

## Experiment no :

**Aim:** Implement authentication and user roles with JWT

### Theory:

Authentication and authorization are critical components of modern web applications to ensure data security and controlled access.

#### 1. Authentication:

Authentication is the process of verifying the identity of a user. In this project, it is implemented using **JSON Web Tokens (JWT)**, a secure and stateless mechanism. After a user registers or logs in, the server generates a JWT containing encoded user details (such as `userId` and `role`). This token is then sent to the client and stored (commonly in `localStorage` or cookies). For subsequent requests, the client includes the token in the request headers, enabling the server to validate the identity of the user without maintaining session state.

#### 2. Authorization and User Roles:

Authorization determines what actions an authenticated user is allowed to perform. In this project, two main roles are considered:

- **Admin:** Has complete control, including the ability to delete items posted by any user.
- **Normal User:** Can post and delete only their own items but cannot modify or delete items posted by others.

3. Role-based access control (RBAC) is enforced using middleware that verifies the JWT .

4. Role-based access control (RBAC) is enforced using middleware that verifies both the JWT and the associated user role before granting access to protected routes (e.g., `/dashboard`, `/add`, `/delete`).

#### 5. Protected Routes:

Routes such as **Add Item**, **Dashboard**, and **Delete** are accessible only to authenticated users. Middleware functions validate the JWT to ensure requests come from legitimate users. Additional checks (e.g., verifying item ownership) ensure that users cannot tamper with or delete resources they do not own, unless they hold the **Admin** role.

#### 6. Real-life Analogy (RLS Policy Equivalent):

- **verifyOwner:** Ensures that only the creator of an item can view or delete it.
- **verifyAdmin:** Grants elevated privileges to Admin users, allowing them to perform actions beyond normal users.

This structure provides both **security** and **fair usage policies** in a multi-user environment, protecting user data while granting admins the ability to moderate content.

## Codes -

### 1. User Model (models/User.js) - Crucial for RBAC

```
// models/User.js
const mongoose = require("mongoose");
const bcrypt = require("bcryptjs");
```

```
const userSchema = new mongoose.Schema({
  username: { type: String, required: true },
  email: { type: String, unique: true, required: true },
  password: { type: String, required: true },
  // Role for RBAC: 'user' for posting/claiming, 'admin' for moderation
  role: { type: String, enum: ["user", "admin"], default: "user" }
});
```

```
// Pre-save hook to hash password before saving (Security)
userSchema.pre('save', async function(next) {
  if (this.isModified('password')) {
    this.password = await bcrypt.hash(this.password, 10);
  }
  next();
});
```

```
module.exports = mongoose.model("User", userSchema);
```

## 2. LostFoundItem Model (models/[LostFoundItem.js](#))

```
// models/LostFoundItem.js
```

```
const mongoose = require("mongoose");
```

```
const itemSchema = new mongoose.Schema({
  title: { type: String, required: true },
  description: { type: String, required: true },
  type: { type: String, enum: ["Lost", "Found"], required: true }, // Lost or Found
  category: { type: String, required: true },
  location: { type: String, required: true },
  // Link to the user who posted it (for ownership RBAC)
  postedBy: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true
  },
  status: { type: String, enum: ["Open", "Claimed", "Resolved"], default: "Open" },
  imageUrl: { type: String } // Optional image
}, { timestamps: true });
```

```
module.exports = mongoose.model("LostFoundItem", itemSchema);
```

## B. Authentication and Authorization Middlewares (RBAC Logic)

These functions check the JWT and the user's role or ownership before granting access.

