Permissions Flow Explanation

This document explains the step-by-step flow of the `PermissionsGuard` when a user interacts with a controller endpoint, such as "Get Blogs". The explanation details what happens when a user has a single scoped permission (`:public` , `:own` , `:all`) versus what happens when they possess multiple permissions simultaneously.

1. Context and Setup

Let's assume the **"Get Blogs"** controller route is decorated with multiple required permissions to allow users with different access levels to hit the same endpoint:

```
@Get()
@RequirePermissions('blogs:read:all', 'blogs:read:own', 'blogs:read:public')
getBlogs(@Req() request: Request) {
  // the 'request.scope' will dictate what data to fetch in the service
}
```

The `PermissionsGuard` evaluates these requirements against the user's actual permissions (extracted from the JWT token or headers).

---

2. The Weighting System

The guard determines the best access level a user has using a "weighting" system:

- **`all`** : weight `3` (Highest Privilege)
- **`own`** : weight `2`
- **`public`** : weight `1` (Lowest Privilege)

When the guard loops through the route's requirements, it checks which ones the user *actually has*, and keeps track of the **highest weight** (the `bestScope`).

---

3. Scenarios: Single Permission

Here is what happens if the user only has **one** of the permitted scopes.

Scenario A: User ONLY has `blogs:read:public`

1. The guard iterates through our required list (`blogs:read:all` , `blogs:read:own` , `blogs:read:public`).
2. It sees the user *does not* have `blogs:read:all` .
3. It sees the user *does not* have `blogs:read:own` .
4. It sees the user **DOES** have `blogs:read:public` .
5. `hasAccess` becomes `true` .
6. The `currentScope` is extracted as `public` (weight: 1).
7. Because it's the first match, `bestScope` is set to `public` .
8. **Result**: The user is granted access, and `request.scope` is set to `'public'` . The service will only return public blogs.

Scenario B: User ONLY has `blogs:read:own`

1. The guard sees the user *does not* have `blogs:read:all` .
2. The guard sees the user **DOES** have `blogs:read:own` .
3. `hasAccess` becomes `true` .
4. The requested scope is extracted as `own` (weight: 2).
5. `bestScope` is set to `own` .
6. The guard checks `blogs:read:public` , user doesn't have it.
7. **Result**: The user is granted access, and `request.scope` is set to `'own'` . The service will return only the user's personal blogs.

Scenario C: User ONLY has `blogs:read:all`

1. The guard sees the user **DOES** have `blogs:read:all` .
2. `hasAccess` becomes `true` .
3. Extracted scope: `all` (weight: 3).
4. `bestScope` becomes `all` .
5. Guard continues, user doesn't have `own` or `public` .
6. **Result**: Access granted, `request.scope` is set to `'all'` . The service will return all blogs in the system.

---

4. Scenario: Multiple Permissions

What happens if a user's role grants them **multiple** permissions for the same resource? For example, an Admin might somehow end up with both `blogs:read:own` AND `blogs:read:all` in their token.

Scenario D: User has Both `blogs:read:all` AND `blogs:read:own`

1. The guard loops through the requirements.
2. **First iteration (`blogs:read:all`)**:
   - The user HAS this permission.
   - `hasAccess = true` .
   - `currentScope = all` (weight: 3).
   - `bestScope` becomes `all` (since it's currently empty).
3. **Second iteration (`blogs:read:own`)**:
   - The user ALSO HAS this permission.
   - `hasAccess = true` (already true).
   - `currentScope = own` (weight: 2).
   - **The Conflict Resolution**: The guard checks if the weight of the new scope (`own` -> 2) is GREATER than the weight of `bestScope` (`all` -> 3).
   - Since `2` is NOT greater than `3` , the `bestScope` **remains `all`** .
4. **Third iteration (`blogs:read:public`)**:
   - User does not have this.
5. **Result**: The guard perfectly resolves the conflict by prioritizing the highest privilege. Access is granted, and `request.scope` is set to `'all'` .

---

5. Summary of the Flow

1. **Extract**: Get required route permissions and user's actual permissions.
2. **Evaluate**: Loop through the *required* permissions. If the user holds the matching permission, flag access as `true` .
3. **Weigh**: Extract the scope (`public` , `own` , `all`) from the matching permission. Compare its weight to any scope found previously.
4. **Elevate**: Keep only the Highest Weight so far (`bestScope`).
5. **Inject**: Inject the overriding `bestScope` into `request.scope` .
6. **Pass/Fail**: Pass the request to the controller, or throw a `ForbiddenException` (403) if `hasAccess` stayed false.