MONGODB ROLE-BASED ACCESS CONTROL (RBAC) EXPLAINED

MongoDB access control enables database administrators to secure MongoDB instances by enforcing user authentication. MongoDB supports multiple authentication methods and grants access through role-based authorization.

Roles are the foundation blocks in MongoDB, providing user isolation for a great degree of security and manageability.

In this article, we'll take a look at the most common roles in MongoDB. Then, we'll explore role management by illustrating many common role-related functions.

(This article is part of our MongoDB Guide. Use the right-hand menu to navigate.)

How MongoDB RBAC works

A user can be assigned one or more roles, and the scope of user access to the database system is determined by those assigned roles. Users have no access to the system outside the designated roles.

Importantly, MongoDB access control is not enabled by default; you have to enable it through the **security.authorization** setting.

When that setting is enabled, users need to authorize themselves before interacting with the database. User privileges can include a specific resource (database, collection, cluster) and actions permitted on that resource.

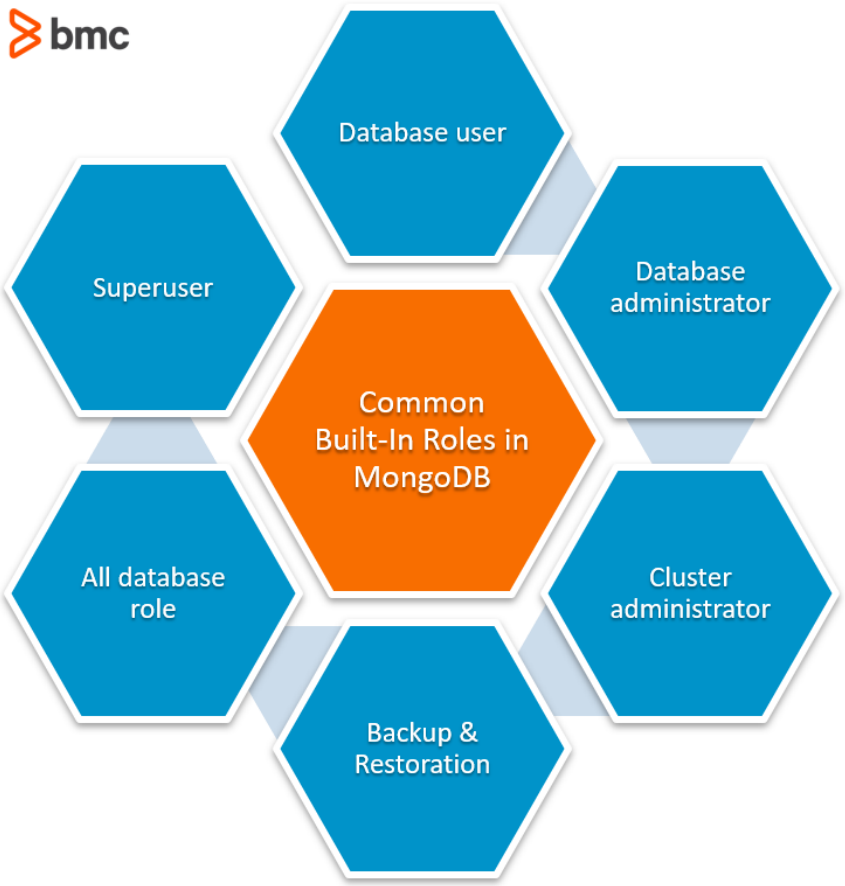
A **role** grants permission to perform particular actions on a specific resource. A single **user** account can consist of multiple roles. Roles can be assigned:

- At the time of user creation
- When updating the roles of existing users

There are two types of Roles in MongoDB:

- **Built-In Roles.** MongoDB provides built-in roles to offer a set of privileges that are commonly needed in a database system.
- **User-Defined Roles.** If built-in roles do not provide all the expected privileges, database administrators can define custom roles using the **createRole** method. Those roles are called User-Defined roles.

Next, let's look at the variety of roles in more details.



Built-in Roles

In this section, we'll go through the most common built-in roles of MongoDB.

Database user roles

Database user roles are normal user roles that are useful in regular database interactions.

