RBAC Vision and Justification

URStreamSight

Raymond Knorr, Avery Cameron, Noah Rowbotham

Version: 1 2021-04-04

Table of Contents

What is RBAC?

Role Based Access Control (RBAC) is defined as “Access control based on user roles. Role permissions ... typically reflect the permissions needed to perform defined functions within an organization. A given role may apply to a single individual or to several individuals.” (*RBAC - Glossary*). Based on this definition, we can understand that individuals in an organization are given specific roles that allow them programmatic access to pieces of an organization’ functionality. A typical example is of an Administrator being given full access to any and all possible actions, while a contractor or lower-level employee would only be given cherry-picked access. In the world of programming and APIs, RBAC is used to decide whether a given individual is able to perform the function associated with a given endpoint.

Authentication vs. Authorization

These two words are often used interchangeably, however in the context of programming, they mean related but separate things. Authentication is the act of determining who a user is, while authorization is concerned with whether the user should be allowed to perform the action they wish to perform. RBAC is concerned with the authorization of the user, but to authorize a user, we first need to know who they are. This is where authentication and JSON Web Tokens come in.

Authentication: JWT Tokens

A JSON Web Token (JWT) is a signed, encoded string that is sent with every request to an API. It is signed by an identity issuer typically when a user logs in, and verifies that the user is who they say they are in the scope of the organization. The act of signing in, being verified by an issuer, and receiving a JWT is what we refer to as authentication.

JWTs contain several recommended fields, and several that can be customized per use case. One of the most important fields however is the `exp` field, which designates when the token will expire. If a user attempts to use this token after expiry, it will not be accepted by the API. This field is normally set to less than an hour after the JWT is issued. This guarantees that a new token is given to a user per sitting.

In our API, we are using AWS Cognito to verify the identity of our users. This is an identity management solution provided by AWS to use with other AWS services. We provide them with the user's login information, and they return a token that we can use between our front-end and API to describe the user. The most important custom field that we receive from our token is the user's `username`.

Authorization: RBAC

The token that our front-end logic receives from AWS Cognito is saved and used per-request to our API. Once the token reaches the API, we can use the user's `username` field to query the database and find their roles. From there, we can determine whether they should be allowed to access the endpoint they are trying to reach.

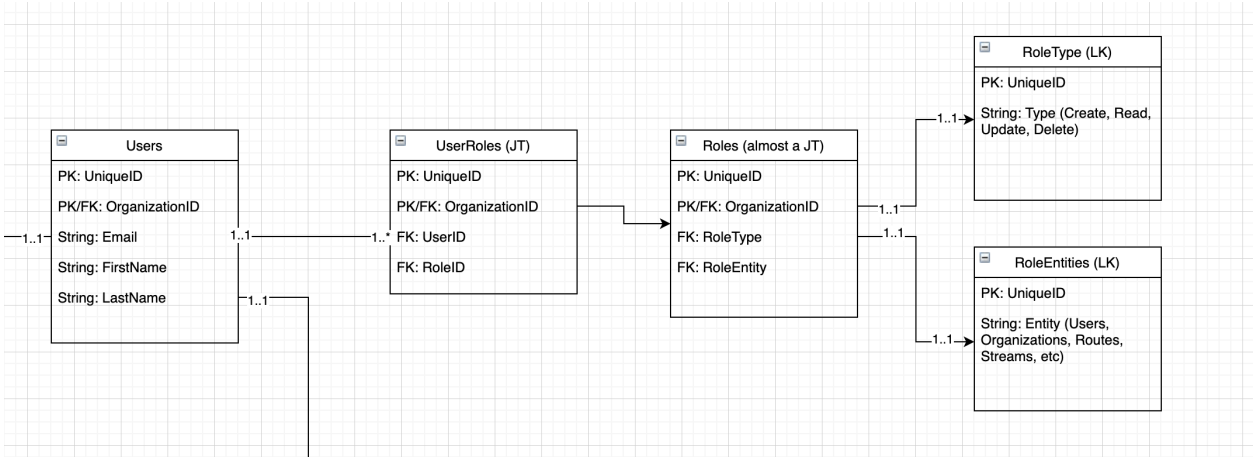
Our Solution

Database

As previously mentioned, we receive the user's token in our API via the `Authorization` header in their request. After verifying that the token is valid and not expired, we extract the username, and use it to conditionally grant access to any given endpoint.

