Role-Based Access Control (RBAC) Documentation

This document outlines the role-based access control system implemented in the eLearning platform. The system defines three distinct roles (ADMIN, FACULTY, and STUDENT) with varying levels of permissions across all application resources.

Access Restrictions

User Collection

Role	Create	Read	Update	Delete
ADMIN	Full access to create any user	Full access to view all users	Full access to update any user	Can delete users with no enrollments
FACULTY	No creation rights	Limited to viewing own profile	Can update own profile only	Cannot delete users
STUDENT	No creation rights	Limited to viewing own profile	Can update own profile only	Cannot delete users

Enrollment Collection

Role	Create	Read	Update	Delete
ADMIN	Full access to create enrollments	Full access to view all enrollments	Full access to update enrollments	Full access to delete enrollments
FACULTY	No creation rights	Can view enrollments for their courses	No update rights	Cannot delete enrollments
STUDENT	No creation rights	Limited to viewing own enrollments	No update rights	Cannot delete enrollments

Course Collection

Role	Create	Read	Update	Delete
ADMIN	Full access to create courses	Full access to view all courses	Full access to update courses	Can delete courses with no enrollments
FACULTY	Can create courses as instructor	Can view all courses (emphasized own)	Limited to updating own courses	Cannot delete courses
STUDENT	No creation rights	Limited to enrolled courses	No update rights	Cannot delete courses

Assignment Collection

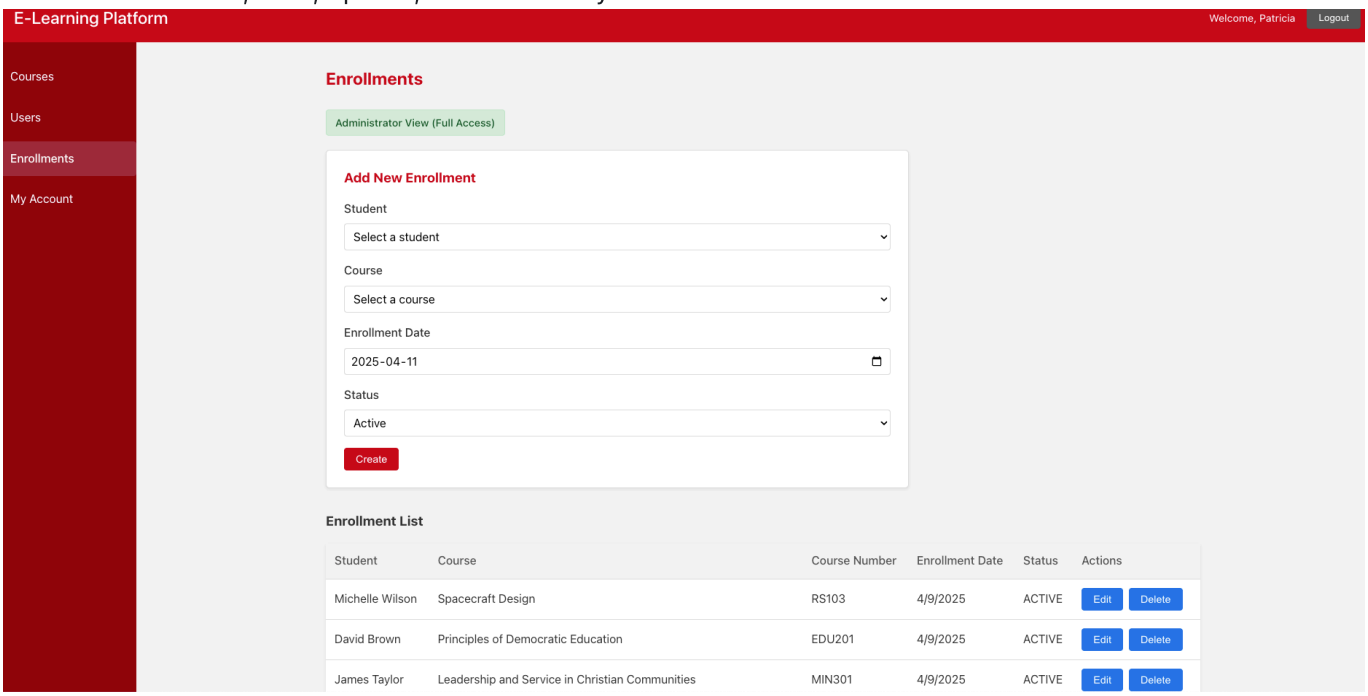
Role	Create	Read	Update	Delete
ADMIN	Full access to create assignments	Full access to view all assignments	Full access to update assignments	Full access to delete assignments
FACULTY	Limited to own courses	Can view assignments for own courses	Limited to updating own assignments	Limited to deleting own assignments
STUDENT	No creation rights	Limited to enrolled course assignments	No update rights	Cannot delete assignments

