RBAC and NORP Structured Project Report: For Real Time Systems (CS 6235)

Namyata Cheduri
ncheduri3@gatech.edu

1 INTRODUCTION

NORP is a data platform used to streamline the visualization and analysis of non- profit data from varied data sources. It aims to refine the collection and visualization of such data to enable better insight generation. Data is collected from a combination of public and private sources such as IRS990, Census data, and Face data. NORP aims to provide a one-stop solution to manage and analyze data sources covering domains of healthcare, education, and childcare through two segments: the NORP website which enhances the process of trend recognition by rendering holistic visualizations of data and the underlying data integration submodule helps in data's journey from acquisition, processing, cleaning, quality control and identifying key statistical information from it.

1.1 Motivation

Identity access and management play an integral role in any online software. For a platform such as NORP which deals with personally identifiable data, having strong access control measures in place is a big requirement. While there are existing features such as email verification, strong password suggestion, I aimed to create a fine-grained access control mechanism which can be controlled via a set of APIs.

1.2 Deliverables

In the limited time that I had during the semester, I was able to implement the following features:

- Role Based Access Controls (RBAC)
- Password Encryption
- APIs to manage RBAC and Database designs (APIs on Roles and Permissions)

2 DESIGN AND IMPLEMENTATION OVERVIEW

While authentication establishes the user's identity, authorization determines the level of access that each user has to the data or resource based on specified policies(permissions) but also the actions the user can take on these resources. Authentication has been implemented on a scaled level by my peers, whereas I have implemented a basic session based login to support my RBAC functionality.

2.1 Database Tables Design

3 tables have been added – Role, Permission and RolePermission. Sequelize is used as Object Relational Mapping tool to connect express and MySQL.

- A user can be assigned a single role leading to a 1:1 relationship from user to role. This is represented by "belongsTo" association of Sequelize.
- A role on the other hand can be assigned to multiple users. Thus the association Role - "hasMany" – Users is added. This is analoguous to 1:many relationship from Role to User.
- Finally, Roles can have multiple permissions in them and a permission can be present in multiple roles. This many to many relationship is represented by "belongsToMany" association of sequelize which on initialization and sync creates an intermediary table RolePermission to cater to this permission. RolePermission stores the mapping from role_id to permission_id.

Further an additional attribute "created_by" is added to all the resources to track the user who created the resource. This although not completely implemented, will help in future to scope the views for a given user. For example, a user can be allowed to delete resources only when the user themselves have created it.