The way that we store the roles given to users is in our MySQL database. Our definition of a role is a combination between the database entity that the user is trying to access, and the type of operation that they are trying to perform. For example, if a user wants to hit one of our API endpoints to receive a list of all ServiceEvents, they will need to possess the role `read:serviceEvents`. The four operation types correspond to the four types of database operations, namely `Create`, `Read`, `Update`, and `Delete`. The possible entities that a role can grant access to are instead `Organizations`, `Routes`, `ServiceAddresses`, `ServiceEvents`, `Users`, `Reviews`, `UserActions`, and `Roles`. The values associated with the type and entity are stored in the two tables `RoleType`, and `RoleEntities` respectively. Below is an image of the relevant table schema's used to perform RBAC.

Figure 1: RBAC Table Schema



Besides the two tables previously mentioned, the best starting point is the `Users` table. We store various pieces of data relating to a user, but notably no reference to roles. The `Roles` table contains an `id`, a reference to its allowed database operation, and the type of entity it grants access for. The `Roles` table also contains an `OrganizationID`, which guarantees that a user is only allowed to perform operations on entities that exist within the scope of their organization. Again, it should be noted that the `Roles` table has no reference to `Users`.

The relationship between `Users` and `Roles` is many-to-many. This means that there could be multiple user's who all possess the same role in an organization, and each user could possess multiple roles. For this reason, their relationship is stored in its own `UserRoles` join table, so that neither `Users` nor `Roles` need to contain a list of the other corresponding entities.

Our solution allows for very granular levels of access via roles. Right now, no UI exists to create a user with roles, and the adding of the user to the database would have to occur manually. This includes adding the user, and any `UserRoles` that they should possess. A single user can have as many as the 22 roles that currently exist in our database. Per user, each of these roles would have to have a corresponding entry into the `UserRoles` table.

Code

In our API, the user's roles are obtained from their username in their token. This username must come from AWS Cognito, and is therefore secure. We query a user's roles via joins on the `UserRoles`, `Roles`, `RoleType`, and `RoleEntities` tables. From the information gathered, we can develop a list of strings corresponding to their roles, given in the form

“{type}:{entity}”. It is in this form that we can check against the roles that are necessary to access a given endpoint.

Our API is written in TypeScript, using a framework called NestJS. This allows us to implement a piece of functionality called a `Guard`. A `Guard` is a function that is accessed before the code of the actual API endpoint is reached. It can decide to pass execution to the endpoint, or terminate the request, depending on it’s internal logic. In our case, we have a `RolesGuard` that compares the user’s roles against the necessary roles specified for a given endpoint, and allows access to the endpoint only if the user has *at least one* of the required roles. The way this looks in practice is included below:

Code Snippet 1: RBAC for Organization POST Endpoint

```
@Post()
@Roles(RoleEnum.CREATE_ORGANIZATIONS)
async createOrganization(@Body() body: Organization): Promise<Organization> {
   ...
}
```

The endpoint being accessed is a POST endpoint to create an organization, and the role necessary is the descriptive `create:organizations`. This string value is obfuscated behind an enum that we use to avoid magic strings in our code. The `@Roles(...)` decorator specifies the required roles. Identical decorators exist above every other endpoint in our app, with the corresponding type and entity.