Quiz Collection

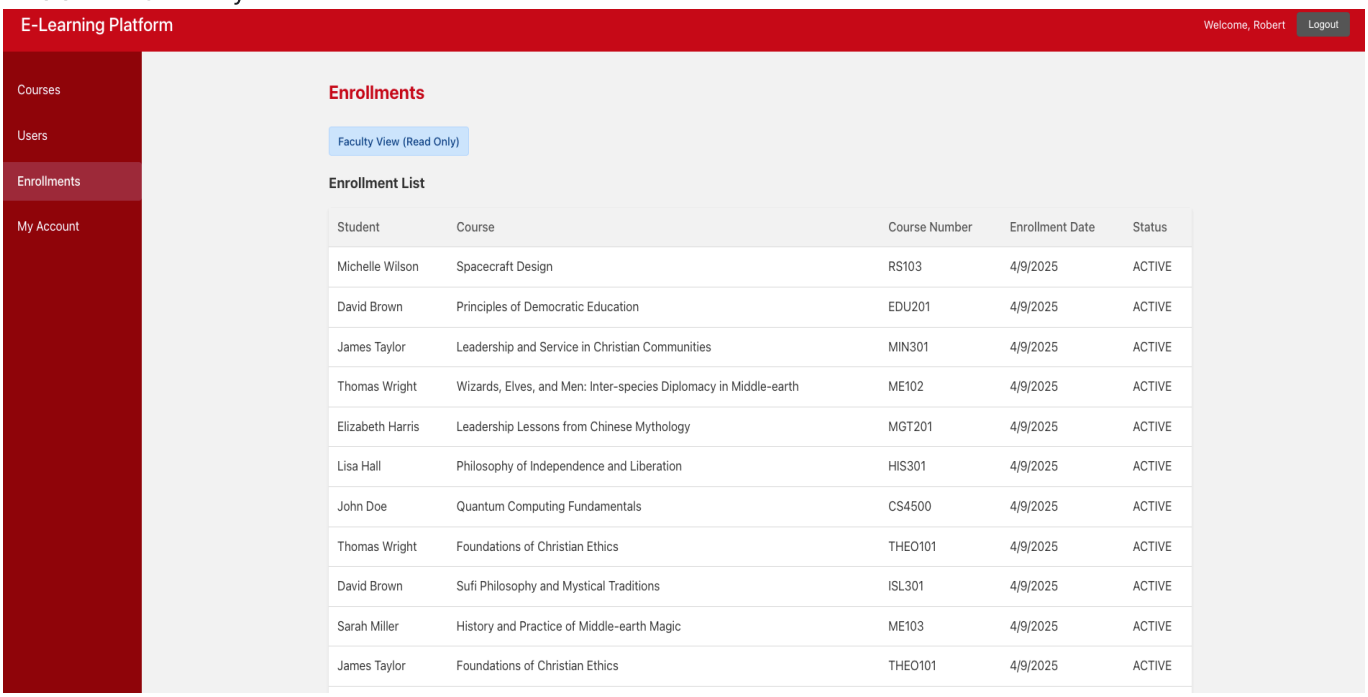
Role	Create	Read	Update	Delete
ADMIN	Full access to create quizzes	Full access to view all quizzes	Full access to update quizzes	Full access to delete quizzes
FACULTY	Limited to own courses	Can view quizzes for own courses	Limited to updating own quizzes	Limited to deleting own quizzes
STUDENT	No creation rights	Limited to enrolled course quizzes	No update rights	Cannot delete quizzes