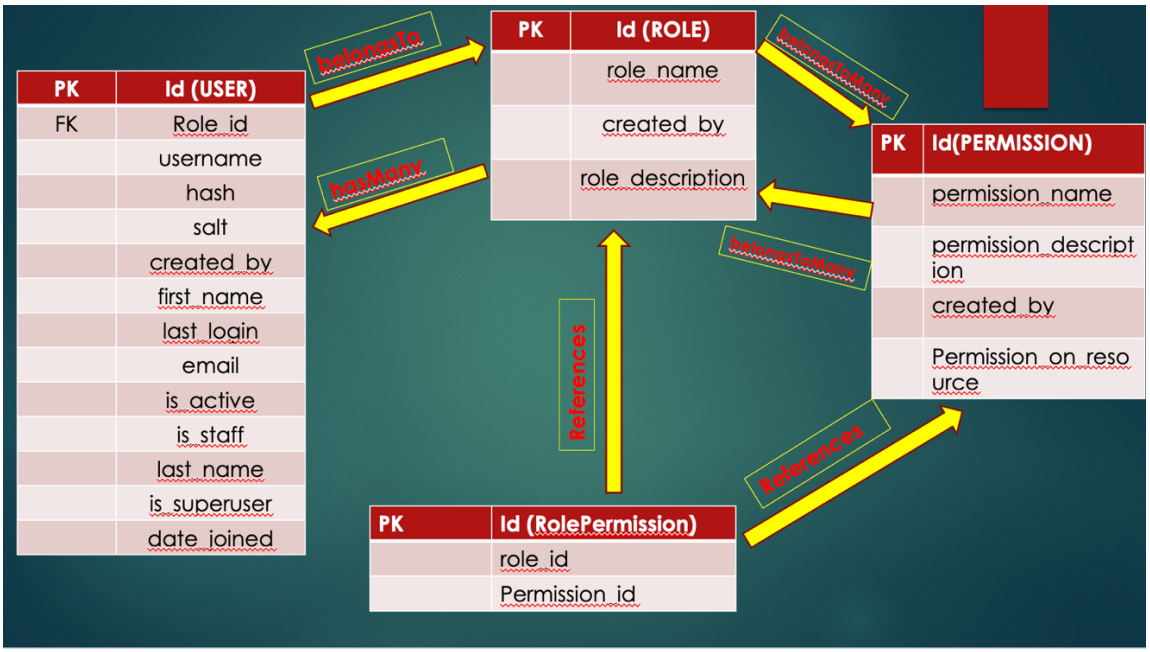


Figure -- FK = Foreign key, PK = Primary Key.

```
class Role extends Model {

    /**
 * Helper method for defining associations.
 * This method is not a part of Sequelize lifecycle.
 * The `models/index` file will call this method automatically.
 */
    static associate (models) {
        // define association here
        Role.hasMany(models.User, {
            foreignKey: 'role_id',
            as: 'users'
          });
        Role.belongsToMany(models.Permission, {through: 'RolePermission', foreignKey: 'role_id', as: 'permissions'})
    }

}
```

Figure -- ASSOCIATIONS AMONG THE RESOURCES.

```
class Role extends Model {

   /**
 * Helper method for defining associations.
 * This method is not a part of Sequelize lifecycle.
 * The `models/index` file will call this method automatically.
 */
   static associate (models) {
      // define association here
      Role.hasMany(models.User, {
         foreignKey: 'role_id',
         as: 'users'
       });
      Role.belongsToMany(models.Permission, {through: 'RolePermission', foreignKey: 'role_id', as: 'permissions'})
   }

}
```

Figure -- ASSOCIATIONS AMONG THE RESOURCES.

```
class Permission extends Model {

   static associate (models) {
      // define association here

      Permission.belongsToMany(models.Role, {through: 'RolePermission', foreignKey: 'permission_id', as: 'roles'})
   }

}
```

Figure -- ASSOCIATIONS AMONG THE RESOURCES.

2.2 API Endpoints Design

CRUD endpoints on permissions and roles has been implemented. This will allow the admin/user to create custom roles with and permissions. Since the permissions are decoupled for every resource (view-graph, view-people etc), fine grained and independent permissions can be assigned to users without compromising the access. The API details are listed in the appendices.

2.3 Nonfunctional Requirement (Password Encryption)

The User table in the in memory MySQL database was storing the password on registration of a new user along with the username. It puts the application on high risk against attacks such as SQL injection, privilege escalation, denial of service etc. Encrypting the password before storing in the database provides a layer of security. The password is hashed and salted following which the hash and salt are stored in the database with the username. Should the user login, the newly entered password is hashed with the stored salt and compared against the stored hash to establish authenticity of the user. Multiple iterations of hashing (10000) and hashing algorithm of SHA-512 is used. Code Snippets are shown in the figure.

The functions are isolated with their responsibilities into a utils file and can be reused in future.

```
function generatePassword(password) {
    var salt = crypto.randomBytes(32).toString('hex');
    //10000 iterations and 64 byte length hash and then convert to hexadecimal string
    var pwdHash = crypto.pbkdf2Sync(password, salt, 10000, 64, 'sha512').toString('hex');
    return {
        salt: salt,
        hash: pwdHash
    }
}

function validatePassword(password, hash, salt) {
    var newHash = crypto.pbkdf2Sync(password, salt, 10000, 64, 'sha512').toString('hex');
    return newHash === hash;
}
```

Figure -- Password hashing functions.