Role	Description
read	Read all non-system collections and the system.js collection
readWrite	Both Read and Write functionality on non-system collections and the system.js collection

Database administration roles

These are roles that are used to carry out administrative operations on databases.

Role	Description
dbAdmin	Perform administrative tasks such as indexing and gathering statistics, but cannot manage users or roles
userAdmin	Provides the ability to create and modify roles and users of a specific database
dbOwner	This is the owner of the database who can perform any action. It is equal to combining all the roles mentioned above: readWrite, dbAdmin, and userAdmin roles

Cluster admin roles

These roles enable users to interact and administrate MongoDB clusters.

Role	Description
clusterManager	Enables management and monitoring functionality on the cluster. Provides access to config and local databases used in sharding and replication
clusterMonitor	Provide read-only access to MongoDB monitoring tools such as Cloud Manager or Ops Manager monitoring agent
hostManager	Provides the ability to monitor and manage individual servers
clusterAdmin	This role includes the highest number of cluster administrative privileges allowing a user to do virtually anything. This functionality is equal to the combination of clusterManager, clusterMonitor, hostManager roles, and dropDatabase action.

Backup & restoration roles

These are the roles that are required for backup and restoring data in a MongoDB instance. They can only be assigned with the admin database.

Role	Description
backup	Provides the necessary privileges to backup data. This role is required for MongoDB Cloud Manager and Ops Manager, backup agents, and the monogdump utility.
restore	Provides the privileges to carry out restoration functions

All database roles

These are database roles that provide privileges to interact with all databases, excluding local and config databases.

Role	Description
readAnyDatabase	Read any database
readWriteAnyDatabase	Provides read and write privileges to all databases
userAdminAnyDatabase	Create and Modify users and roles across all databases
dbAdminAnyDatabase	Perform database administrative functions on all databases

Superuser roles

MongoDB can provide either direct or indirect system-wide superuser access. The following roles grant superuser privileges scoped to a specified database or databases.

- dbOwner
- userAdmin
- userAdminAnyDatabase

The true superuser role is the **root** role, which provides systemwide privileges for all functions and resources.

For a detailed explanation of all the user roles and privileges, please refer to the official MongoDB documentation.

User-Defined Roles

In situations where built-in roles are unable to provide the necessary privileges covering the scope of the access requirements or to restrict the scope and actions of a user, you can define custom User-Defined roles.

MongoDB Role Management provides the necessary methods to create and manage user-defined roles. The most commonly used methods for user-defined role creation are shown in the following table. (A complete list of methods can be found here.)

Method	Description
db.createRole()	Create a role and its privileges
db.updateRole()	Update the user-defined role
db.dropRole()	Delete a user-defined role
db.grantPrivilegesToRole()	Assigns new privileges to a role
db.revokePrivilegesFromRole()	Removes privileges from a role

MongoDB role management

In this section, Let's discuss how to create and modify roles within a MongoDB instance.

Roles are defined using the following syntax:

```
roles:
```

Assigning user roles at user creation

First, create a user with read and write access to a specific database (supermarket) using the **createUser** method with roles parameter.

```
db.createUser(
{
user: "harry",
pwd: "test123",
```

```
roles:
}
)
```

Result:

```
> db.createUser(
...     {
...         user: "harry",
...         pwd: "test123",
...         roles: [
...             {
...                 role: "readWrite",
...                 db: "supermarket"
...             }
...         ]
...     }
... )
Successfully added user: {
        "user" : "harry",
        "roles" : [
                {
                        "role" : "readWrite",
                        "db" : "supermarket"
                }
        ]
}
>
```

Then, create an administrative user to the supermarket database with roles granting read access to all other databases and the backup and restore functions.

```
db.createUser(
{
user: "harryadmin",
pwd: "admin12345",
roles:
}
)
```

Result:

```
> db.createUser(
...     {
...         user: "harryadmin",
...         pwd: "admin12345",
...         roles: [
...             {
...                 role: "dbOwner", db: "supermarket"
...             },
...             {
...                 role: "readAnyDatabase", db: "admin"
...             },
...             {
...                 role: "backup", db: "admin"
...             },
...             {
...                 role: "restore", db: "admin"
...             }
...         ]
...     }
... )
Successfully added user: {
        "user" : "harryadmin",
        "roles" : [
                {
                        "role" : "dbOwner",
                        "db" : "supermarket"
                },
                {
                        "role" : "readAnyDatabase",
                        "db" : "admin"
                },
                {
                        "role" : "backup",
                        "db" : "admin"
                },
                {
                        "role" : "restore",
                        "db" : "admin"
                }
        ]
}
>
```

As shown in the above code-block, you can create users with multiple roles providing access to a wide range of functionality within the database.