Example of Role-Based Access Control:

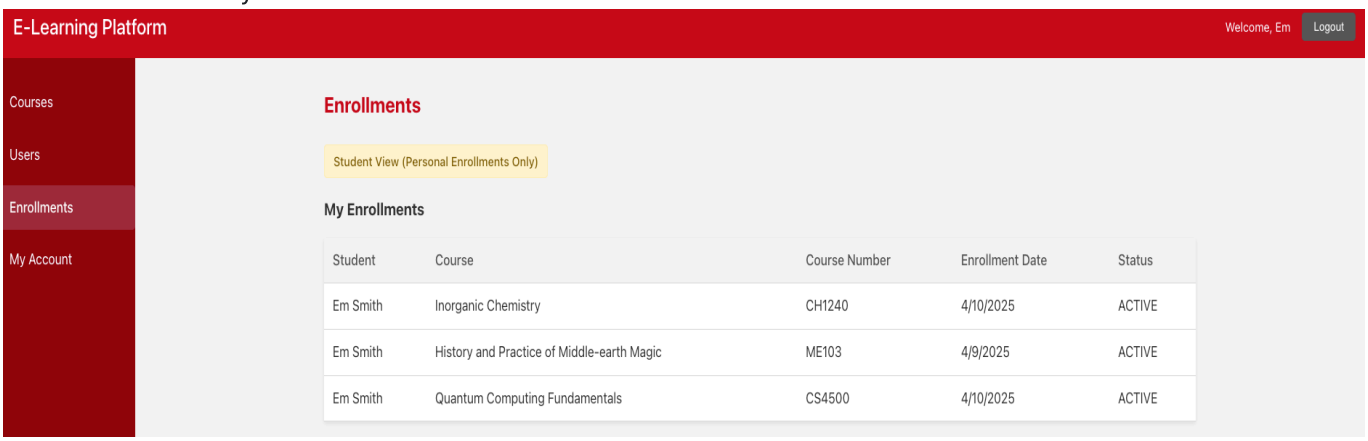
- **ADMIN**: Can create, read, update, and delete any enrollment.



- **FACULTY**: Can only view enrollments



- **STUDENT**: Can only view their own enrollments.



Design Description

RBAC Implementation

The eLearning platform implements RBAC through several coordinated mechanisms:

1. **Authentication via JWT**:
    - Users authenticate using username/password
    - Upon successful authentication, a JWT token is issued containing user ID and role
    - This token is included in subsequent API requests
2. **Middleware Layer**:
    - `verifyToken` : Validates JWT tokens and extracts user ID and role
    - `isAdmin` : Ensures only ADMIN users can access a route
    - `isFacultyOrAdmin` : Ensures only FACULTY or ADMIN users can access a route

- `filterEnrollmentsByRole` : Applies role-specific filtering to enrollment data

3. **Role-Based Route Protection**:
   - Each API endpoint is protected with appropriate middleware
   - Data access is filtered based on the user's role and relationship to the resource
4. **Frontend Integration**:
   - UI elements are conditionally rendered based on the user's role
   - Interface displays appropriate indicators of the user's access level
   - Forms and controls are only shown when the user has appropriate permissions