2.4 File Structure and Reusability

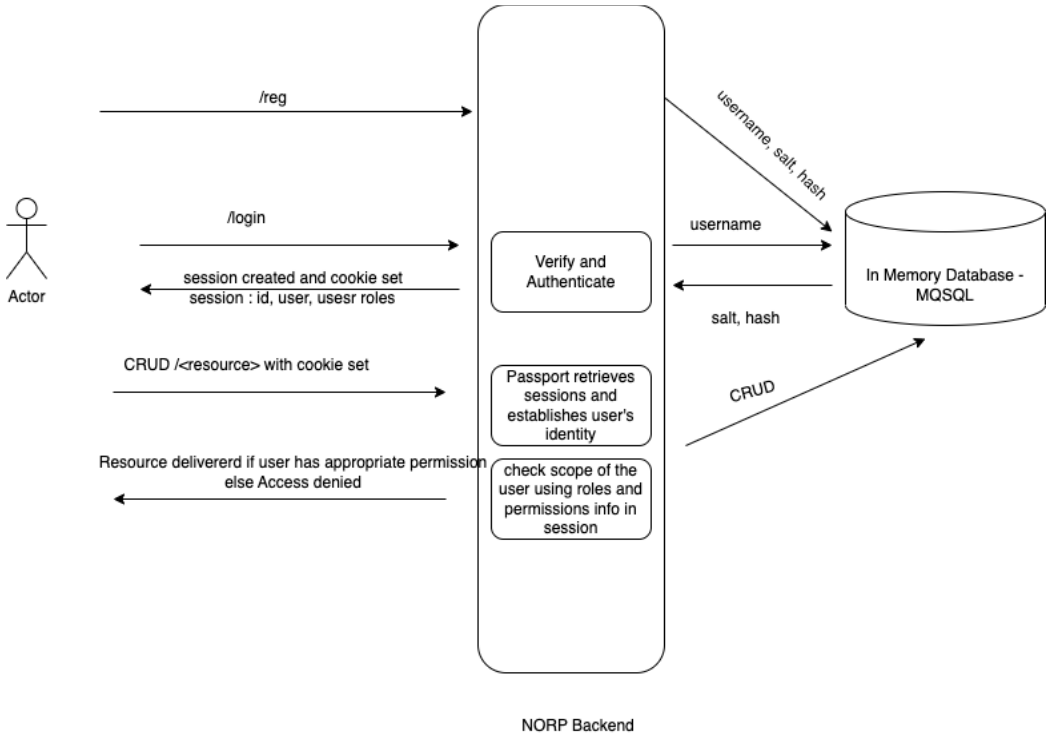
The roles, permissions and rolepermissions models sit in an MVC design pattern and are easy to scale in future. The helper function which checks the scope of the user by verifying permissions is in utils folder. If a new resource is to be added, the helper function can be reused for new resource's authorization requirements. The following figure shows the helper function.

```
module.exports.checkScopeOfUser = async (role_id, permission_name, res) => {
    const permission = await Permission.findOne({
        where: {
            permission_name: permission_name
        }
    })
    if(permission == null || role_id == null) {
        return false
    }
    const rolePermission = await RolePermission.findOne({
        where: {
            role_id: role_id,
            permission_id: permission.id
        }
    })
    if(rolePermission == null)
        return false
    return true


}
```

Figure -- Password hashing functions.

2.5 Sample Workflow to Test



Following is a sample workflow to help understand the RBAC feature.

- The user will not be able to hit any endpoints on resources before logging in.
- The user logs in and passport stores the information about user (salt, hash, roles) in the session and sets the cookie in response headers.
- The user hits the CRUS endpoints with the set cookie. Passport established authenticity with the help of session id in the cookie.
- checkScopeofUser comes into play and verifies if the role assigned to the user making the request, has the necessary permissions to perform the transaction. If the user is authorized, the transactions is performed else a, “Access Denied” response is served. Following figure shows the get user route with the required permission “view-user”.

```
// get User
const getUser = async (req,res) => {
  const data = req.body
  try {
    const isInScope = await checkScopeOfUser(req.user.role_id, "view_user", res)
    if( isInScope == false){
      res.status(403).send("Access denied!")
      return
    }
```

Note : Meta RBAC is implemented by defining an admin role which the only role that allows the user to create, update, delete and view roles, permissions and user resources.

3 LIMITATIONS AND FUTURE WORK

3.1 Difficulties Faced

- New Tech stack : I have never worked on ORM, express and nextJs before. Having to pick up the tech stack while keeping the deliverables on track was a challenge.
- Dependencies among the developers: The login and login via google auth has been implemented by my peers. Since authorization sits on top on authentication, I was dependent on their work. I resolved this by implementing a basic login functionality to support RBAC.
- Fixing functions on the go : The code repository is in its nascent stage and has a huge scope of improvement. A set code standards has to be agreed upon to maintain uniformity. An OpenAPI specification of the endpoints would have been useful.
- Complex interactions among the new endpoints. For example, the current version doesn't invalidate the permission from the roles it has been added to when the permission is deleted by DELETE /permission. Such interactions were difficult to incorporate within the time frame of the project.