Retrieving role information

Using the **getRole** method, users can obtain information about a specific role. The below code shows how to get the details of the **readWriteAnyDatabase** role from the admin database.

```
db.getRole("readWriteAnyDatabase")
```

Result:

```
> db.getRole("readWriteAnyDatabase")
{
        "role" : "readWriteAnyDatabase",
        "db" : "admin",
        "isBuiltin" : true,
        "roles" : [ ],
        "inheritedRoles" : [ ]
}
>
```

To obtain the privileges associated with that role, use the **showPrivileges** option by setting its value as 'true'. This can also be used in the **getUser** method.

```
db.getRole("readWriteAnyDatabase", { showPrivileges: true})
```

Result:

```
> db.getRole("readWriteAnyDatabase", { showPrivileges: true})
{
        "role" : "readWriteAnyDatabase",
        "db" : "admin",
        "isBuiltin" : true,
        "roles" : [ ],
        "inheritedRoles" : [ ],
        "privileges" : [
                {
                        "resource" : {
                                "db" : "",
                                "collection" : ""
                        },
                        "actions" : [
                                "changeStream",
                                "collStats",
                                "convertToCapped",
                                "createCollection",
                                "createIndex",
                                "dbHash",
                                "dbStats",
                                "dropCollection",
                                "dropIndex",
                                "emptycapped",
                                "find",
                                "insert",
                                "killCursors",
                                "listCollections",
                                "listIndexes",
                                "planCacheRead",
                                "remove",
                                "renameCollectionSameDB",
                                "update"
                        ]
                },
```

Identifying assigned user roles

The **getUser** method enables you to identify the roles assigned to a specific user by using the following syntax:

```
db.getUser("<Username>")
```

In the below example, you can retrieve the user details of the users, “harry” and “harryadmin” with their roles using the **getUser** method.

```
db.getUser("harry")
```

```
db.getUser("harryadmin")
```

Result:

```
> db.getUser("harry")
{
        "_id" : "admin.harry",
        "userId" : UUID("765e61d8-a508-45f8-810f-aed2fcb712fe"),
        "user" : "harry",
        "db" : "admin",
        "roles" : [
                {
                        "role" : "readWrite",
                        "db" : "supermarket"
                }
        ],
        "mechanisms" : [
                "SCRAM-SHA-1",
                "SCRAM-SHA-256"
        ]
}
> db.getUser("harryadmin")
{
        "_id" : "admin.harryadmin",
        "userId" : UUID("675bbb3b-6084-479a-b860-f0cb22ca6bf1"),
        "user" : "harryadmin",
        "db" : "admin",
        "roles" : [
                {
                        "role" : "dbOwner",
                        "db" : "supermarket"
                },
                {
                        "role" : "readAnyDatabase",
                        "db" : "admin"
                },
                {
                        "role" : "backup",
                        "db" : "admin"
                },
                {
                        "role" : "restore",
                        "db" : "admin"
                }
        ],
        "mechanisms" : [
                "SCRAM-SHA-1",
                "SCRAM-SHA-256"
        ]
}
>
```

Granting & revoking user roles

Using the **grantRolesToUser** and **revokeRolesFromUser** methods, you can modify the roles assigned to existing users. These methods use the following syntax:

```
db.<grantRolesToUser | revokeRolesFromUser> (
"<Username>",

)
```

The below example shows how to grant user roles. You will add a new role to the user “harry” providing the necessary privileges to act as the database admin (**dbAdmin**) to gather statistics from the “vehicles” table.

```
db.grantRolesToUser(
"harry",

)
```