Implementation Examples

Backend Route Protection:

```
// filepath: Example route protection
// Admin-only endpoint
router.post("/api/enrollments", verifyToken, isAdmin, async (req, res) => {
  // Only admins can reach this code
});

// Faculty/Admin endpoint with additional checks
router.put('/api/courses/:id', verifyToken, async (req, res) => {
  // Faculty can only update courses they teach
  if (req.userRole === 'FACULTY' && course.instructor.toString() !== req.userId) {
    return res.status(403).json({ message: 'Access denied' });
  }
  // Implementation continues...
});
```

Frontend Role-Based UI:

```
// filepath: Example UI components
// Role-specific indicators in UI
<div className="role-indicator">
  {currentUser && currentUser.role === 'ADMIN' &&
    <p className="role-badge admin">Administrator View (Full Access)</p>}
  {currentUser && currentUser.role === 'FACULTY' &&
    <p className="role-badge faculty">Faculty View</p>}
  {currentUser && currentUser.role === 'STUDENT' &&
    <p className="role-badge student">Student View (Read Only)</p>}
</div>

// Conditional rendering based on permissions
{canModifyEnrollments && (
  <form onSubmit={handleSubmit} className="form">
    {/* Form elements would be here */}
  </form>
)}
```

Database Schema Support

The user schema includes a role field that supports the RBAC system:

```
// filepath: User schema role definition
role: {
  type: String,
  enum: ["STUDENT", "FACULTY", "ADMIN", "USER"],
  default: "USER",
}
```

Backend and Database-Level Access Enforcement

The application enforces access restrictions at multiple layers beyond just the API routes:

1. **Database Query Filtering**:
    - Queries are dynamically modified based on user role
    - For STUDENT users, queries include filters to only return resources they have access to
    - For FACULTY, queries include instructor-specific filters
    - For ADMIN, no additional filters are applied, enabling full access
2. **Data Access Objects (DAOs)**:
    - Each collection has a dedicated DAO layer
    - DAOs implement role-aware operations that enforce access policies
    - For example, the enrollment DAO enforces that students can only view their own enrollments:

```
// filepath: Example from enrollment DAO
export const findEnrollmentsByUser = async (userId) => {
  try {
    return await Enrollment.find({ user: userId })
      .populate("user", "username firstName lastName email")
      .populate("course", "number name");
  } catch (error) {
    throw new Error(`Error finding enrollments: ${error.message}`);
  }
};
```

3. **Transactional Operations**:
   - Critical operations use MongoDB transactions to maintain data integrity
   - Role-based checks are performed within transaction boundaries
   - Example from enrollment creation:

```
// filepath: Example transaction with role checks
export const createEnrollment = async (enrollmentData) => {
  return await withTransaction(async (session) => {
    // Check for duplicates first
    const existingEnrollment = await Enrollment.findOne({
      user: enrollmentData.user,
      course: enrollmentData.course
    }).session(session);

    if (existingEnrollment) {
      throw new Error("Student is already enrolled in this course");
    }

    // Create enrollment within transaction
    const newEnrollment = new Enrollment(enrollmentData);
    return await newEnrollment.save({ session });
  });
};
```

4. **Foreign Key Constraints**:
   - MongoDB schema design implements reference integrity checks
   - Example: Prevention of user deletion when they have enrollments

```
// filepath: Example deletion constraint check
export const deleteUser = async (id) => {
  try {
    // Check if user has any enrollments
    const enrollments = await Enrollment.find({ user: id });
    if (enrollments.length > 0) {
      throw new Error("Cannot delete user with existing enrollments. Please
remove their enrollments first.");
    }

    // If no enrollments, proceed with deletion
    return await User.findByIdAndDelete(id);
  } catch (error) {
    throw new Error(`Error deleting user: ${error.message}`);
  }
};
```

5. **Compound Indexes for Uniqueness**:
   - Database schema uses compound indexes to enforce business rules
   - Example from enrollment schema to prevent duplicate enrollments:

```
// filepath: Example compound index from enrollment schema
enrollmentSchema.index({ user: 1, course: 1 }, { unique: true });
```

This implementation ensures security by enforcing access controls at multiple layers of the application, from the database to the API and frontend interface.