### 1. middleware/[auth.js](#)

```
// middleware/auth.js
```

```

const jwt = require('jsonwebtoken');

// 1. Basic Token Validation
exports.verifyToken = (req, res, next) => {
  // Check for token in 'Authorization: Bearer <token>' header
  let token;
  if (req.headers.authorization && req.headers.authorization.startsWith('Bearer')) {
    token = req.headers.authorization.split(' ')[1];
  }

  if (!token) {
    return res.status(401).json({ msg: 'Access denied. No token provided.' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    // Attach user info (id and role) to the request object
    req.user = decoded;
    next();
  } catch (err) {
    res.status(401).json({ msg: 'Invalid token.' });
  }
};

// 2. Role-Based Access Control (RBAC) - Admin Check
exports.verifyAdmin = (req, res, next) => {
  // Assumes verifyToken has run and req.user exists
  if (req.user.role !== 'admin') {
    return res.status(403).json({ msg: 'Forbidden. Admin access required.' });
  }
  next();
};

// 3. Ownership Check (Item-specific RBAC)
const LostFoundItem = require('../models/LostFoundItem');

exports.verifyOwner = async (req, res, next) => {
  try {
    // Find the item by ID
    const item = await LostFoundItem.findById(req.params.id);

    if (!item) {
      return res.status(404).json({ msg: 'Item not found.' });
    }

    // Check if the logged-in user (req.user.id) is the poster (item.postedBy)
    if (item.postedBy.toString() !== req.user.id) {
      return res.status(403).json({ msg: 'Forbidden. You do not own this item.' });
    }
  }
};

```

```

    req.item = item; // Attach item for later use
    next();
  } catch (err) {
    res.status(500).json({ msg: 'Server error during ownership check.' });
  }
};

```

### C. API Endpoints (`routes/auth.js` and `routes/items.js`)

**Auth Routes (routes/auth.js): Handles login and registration.**

```

// routes/auth.js
const express = require('express');
const router = express.Router();
const User = require('../models/User');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');

// Helper to generate JWT
const generateToken = (user) => {
  return jwt.sign(
    { id: user._id, role: user.role }, // Payload with user ID and ROLE
    process.env.JWT_SECRET,
    { expiresIn: '1h' }
  );
};

// POST /api/auth/register
router.post('/register', async (req, res) => {
  const { username, email, password } = req.body;
  // ... validation and check for existing user ...
  try {
    const user = new User({ username, email, password, role: 'user' }); // Default role is 'user'
    await user.save();
    const token = generateToken(user);
    res.status(201).json({ token, user: { id: user._id, username: user.username, role: user.role } });
  } catch (err) {
    res.status(500).json({ msg: 'Server error during registration.' });
  }
});

// POST /api/auth/login
router.post('/login', async (req, res) => {
  const { email, password } = req.body;

```

```

try {
  const user = await User.findOne({ email });
  if (!user) return res.status(400).json({ msg: 'Invalid Credentials.' });

  const isMatch = await bcrypt.compare(password, user.password);
  if (!isMatch) return res.status(400).json({ msg: 'Invalid Credentials.' });

  const token = generateToken(user);
  res.json({ token, user: { id: user._id, username: user.username, role: user.role } });
} catch (err) {
  res.status(500).json({ msg: 'Server error during login.' });
}
});

module.exports = router;

```

### **Item Routes (routes/items.js): Protected by JWT and RBAC middlewares.**

```

// routes/items.js
const express = require('express');
const router = express.Router();
const LostFoundItem = require('../models/LostFoundItem');
const { verifyToken, verifyAdmin, verifyOwner } = require('../middleware/auth');

// GET /api/items - View all items (Public access)
router.get('/', async (req, res) => {
  try {
    const items = await LostFoundItem.find().populate('postedBy', 'username email'); // Show poster info
    res.json(items);
  } catch (err) {
    res.status(500).json({ msg: 'Server Error.' });
  }
});

// POST /api/items - Post a new item (Protected: verifyToken)
router.post('/', verifyToken, async (req, res) => {
  const { title, description, type, category, location, imageUrl } = req.body;
  try {
    const newItem = new LostFoundItem({
      title, description, type, category, location, imageUrl,
      postedBy: req.user.id // Set the poster ID from the JWT payload
    });
    const item = await newItem.save();
    res.status(201).json(item);
  }
});