Result:

```
> db.grantRolesToUser(
...     "harry",
...     [
...         { role: "dbAdmin", db: "vehicles" }
...     ]
... )
> db.getUser("harry")
{
        "_id" : "admin.harry",
        "userId" : UUID("765e61d8-a508-45f8-810f-aed2fcb712fe"),
        "user" : "harry",
        "db" : "admin",
        "roles" : [
                {
                        "role" : "dbAdmin",
                        "db" : "vehicles"
                },
                {
                        "role" : "readWrite",
                        "db" : "supermarket"
                }
        ],
        "mechanisms" : [
                "SCRAM-SHA-1",
                "SCRAM-SHA-256"
        ]
}
>
```

Now you have added a new role to the user, “harry”. So, let's remove that newly granted role using the **revokeRolesFromUser** method.

```
db.revokeRolesFromUser(
"harry",

)
```

Result:

```
> db.revokeRolesFromUser(
...     "harry",
...     [
...         { role: "dbAdmin", db: "vehicles" }
...     ]
... )
> db.getUser("harry")
{
        "_id" : "admin.harry",
        "userId" : UUID("765e61d8-a508-45f8-810f-aed2fcb712fe"),
        "user" : "harry",
        "db" : "admin",
        "roles" : [
                {
                        "role" : "readWrite",
                        "db" : "supermarket"
                }
        ],
        "mechanisms" : [
                "SCRAM-SHA-1",
                "SCRAM-SHA-256"
        ]
}
>
```

Creating user-defined roles

Using the **createRole** method, you can create a new role according to your needs. The basic structure of the createRole method is shown below.

Note: Using the roles parameter within the createRole method allows you to inherit the privileges of other roles within the user-defined role. If no other privileges are required, you need to define an empty roles parameter when creating a user-defined role.

```
db.createRole(
{
role: "<RoleName>",
privileges:
}
],
roles:
}
)
```

To associate a user-defined role to all databases or collections, you can specify the resources with empty double quotes, as shown below.

```
resource: { db: "", collection: ""}
```

In the next code block, you will create a role that limits a user's access to a specific database collection (inventory collection of supermarket database). It limits the user actions to find and update commands without inheriting any other privileges.

```
db.createRole(
{
role: "inventoryeditor",
```

```
privileges:
}
],
roles:
}
)
```

Result:

```
> db.createRole(
...     {
...         role: "inventoryeditor",
...         privileges: [
...             {
...                 resource: { db: "supermarket", collection: "inventory"},
...                 actions: [ "find", "update" ]
...             }
...         ],
...         roles: [ ]
...      }
... )
{
        "role" : "inventoryeditor",
        "privileges" : [
                {
                        "resource" : {
                                "db" : "supermarket",
                                "collection" : "inventory"
                        },
                        "actions" : [
                                "find",
                                "update"
                        ]
                }
        ],
        "roles" : [ ]
}
>
```

We can identify the user-defined roles using the **isBuiltin** parameter. The **inventoryeditor** role has false for the that parameter, indicating it as a non-built-in role.

```
> db.getRoles()
[
        {
                "role" : "inventoryeditor",
                "db" : "admin",
                "isBuiltin" : false,
                "roles" : [ ],
                "inheritedRoles" : [ ]
        }
]
>
```

Now you know how to create a standalone user-defined role. Next, you will create a user-defined role that inherits some privileges from another built-in role.

In the below code block, you will create an **inventorymanager** role with all the CURD privileges and inherit the privileges from the **userAdmin** role.

```
db.createRole(
{
```

```
role: "inventorymanager",
privileges:
}
],
roles:
}
)
```

Result:

```
> db.createRole(
...     {
...         role: "inventorymanager",
...         privileges: [
...             {
...                 resource: { db: "supermarket", collection: "inventory"},
...                 actions: [ "find", "update", "insert", "remove" ]
...             }
...         ],
...         roles: [
...             { role: "userAdmin", db: "supermarket" }
...         ]
...      }
... )
{
        "role" : "inventorymanager",
        "privileges" : [
                {
                        "resource" : {
                                "db" : "supermarket",
                                "collection" : "inventory"
                        },
                        "actions" : [
                                "find",
                                "update",
                                "insert",
                                "remove"
                        ]
                }
        ],
        "roles" : [
                {
                        "role" : "userAdmin",
                        "db" : "supermarket"
                }
        ]
}
>
```

