Multi-Tenant SaaS Platform

PROJECT REPORT

Submitted by

Jaskeerat Singh - 23BCS10601
Aditya Tandon - 23BCS13482
Gurkirat Singh - 23BCS12193

in partial fulfilment for the award of the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

Chandigarh University

November 2025

BONAFIDE CERTIFICATE

Certified that this project report **"MULTI-TENANT SAAS PLATFORM"** is the bonafide work of **"Drishti Salar (23IBD70004)"** who carried out the project work under my/our supervision.

<<Signature of the Supervisors>>

SIGNATURE

<<Signature of the AGM-Technical>>

SIGNATURE

Submitted for the project viva-voce examination held

INTERNAL EXAMINER

EXTERNAL EXAMINER

Project Title

Multi-tenant SaaS Platform

Project Description

This project is a Multi-Tenant SaaS (Software as a Service) Application developed using the MERN stack (MongoDB, Express.js, React.js, Node.js).
The goal of the application is to allow multiple organizations (called tenants) to use the same software, while ensuring:

1. Each tenant's data is completely isolated
2. Each tenant sees their own theme, colors, and branding
3. Users can only log in to their assigned tenant
4. All tenants use a single shared backend and database, but securely separated

This type of application is commonly used in real-world SaaS platforms like Shopify, Freshdesk, Slack, and Zoho.

☑ Project Overview

A multi-tenant system means that one application serves multiple clients, but each client feels like the software was built only for them.
In this project:

- Multiple tenants share the same server and database
- Each tenant has its own users, own resources, and own theme
- The system identifies the tenant through the login URL or tenant ID
- All operations are automatically filtered using the `tenantId`

This ensures that Tenant A cannot see anything from Tenant B, making the system secure.

☑ Core Features

1. Tenant Identification (Tenant Resolver)

When a user logs in or makes a request, the system detects which tenant they belong to.
This is done through:

- The URL path (example: `/api/auth/login/tenant1`)
- The tenant ID entered on the login page

The backend attaches this tenantId to every operation, ensuring:

1) Each request is tenant-specific
2) All queries run with tenant filters
3) No cross-tenant access is possible

2. Data Isolation

Each database record—Users, Tenants, Resources—contains a tenantId.
All database operations are performed as:

```
find({ tenantId: currentTenant })
```

This ensures:

1) Tenant1 never sees Tenant2 data
2) Data stays separated inside a single shared MongoDB
3) Clean and professional SaaS-level security

3. Secure Authentication

The login is performed using:

- Tenant ID
- Email
- Password

The backend validates the user and generates a JWT that includes:

- userId
- tenantId

Every protected route checks this token and ensures the user is authorized for their own tenant.

1) Users cannot log into the wrong tenant
2) Users cannot change tenant from the frontend
3) Full protection from unauthorized access

4. Dynamic Theme and Branding

The project supports different themes for each tenant.
Each tenant has settings such as:

- Primary color
- Secondary color

- Logo
- Typography

When a user logs in, the theme is fetched and applied using:

- CSS variables
- React context
- Inline styles

This allows:

☑ Tenant1 → Blue theme
☑ Tenant2 → Green theme
☑ Real SaaS-style customization

No redeployment is needed when a tenant updates their theme.

5. Admin Panel for Each Tenant

Every tenant has its own Admin Dashboard where the admin can:

- Change the tenant name
- Update theme colors
- Upload a new logo
- Enable or disable features/modules

These changes are applied instantly to the tenant's frontend.

6. Resource Management

Each tenant can add and manage their own resources (like projects, data records, assets, etc.).
Resource operations such as:

- Create
- Read
- List

are done through the React interface using simple forms and buttons.

Backend ensures every resource is stored under:

tenantId: currentTenant

☑ Full isolation
☑ Scalable
☑ Easy to manage

7. No External API Tool Used ☑

All testing and validation were done through:

- The React frontend
- Browser requests (fetch API)
- MongoDB Compass
- Console logs

No Postman or external testing tool was used.

8. Seeding Initial Tenants

When the backend starts, the system automatically creates two sample tenants:

Tenant 1
- ID: tenant1
- Email: admin@tenant1.com
- Password: 123456

Tenant 2
- ID: tenant2
- Email: admin@tenant2.com
- Password: 123456

These allow easy testing.