```

```

    } catch (err) {
      res.status(500).json({ msg: 'Server Error creating item.' });
    }
  });

// PUT /api/items/:id - Update an item (Protected: verifyToken & verifyOwner)
router.put('/:id', verifyToken, verifyOwner, async (req, res) => {
  // verifyOwner has already checked permission and attached req.item
  const { title, description, status } = req.body;
  try {
    const updatedItem = await LostFoundItem.findByIdAndUpdate(
      req.params.id,
      { $set: { title, description, status } },
      { new: true }
    );
    res.json(updatedItem);
  } catch (err) {
    res.status(500).json({ msg: 'Server Error updating item.' });
  }
});


// DELETE /api/items/:id - Delete an item (Protected: verifyToken & (verifyAdmin OR verifyOwner -
// simpler to use verifyAdmin for moderation))
router.delete('/:id', verifyToken, verifyAdmin, async (req, res) => {
  try {
    const item = await LostFoundItem.findById(req.params.id);
    if (!item) return res.status(404).json({ msg: 'Item not found' });

    await item.remove();
    res.json({ msg: 'Item removed' });
  } catch (err) {
    res.status(500).json({ msg: 'Server Error deleting item.' });
  }
});

module.exports = router;

```

**OUTPUT :**


 **Lost & Found**

[Browse Items](#) [My Items](#) [+ Add Item](#)



## Welcome back

Sign in to your account

Email Address

 admin@portal.com

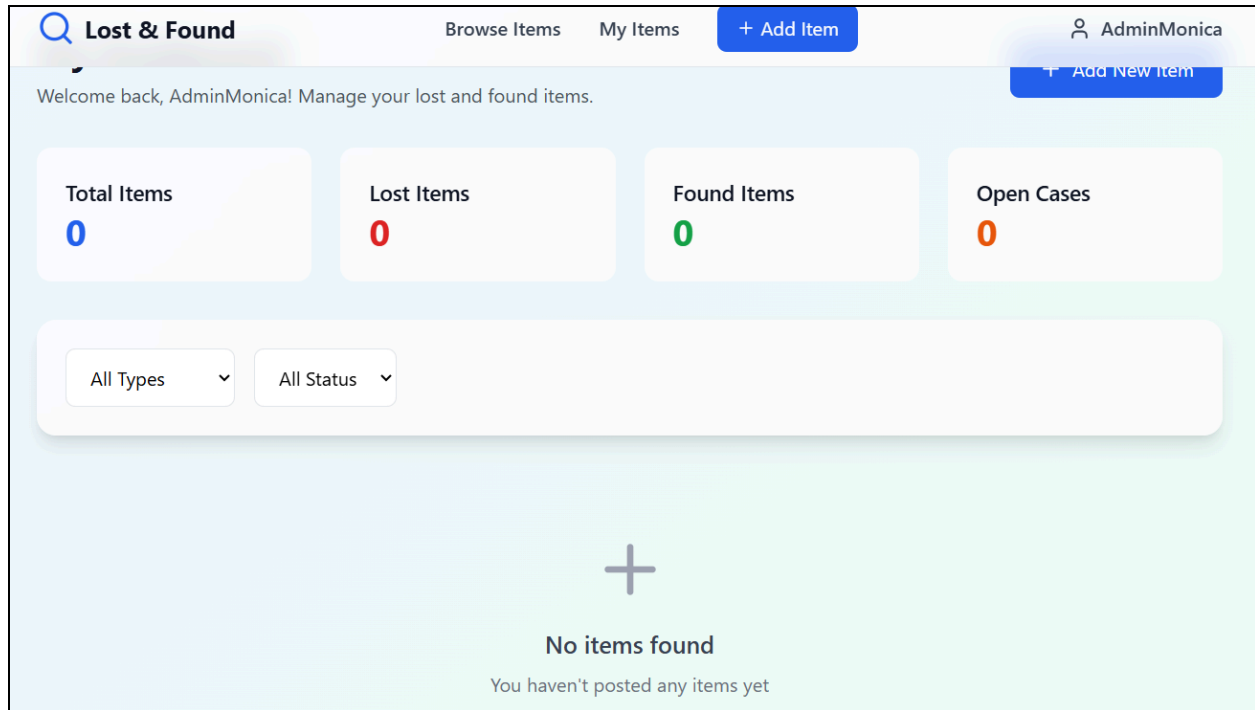
Password

 ..... 

