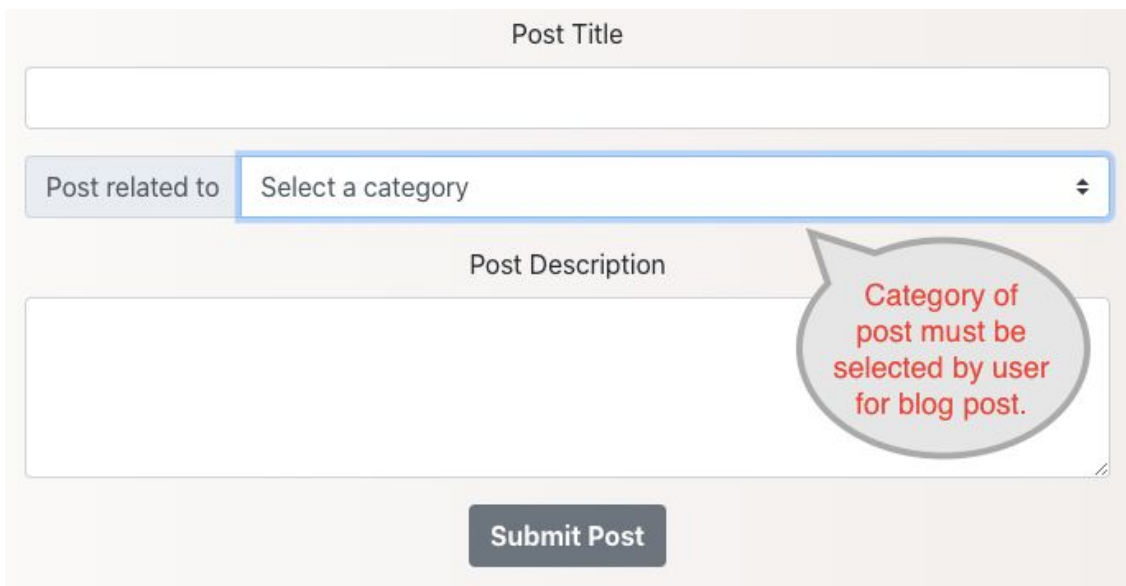
Password:

[Login](#)

- ★ The user sees his Feed which is based upon certain factors such followers and categories he is interested in.



- ★ The user can create their own post which must be related to a category which help to categories the blogs.

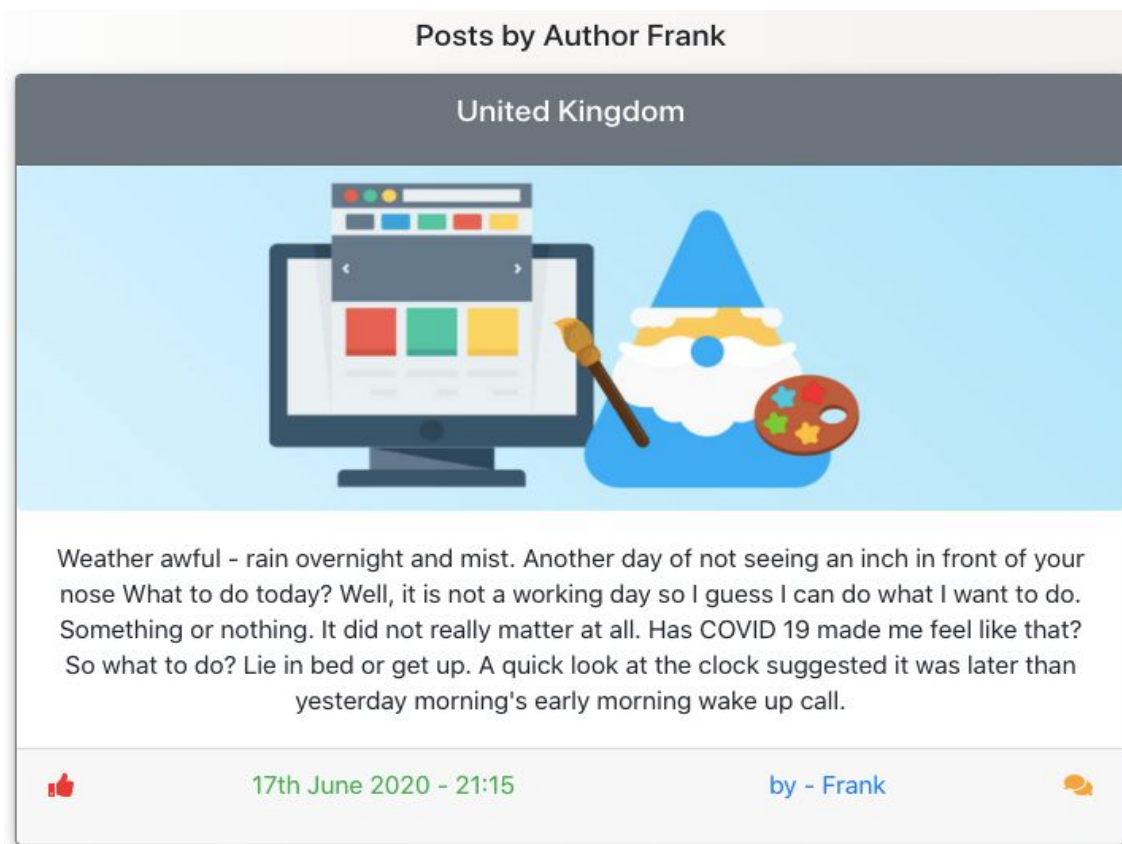
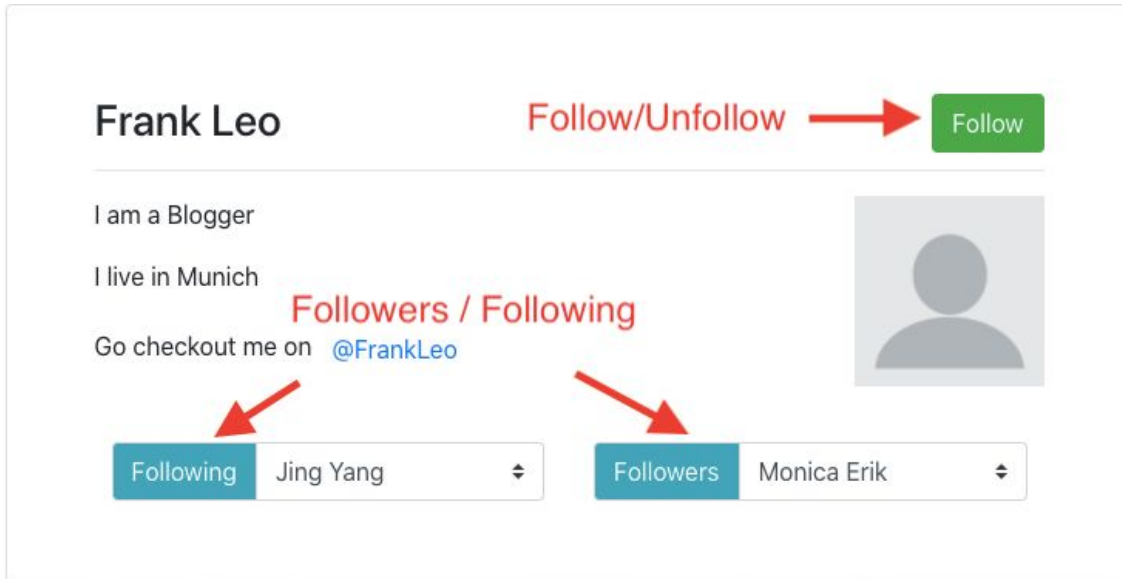