9. GitHub Integration

The entire project (backend + frontend) is uploaded to a public GitHub repository for submission and evaluation.

☑ Overall System Flow

1. User enters tenant ID and login details
2. Backend verifies tenant & user
3. Backend sends JWT with tenant ID
4. Frontend loads the tenant’s theme

5. User accesses dashboard and resources
6. All operations run with tenant-specific filters

Hardware/Software Requirements

☑ Hardware Requirements

Minimum Hardware:

- Processor: Intel Core i3 or equivalent
- RAM: 4 GB
- Storage: 2 GB free space
- Display: 1366 × 768 resolution

Recommended Hardware:

- Processor: Intel Core i5 or above
- RAM: 8 GB or more
- Storage: 10 GB free (for Node, MongoDB, and project workspace)
- Display: Full HD (1920 × 1080)

These requirements ensure smooth running of Node.js server, MongoDB database, and React development environment.

☑ Software Requirements

Operating System:

- Windows 10 / 11
- macOS
- Linux (Ubuntu)

Frontend Software:

- React.js
- Node.js (v18 or later)
- Vite (React build tool)
- npm (Node Package Manager)
- Code editor: Visual Studio Code

Backend Software:

- Node.js
- Express.js
- Mongoose (ORM for MongoDB)
- JSON Web Token (JWT)
- bcrypt.js
- dotenv

Database Software:

- MongoDB Community Server
- MongoDB Compass (for viewing collections)

Version Control:

- Git
- GitHub (public repository)

Browser:

- Google Chrome / Microsoft Edge / Firefox

Additional Tools (Used Indirectly):

- Terminal / Powershell (for running commands)
- npm scripts (for starting backend & frontend)

ER Diagram

My project contains three main entities:

☑ 1. Tenant Entity

What is a Tenant?

A *tenant* represents a company or organization using the SaaS platform.
Example: *tenant1, tenant2*

Attributes:

- tenantId → Unique ID for the tenant (example: tenant1)
- tenantName → Name of the company
- theme → Stores color settings
- logo → Stores logo URL
- modules → Which features are enabled

Why it is important?

Every user and every resource belongs to a specific tenant.
This ensures data isolation in the multi-tenant system.

☑ 2. User Entity

What is a User?

A user belongs to one company (tenant) and logs in using email + password.

Attributes:

- userId → Unique user ID

- name
- email
- password
- role → admin or normal user
- tenantId → Foreign key (connects the user to a tenant)

Relationship:

A single Tenant can have many Users
(1-to-many relationship)

☑ 3. Resource Entity

What is a Resource?

A resource is any item created by a tenant.
(Example: projects, records, tasks — depending on system design)

Attributes:

- resourceId
- title
- description
- createdAt
- tenantId → Foreign key (links resource to a tenant)

Relationship:

A single Tenant can have many Resources
(1-to-many relationship)