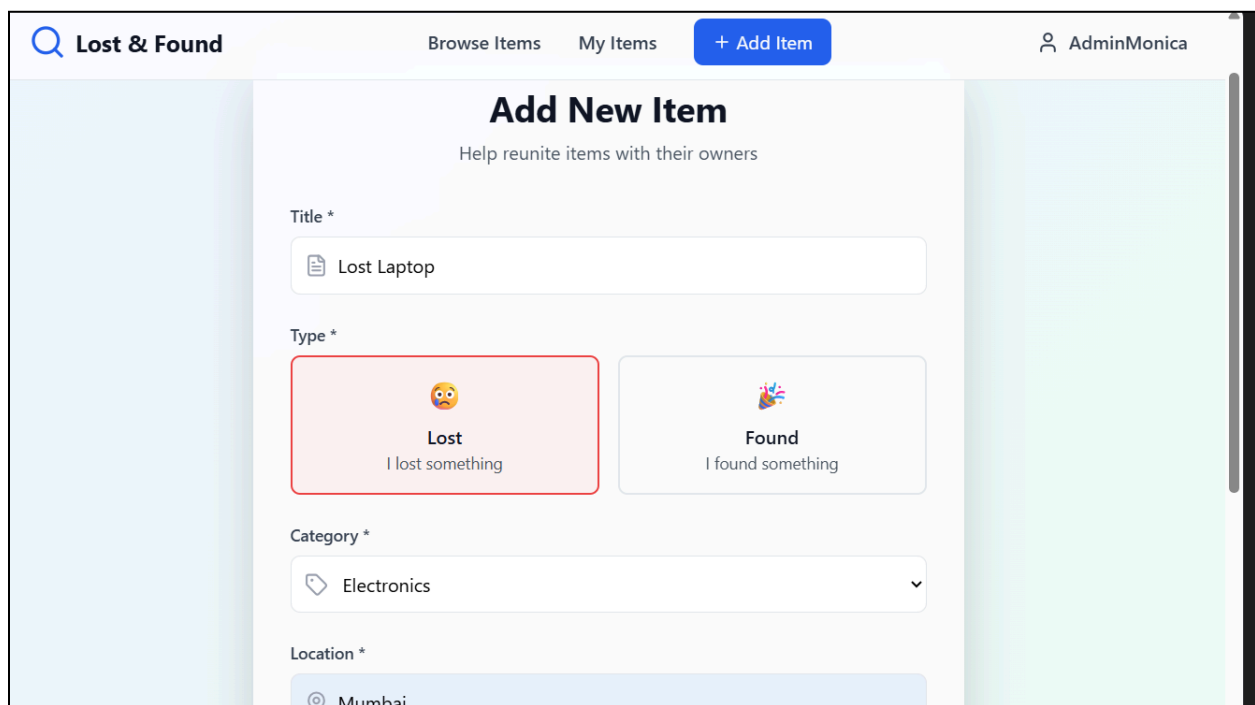
Sign In

Don't have an account? [Sign up](#)

Fill out the form (e.g., [admin@portal.com](#)). Click Register. **Result:** Redirects to the **Dashboard/Home Page** (Success).