An important note is that since our endpoints currently only perform rudimentary operations on data, each endpoint has only one required role. Another example of an endpoint protected with the `RolesGuard` would be as follows:

Code Snippet 2: RBAC for ServiceEvents GET Endpoint

```
@Get('serviceAddresses/:serviceAddressId/recycling')
@Roles(RoleEnum.READ_SERVICE_EVENTS)
async findAllByAddress(
  @Param('orgId') orgId: string,
  @Param('serviceAddressId') serviceAddressId: string,
  @Query('offset') offset: number
): Promise<BulkServiceEventsDto> {
  ...
```

```
}
```

Similar to the `Organizations` endpoint, we see that a `read:serviceAddresses` role is necessary to perform a GET on the ServiceAddresses endpoint.

When a user does not have the correct role to access an endpoint, the standard HTTP status code of `403 Forbidden` is given as a response. This relays to our front-end that the given user is not allowed to access whatever they just tried to. Here is a response from one of our endpoints when the user does not have the required role:

Code Snippet 3: Forbidden Request

```
{
   "statusCode": 403,
   "message": "Forbidden resource",
   "error": "Forbidden"
}
```

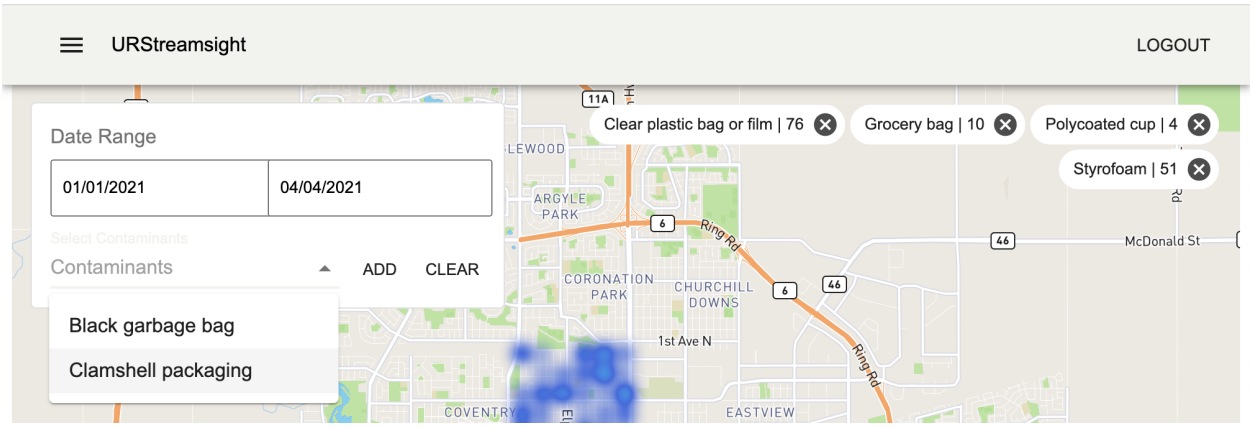
Overall, RBAC allows us to grant very granular access to our API and business logic of our application.

Potential Improvements

Front-End UI

As previously mentioned, our current solution has very granular access, but is not set up to easily add users. One main improvement that is planned for the front-end after our project finishes is a user management page. This page would allow users to easily create other users with *up to the same level as access as the creating user*. This makes sure that a non-admin user will not be able to create an admin user for example. The UI for this page would probably involve a sort of multi-select similar to our front-end mapping page. By default, all the available roles would be selected, and the user would be able to delete, or add back any roles they see fit. Here is an image of the relevant parts of our map page, notice the multi-select field on the left, and the selected instances on the right:

Image 1: UI for URStreamSight Mapping Page



Role Groups

The second important addition that could be made is the concept of role groups. This would involve another many-to-many relationship in our database between roles and role groups. Each role group contains multiple roles, and a single role can be part of many role groups. A role-group could have a name, such as “Admin” or “Read-Only”, which would allow a more user-friendly option of selecting which roles a new user can have. This layout also looks more similar to what someone would expect Role Based Access Control to look like.

References

RBAC - Glossary. (n.d.). Retrieved April 04, 2021, from https://csrc.nist.gov/glossary/term/RBAC