If you check the **inventorymanager** role using the **getRole** method, it will display the details of the role, including the inherited permissions.

```
> db.getRole("inventorymanager")
{
        "role" : "inventorymanager",
        "db" : "admin",
        "isBuiltin" : false,
        "roles" : [
                {
                        "role" : "userAdmin",
                        "db" : "supermarket"
                }
        ],
        "inheritedRoles" : [
                {
                        "role" : "userAdmin",
                        "db" : "supermarket"
                }
        ]
}
>
```

Assigning user-defined roles to users

You can assign user-defined roles to a new user or update the roles of an existing user in the same way you do it with a built-in role. Let's create a new user with the **inventorymanager** role assigned and update an existing user with the **inventoryeditor** role in the following examples.

Creating a new user:

```
db.createUser(
{
user: "managerjerry",
pwd: "manager123",
roles:
}
)
```

Result:

```
...     {
...         user: "managerjerry",
...         pwd: "manager123",
...         roles: [
...             {
...                 role: "inventorymanager",
...                 db: "admin"
...             }
...         ]
...     }
... )
Successfully added user: {
        "user" : "managerjerry",
        "roles" : [
                {
                        "role" : "inventorymanager",
                        "db" : "admin"
                }
        ]
}
```

Granting new roles:

```
db.grantRolesToUser(
"repairmanager",
```

)

Result:

```
> db.grantRolesToUser(
...     "repairmanager",
...     [
...         { role: "inventoryeditor", db: "admin" }
...     ]
... )
> db.getUser("repairmanager")
{
        "_id" : "admin.repairmanager",
        "userId" : UUID("a921e1b2-5e02-4b8e-9afd-e9186478655f")
        "user" : "repairmanager",
        "db" : "admin",
        "roles" : [
                {
                        "role" : "dbOwner",
                        "db" : "vehicles"
                },
                {
                        "role" : "inventoryeditor",
                        "db" : "admin"
                }
        ],
        "mechanisms" : [
                "SCRAM-SHA-1",
                "SCRAM-SHA-256"
        ]
}
```

Updating & deleting user-defined roles

You can update the user-defined roles using the **updateRole** method and delete the roles using the **dropRole** method.

Let's update the **inventoryeditor** role with an additional privilege to create documents within the specified collection, in the below example.

```
db.updateRole(
"inventoryeditor",
{
privileges:
}
],
roles:
}
)
```

Result:

```
> db.updateRole(
...     "inventoryeditor",
...     {
...         privileges: [
...             {
...                 resource: { db: "supermarket", collection: "inventory"},
...                 actions: [ "find", "update", "insert" ]
...             }
...         ],
...         roles: [ ]
...     }
... )
> db.getRole("inventoryeditor", { showPrivileges: true })
{
        "role" : "inventoryeditor",
        "db" : "admin",
        "isBuiltin" : false,
        "roles" : [ ],
        "inheritedRoles" : [ ],
        "privileges" : [
                {
                        "resource" : {
                                "db" : "supermarket",
                                "collection" : "inventory"
                        },
                        "actions" : [
                                "find",
                                "insert",
                                "update"
                        ]
                }
        ],
        "inheritedPrivileges" : [
                {
                        "resource" : {
                                "db" : "supermarket",
                                "collection" : "inventory"
                        },
                        "actions" : [
                                "find",
                                "insert",
                                "update"
                        ]
                }
        ]
}
```

The most important thing to keep in mind when updating roles is that it will completely replace old values in the privileges and roles arrays. Therefore, you need to provide the complete arrays with the modifications when updating a user-defined role.

The **dropRole** method has a single functionality to remove a user-defined role. You can remove the inventoryeditor role from the database, as shown below.

```
db.dropRole("inventoryeditor")
```

Result:

```
> db.dropRole("inventoryeditor")
true
> db.getRole("inventoryeditor", { showPrivileges: true })
null
>
```