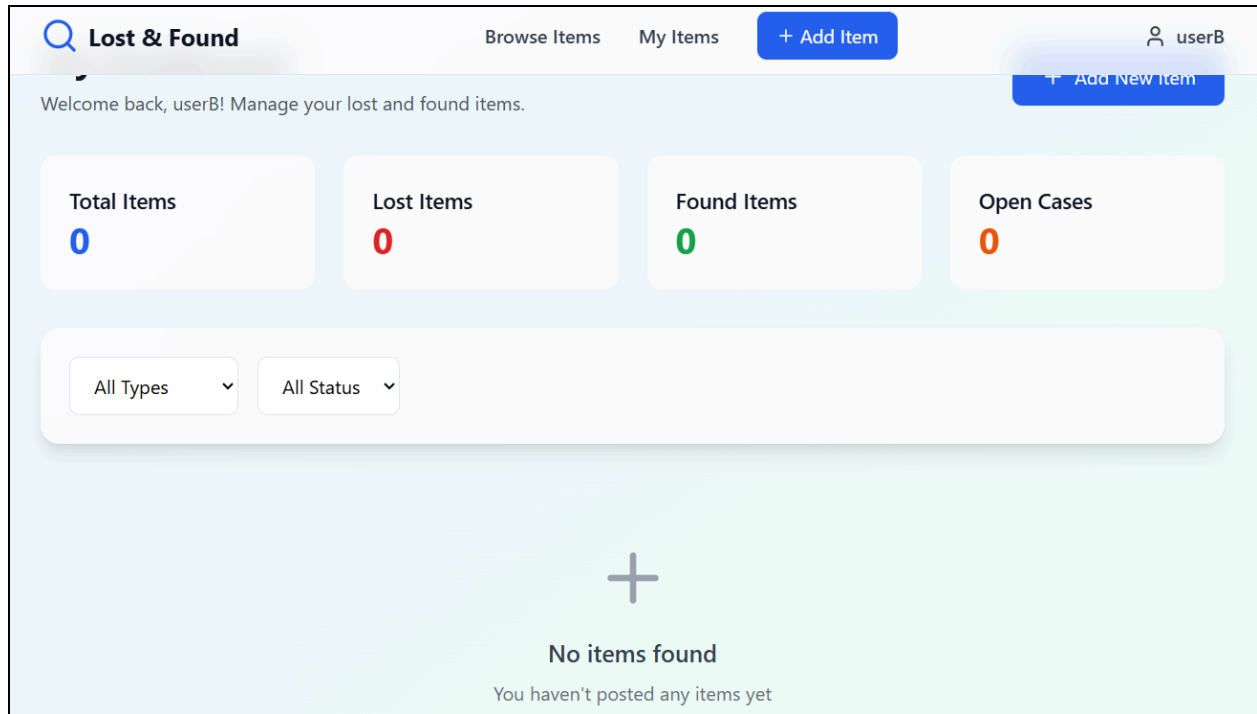


Navigate to **Add Item** (/add). Post "Lost Laptop." **Result:** Item appears on the Dashboard list.



**Log out.** Go to Register. Create a second user (userB@portal.com). Click Register. **Result:** Redirects to the Dashboard