The screenshot shows the 'Create Post' form. It includes the following fields and elements:

- Post Title:** A text input field.
- Post related to:** A dropdown menu with the text 'Select a category'.
- Post Description:** A large text area for the post content.
- Submit Post:** A button to submit the post.

A callout bubble points to the 'Post related to' dropdown menu with the text: 'Category of post must be selected by user for blog post.'

- ★ The user can follow and unfollow another user by going to his/her profile and user's specific post and details



- ★ Apart from users there is one admin who has the list of all users who can also perform the create and delete blogs. Admin also has rights to disable/enable the user profile.

City blogging Admin Blog Create Post Profile Sign Out							
5edd33e4387f21790...	Test	One	test1	test1@gmail.com	true	Disable	Activate
5edd3406387f21790...	Test	Two	test2	test2@gmail.com	true	Disable	Activate
5edf40d3f327d8954...	John	Evans	John	John@gmail.com	true	Disable	Activate
5edf4210f327d8954...	Monica	Lucifer	Lucifer	test4@gmail.com	true	Disable	Activate
5edf4224f327d8954...	Max	Muller	Max	max@gmail.com	true	Disable	Activate
5edf426bf327d8954...	Donald	Duck	Donald	Donald@gmail.com	true	Disable	Activate
5edf4280f327d8954...	Narendra	Mohan	Narendra	Narendra@gmail.com	true	Disable	Activate
5edf4299f327d8954...	Frank	Leo	Frank	Frank@gmail.com	true	Disable	Activate
5edf42bdf327d8954...	Jing	Yang	Jing Yang	Jing@gmail.com	true	Disable	Activate
5edf42d3f327d8954...	Dinesh	Chugtai	Dinesh	Dinesh@gmail.com	false	Disable	Activate
Previous		Page 2 of 5		10 rows		Next	

9. Evaluation

What was achieved?

While doing the Project we gained insights into NoSQL Databases and their relevance to the industry. The part where most of the energy went was Data Modelling, though some changes were necessary when developing the use cases.

The main goal that was kept during the design was scalability, consistency, security and efficiency. The City Blogging project revolves around a social media application so these qualities were decided upon and marked as Must-Haves in the MoSCoW prioritization phase.

Having all the user's data and the posts in MongoDB ensures safety and scalability. The group immediately agreed to have Redis for session management which ensures the fast Read operations and reduces the load of the server for handling sessions. The smart part of our application is implemented through Neo4j which opens up many different doors to several good use cases. Catering a user with a feed and search related to itself was achieved through Neo4j. With the combination of all of the optimized, efficient queries, and integration of polyglot persistence with backend and frontend, an encouraging application was developed.

What was missed?

The Should-Have and Could-Have which included features like monetization, image handling and using Redis as a cache were missed. Generating a distinct keyword log of daily searches through Redis and using it for target marketing (monetization) was also missed.

10. Bibliography

Here is the list of the links and resources used for making this project. -

- [Neo4j Documentation](#)
- [Redis](#)
- [MongoDB Documentation](#)
- [v5.9.18: API docs](#)
- [Express 4.x - API Reference](#)
- [Diagrams - <https://app.lucidchart.com>, \[Arrow Tool\]\(#\)](#)
- [Docker Documentation](#)
- <https://neo4j.com/docs/cypher-manual/3.5/schema/index/#schema-index-fulltext-search>
- [JavaScript](#)
- [React – A JavaScript library for building user interfaces](#)
- [Book - O'Reilly Graph Databases by Emil Eifrem, Ian Robinson, Jim Webber](#)