3.2 Future Work

- The current UI doesn't have the provision to perform transactions on the resources and relies on APIs. Due to the limited scope of the project, the RBAC as well was implemented on back-end only. In future we can plan

to add a UI page/modal to allow the admin/superuser to create users, custom roles and permissions and authorize users using the permissions.

- Currently the resources are predefined (User, People, Publication, Graph and Videos). In future we can plan to bootstrap the generation of basic permisssions on the newly added resource by providing an API with factory design pattern to create permissions (view-<resource>, edit-<resource> etc).
- As the code repository grows, it will be difficult to track the updates and feature addition. This might lead to regression bugs. Implementing and maintaining a good test coverage with unit tests and workflow tests will help the code repo to be clean and reliable.
- Graph database will be more efficient to represent and query on the relationships in RBAC.

4 REFERENCES

1. Chatekar, S. (2019, May 8). *Visualising complex apis using API map*. Medium. Retrieved December 3, 2022, from https://medium.com/@suhas_chatekar/visualising-complex-apis-using-api-map-f09f617acb32
2. *What is Data Access Control?* SailPoint. (2021, August 17). Retrieved December 3, 2022, from https://www.sailpoint.com/identity-library/what-is-data-access-control/
3. Bezkoder. (2021, August 17). *Sequelize many-to-many association example - node.js & mysql*. BezKoder. Retrieved December 3, 2022, from https://www.bezkoder.com/sequelize-associate-many-to-many/
4. J., D. (2022, May 1). *Authentication role permission API using Node Express mysql*. Djamware.com. Retrieved December 3, 2022, from https://www.djamware.com/post/6158fee27523f53fb5c1b2f4/authentication-role-permission-api-using-node-express-mysql
5. Reales, A. (2022, April 25). *How to use router-level middleware in express.js using typescript*. Become A Better Programmer. Retrieved December 3, 2022, from https://www.becomebetterprogrammer.com/router-level-middleware-express-typescript/

5 APPENDICES

- The postman collection for APIs on all the resources is included in the zip file. This will help while on boarding new developers in future.
- Branch Name for the RBAC code – ncheduri3. Link → https://github.gatech.edu/NORP/norp-next/tree/ncheduri3.
- The basic workflow, postman collection is sufficient for a user to reproduce and test the flow.
- The API endpoints and their descriptions :

RESOURCE NAME: ROLE		
GET	**/role → Get all roles /role/:id → Get role by role_id**	**view_role**
PUT	**/role → updates and existing role**	**update_role**
POST	**/role → creates a new role /role:id → Add a permission to role**	**create_role, update_role**
DELETE	**/role → Deleted the given role id (body)**	**delete_role**

RESOURCE NAME: USER		
GET	/user → get user by username in the body	view_user
PUT	/user→ updates and existing user	update_user
POST	/user → creates a new user	create_user
DELETE	/user → Deleted the given user name (body)	delete_user

RESOURCE NAME: PERMISSION		
GET	/permission → Get all permissions	view_permission
PUT	/permission → updates and existing permission	update_permission
POST	/permission → creates a new permission	create_permission
DELETE	/permission → Deleted the given permission id (body)	delete_permission

RESOURCE NAME: GRAPH		
GET	/graph → get all graphs	view_graph
PUT	/graph→ updates and existing graph	update_graph
POST	/graph → creates a new graph	create_graph
DELETE	/graph → Deleted the given id(body)	delete_graph

RESOURCE NAME: VIDEO		
GET	/video → get all videos	view_video
PUT	/video→ updates and existing video	update_video
POST	/video → creates a new video	create_video
DELETE	/video → Deleted the given id(body)	delete_video

RESOURCE NAME: PEOPLE		
GET	/perople → get all people	view_people
PUT	/perople→ updates and existing person record given username	update_people
POST	/perople → creates a new person record	create_people
DELETE	/perople → Deleted the given id(body)	delete_people
RESOURCE NAME: PUBLICATION		
GET	/publication → get all publications	view_publication
PUT	/publication→ updates and existing publication	update_publication
POST	/publication → creates a new publication	create_publication
DELETE	/publication → Deleted the given id(body)	delete_publication