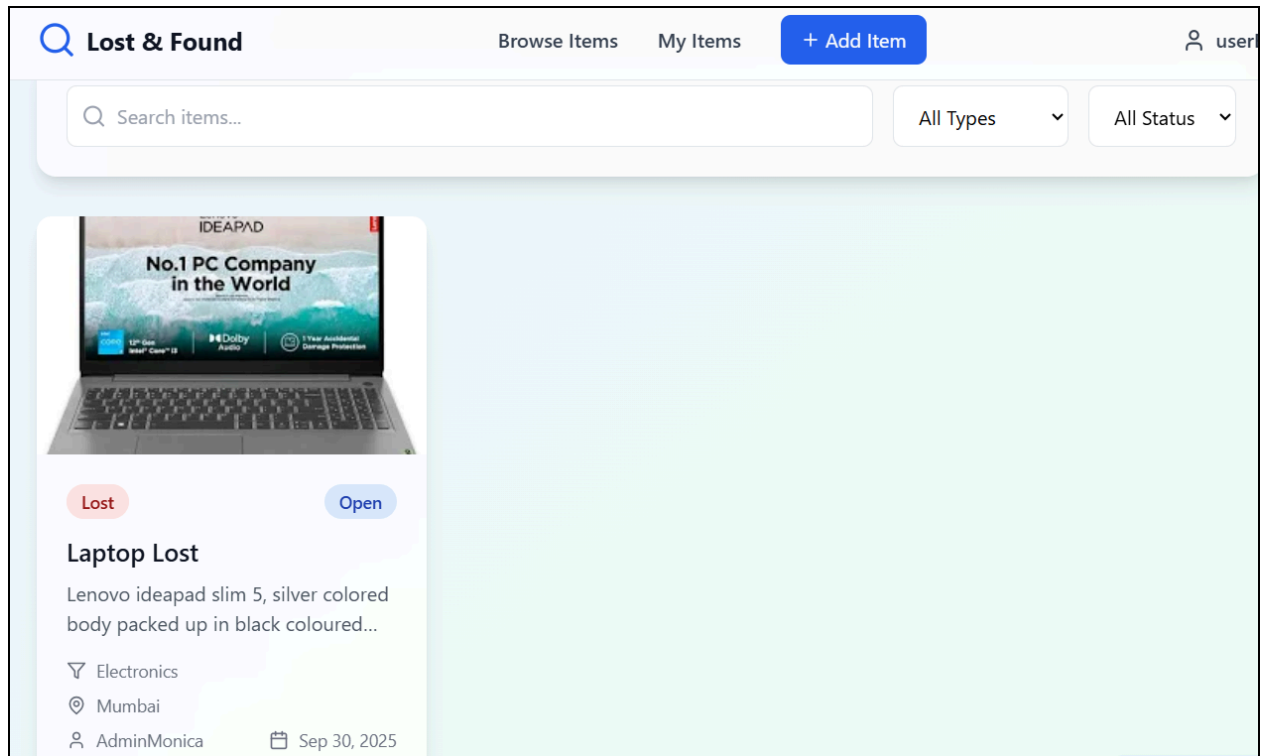




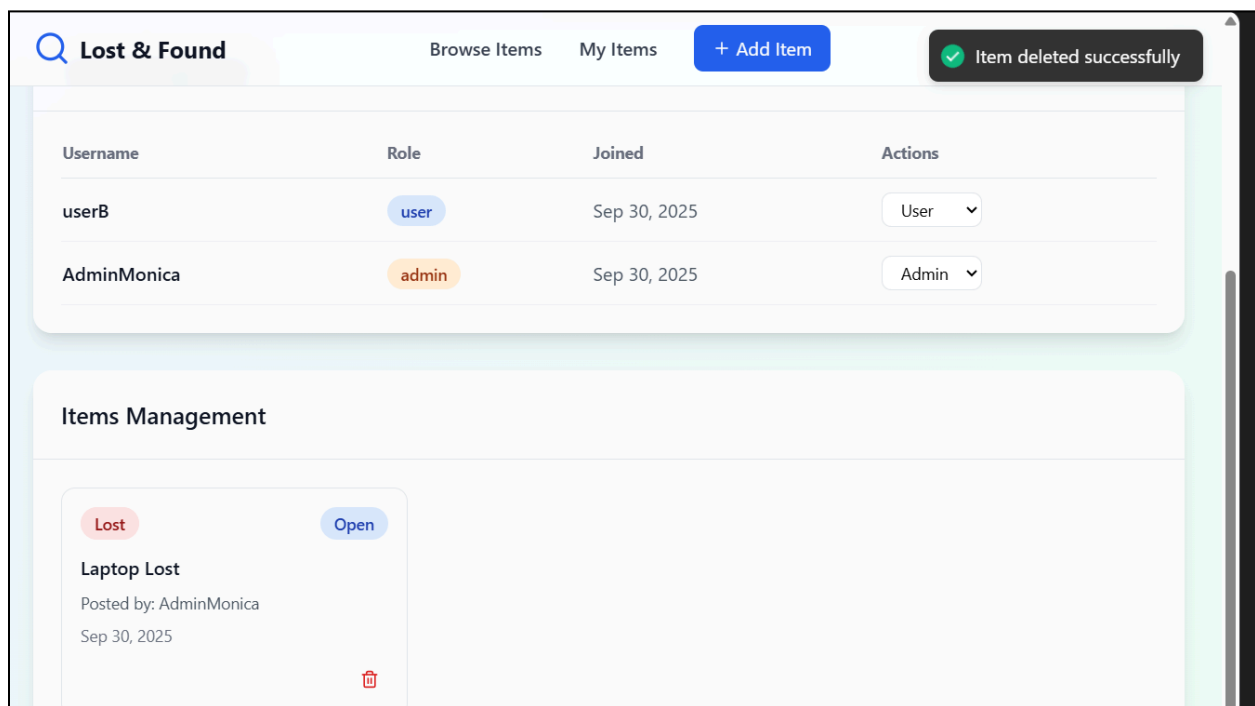
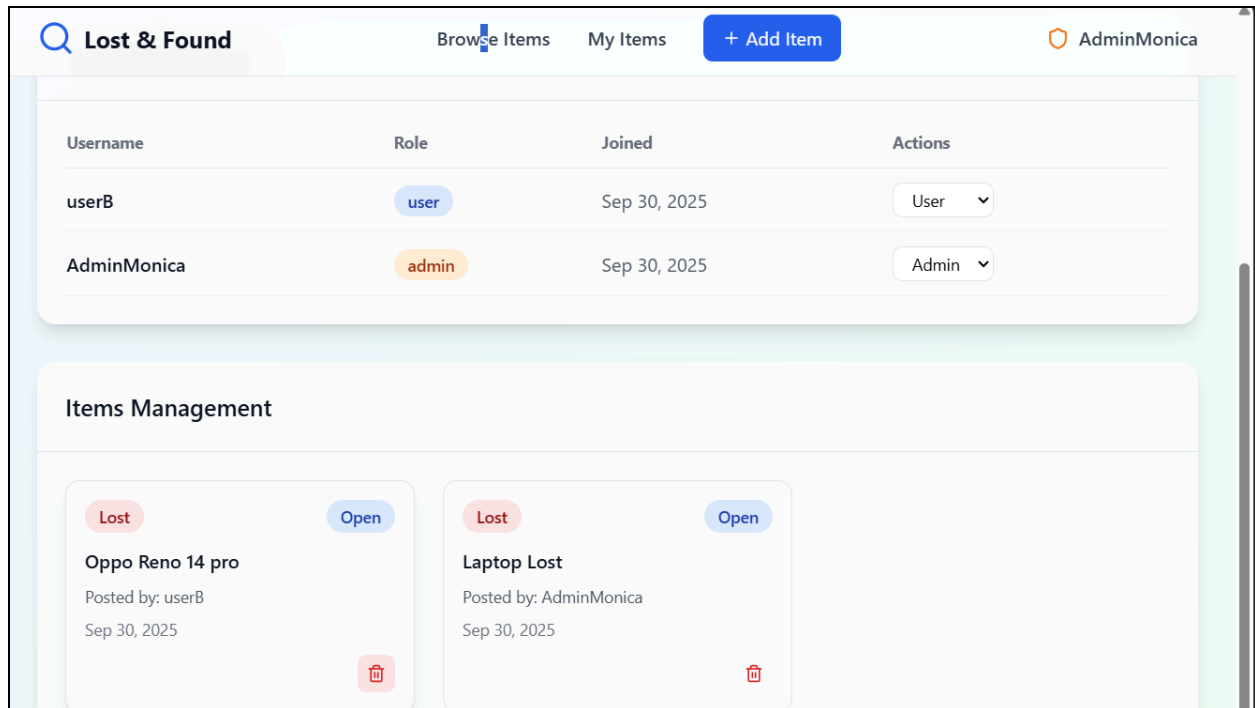
Navigate through protected routes (e.g., </dashboard>). **Result:** Access is granted.

	id	username	role	created_at
	8c007076-594c-41f2-860e-76c04...	userB	user	2025-09-30 06:00:35.492808+
	b49batt1a-6f5d-4572-877b-677a52...	AdminMonica	admin	2025-09-30 05:54:49.952409+

While logged in as **User B**, find **Item A** (the laptop posted by User A) on the Home/Dashboard list. **Find the Delete button** associated with Item A. **Result:** Not visible as this item was not added by UserB (This validates the `verifyOwner` equivalent RLS policy).



**Log out, then Log back in as User A (Admin).** Find **Item A** (their own) and **Item B** (posted by User B). **Click the Delete button** on **Item B** (the non-owned item). **Result:** Item B instantly disappears, and a **Success Message/Toast** appears: **"Item deleted successfully."** (This validates the **verifyAdmin** equivalent RLS policy).



## Conclusion:

The experiment successfully demonstrated the implementation of **authentication and role-based authorization** using JWT.