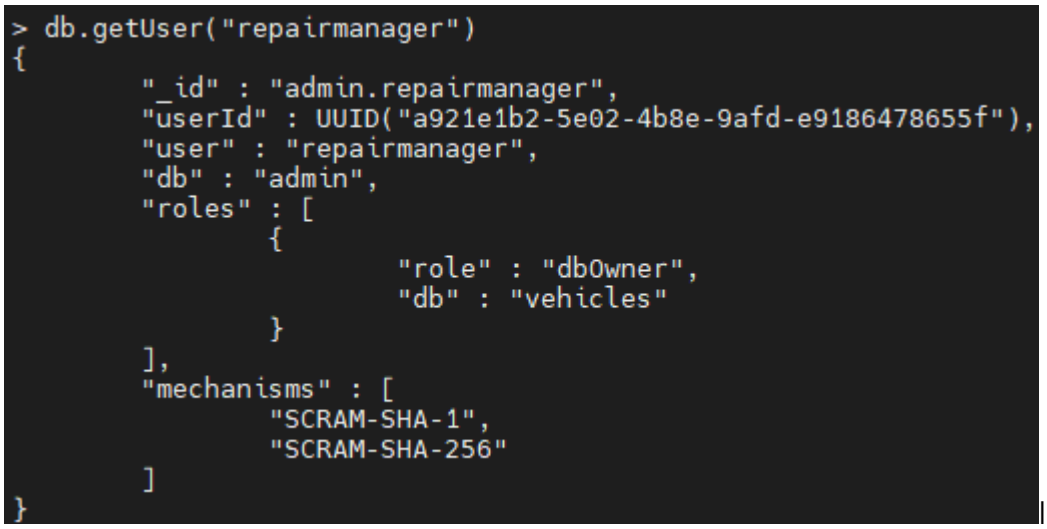
The above screenshot indicates that the role deletion was successful, and if we try to retrieve the deleted role, it will return a null value.

When a role is deleted, it will affect all the users associated with the deleted role, revoking all the privileges granted by that user-defined role.

Earlier, you assigned the **inventoryeditor** role to the user **repairmanager**. Now, if you check that user details, you can notice that the “inventoryeditor” role is no longer assigned to the “repairmanager” user.

```
db.getUser("repairmanager")
```

Result:



In situations where only a single user-defined role is assigned to a user when the said role is deleted, it will result in an empty roles array, effectively denying all user privileges.

You can check this scenario by deleting the **inventorymanager** role and getting the **managerjerry** user details. This will result in an empty roles array for the user managerjerry, demonstrated below by comparing the user details before and after the role deletion.

```
db.getUser("managerjerry")
db.dropRole("inventorymanager")
db.getUser("managerjerry")
```

Result:

```
> db.getUser("managerjerry")
{
        "_id" : "admin.managerjerry",
        "userId" : UUID("c7c7c85e-438c-419a-acfa-dd55501ebc57"),
        "user" : "managerjerry",
        "db" : "admin",
        "roles" : [
                {
                        "role" : "inventorymanager",
                        "db" : "admin"
                }
        ],
        "mechanisms" : [
                "SCRAM-SHA-1",
                "SCRAM-SHA-256"
        ]
}
> db.dropRole("inventorymanager")
true
> db.getUser("managerjerry")
{
        "_id" : "admin.managerjerry",
        "userId" : UUID("c7c7c85e-438c-419a-acfa-dd55501ebc57"),
        "user" : "managerjerry",
        "db" : "admin",
        "roles" : [ ],
        "mechanisms" : [
                "SCRAM-SHA-1",
                "SCRAM-SHA-256"
        ]
}
```

Thus, it is paramount that you check if the roles are associated with any users and how it will affect the user's functional scope *before* deleting any user-defined roles.

As a final remark, the **dropRole** method can only delete user-defined roles. If you try to delete a built-in role using that method, it will result in an error, as shown below.

```
> db.dropRole("readWrite")
uncaught exception: Error: readWrite@admin is a built-in role and cannot be modified. :
_getErrorWithCode@src/mongo/shell/utils.js:25:13
DB.prototype.dropRole@src/mongo/shell/db.js:1702:11
@(shell):1:1
>
```

That concludes this comprehensive look at MongoDB roles and role management.

Related reading

- BMC Machine Learning & Big Data Blog
- 23 Common MongoDB Operators & How To Use Them, part of our MongoDB Guide
- BMC Guides, series of articles on topics like Apache Cassandra & Spark, AWS, Machine Learning, and Data Visualization
- Data Storage Explained: Data Lake vs Warehouse vs Database