Overview of the System:

The **RBAC-based User Authentication and Authorization System** is designed to manage users, their roles, and the permissions associated with these roles while ensuring secure access to resources. The system leverages **Role-Based Access Control (RBAC)** to manage what users can and cannot access based on their assigned roles. Here's a brief breakdown of the core components and concepts involved:

1. **Roles**:
    - Roles define the type of user in the system. Each role has a set of permissions attached to it. In this case, three roles were defined:
        - **Admin**: Full access to the system (view, edit, and delete data).
        - **User**: Limited access, only to view data.
        - **Moderator**: Can view and edit data.
2. **Permissions**:
    - Permissions represent actions that a role can perform. Permissions are assigned to each role to determine what users with that role can do in the system. For example, the Admin role has all three permissions (view, edit, delete), while the User role only has permission to view data.
3. **Authentication**:
    - **JWT (JSON Web Token)** or **OAuth** manages user authentication. This ensures that only authenticated users can access protected resources. Authentication is handled via the login API endpoint where users submit their credentials (username and password), which are verified, and if valid, they are provided with a JWT token for further requests.
4. **Authorization**:
    - Once authenticated, **authorization** determines which resources a user can access. This is controlled using RBAC, where each user's role is checked before granting access to resources. The system uses the user's role to determine whether they can access specific views or endpoints.
    - For example, an Admin has full permissions (view, edit, delete), while a User can only view data.
5. **Role-Based Access Control (RBAC)**:
    - RBAC was implemented by creating **roles** and **permissions models** and associating them with many-to-many relationships. Each user is assigned a role, which dictates the permissions granted to the user.
    - The permissions are checked using the `has_permission()` method, which checks the user's assigned role for the required authorization.

Implementation Details:

1. **Models**:
    - **Role Model**: A model to define different user roles such as Admin, User, and Moderator.
    - **Permission Model**: A model to define various permissions (e.g., can view, can edit, can delete).
    - **RolePermissions Model**: A model to associate roles with permissions, using a many-to-many relationship.
    - **User Model**: A custom user model inherited from Django's `AbstractUser` model, which includes an additional `role` field to assign a user a role.
2. **Role and Permission Assignment**:
    - Permissions were created for each role and then assigned accordingly. For example:
        - Admin role: full access with all permissions.
        - User role: only view permissions.
        - Moderator role: can view and edit data.
3. **User Authentication & JWT Token Generation**:
    - The **login** API takes the user's credentials (username and password) and verifies them.
    - If valid, a **JWT token** is returned, which is required for making further requests to access protected resources.
    - The **JWT token** is stored and sent with each subsequent request to verify the user's identity.
4. **Authorization Check**:
    - The `has_permission()` method was added to the custom User model, which checks if a user's assigned role has the specified permission.
5. **Deployment**:
    - The application was deployed on **Heroku**, with SQLite used as the database. The system was tested locally before the deployment.
    - **Heroku** is a cloud platform that provides free hosting for applications, and it was used to deploy the system.

Sample Workflow:

1. **User Registration**: A user registers by providing a username and password. The user is assigned a role (Admin, User, Moderator).

2. **Login**: The user logs in with their credentials. A JWT token is issued, and the user can use this token to authenticate future requests.

3. **Permission Check**: When a user attempts to access a resource, the system checks if the user has the required permissions based on their role. If the user has the necessary permissions, they can access the resource; otherwise, access is denied.

4. **Role Check**: When accessing a resource, the role of the user determines the level of access they have:

    - Admins can access and modify everything.
    - Moderators can view and edit data but cannot delete it.
    - Users can only view data.

Next Steps:

1. **Testing**:

    - Test the system using various users with different roles and permissions.
    - Ensure that users can only access what their role allows.
2. **Documentation**:

    - Create further documentation on the API endpoints, role assignments, and how to use the system effectively.

Relevant Concepts:

1. **RBAC**:

    - **Role-Based Access Control** is a method used to restrict access to resources based on the roles assigned to users. This helps maintain the principle of least privilege.
2. **JWT Authentication**:

    - **JSON Web Token (JWT)** is used for securely transmitting information between the client and server. It is widely used for managing user sessions in web applications.
3. **Django ORM**:

    - Django's **Object-Relational Mapping (ORM)** is used to interact with the database. It allows us to define models and query them in an object-oriented way.

This project demonstrates a secure and scalable approach to managing users, their roles, and permissions in a web application, and it ensures that each user can access only the resources they are authorized to use.