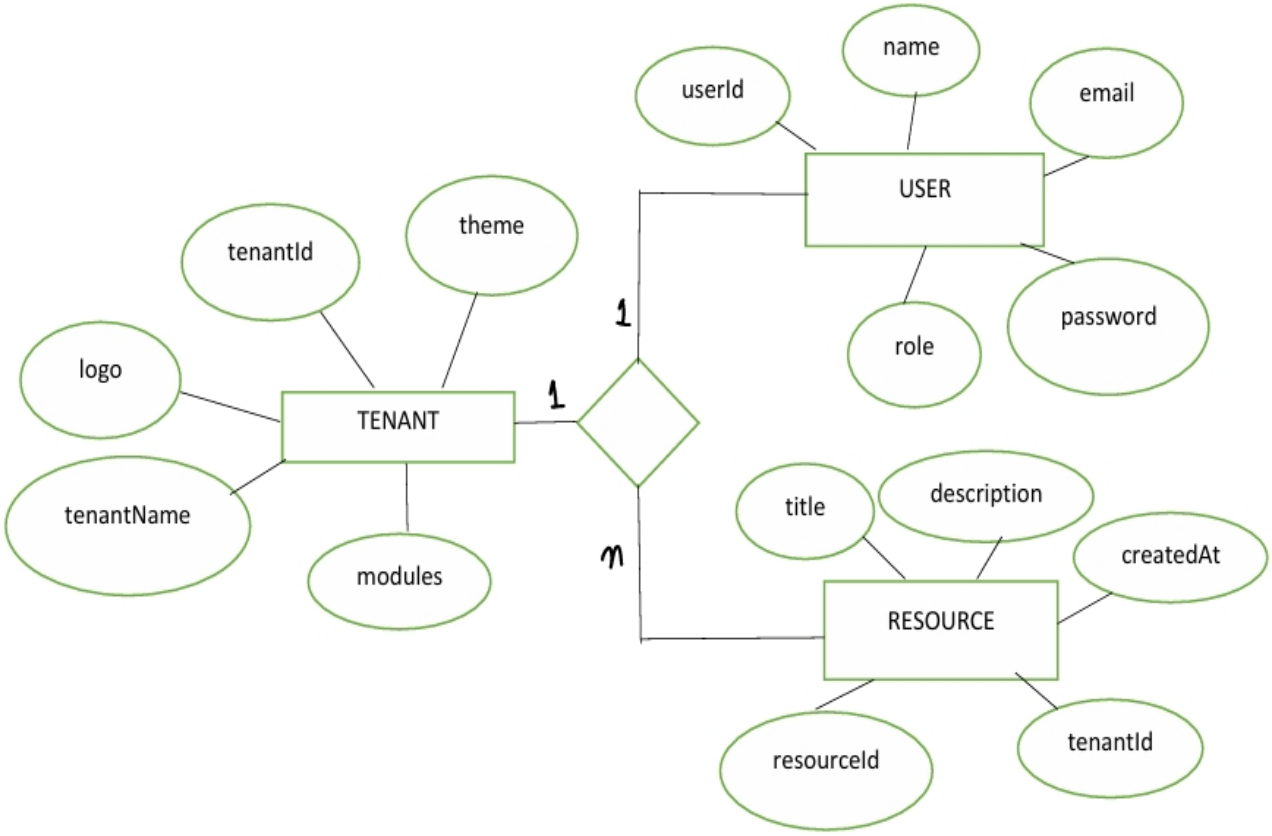
☑ All Relationships Together

☑ Relationship 1: Tenant → User

One Tenant can have many Users
But one User belongs to only one Tenant

☑ Relationship 2: Tenant → Resource

One Tenant can have many Resources
But one Resource belongs to only one Tenant



Database Schema

1. Introduction

A Database Schema defines the structure of the database used in the Multi-Tenant SaaS Application.
It describes how the data is organised, what collections (tables) exist, what fields they contain, and how they relate to each other.

Since this project uses MongoDB, the schema is represented through collections and documents instead of relational tables.
However, the overall structure follows clear relationships using unique identifiers.

The database is designed to support a multi-tenant architecture, where each tenant's data remains securely isolated using the tenantId field.

☑ 2. Collections and Fields

Below is the complete schema for all collections used in the project.

☑ 2.1 Tenants Collection

Stores information about each tenant who uses the SaaS system.

Field	Type	Description
_id	ObjectId	Auto-generated unique ID
tenantId	String	Unique identifier for tenant (e.g., "tenant1")
tenantName	String	Tenant’s business/organization name
theme	String	UI theme selected by tenant
logo	String	Logo URL or file path
modules	Array	Enabled modules/features

☑ 2.2 Users Collection

Stores all users who belong to a specific tenant.

Field	Type	Description
_id	ObjectId	Auto-generated unique ID
tenantId	String	Links user to specific tenant
name	String	User's full name
email	String	User login email

Field	Type	Description
password	String	Hashed password
role	String	User role (Admin / User)

☑ 2.3 Resources Collection

Stores resources uploaded or created by users within a tenant.

Field	Type	Description
_id	ObjectId	Auto-generated unique ID
tenantId	String	Identifies which tenant the resource belongs to
title	String	Resource title
description	String	Details about the resource
createdAt	Date	Timestamp of creation

☑ 3. Relationships Between Collections

Although MongoDB is non-relational, the project maintains logical relationships using the tenantId.

☑ Relationship 1: Tenant → Users

- One Tenant can have many Users
- Users include tenantId
- Ensures users only access their own tenant data

☑ Relationship 2: Tenant → Resources

- One Tenant can have many Resources
- Every resource contains tenantId
- Prevents cross-tenant data mixing

☑ Relationship 3: User → Resources

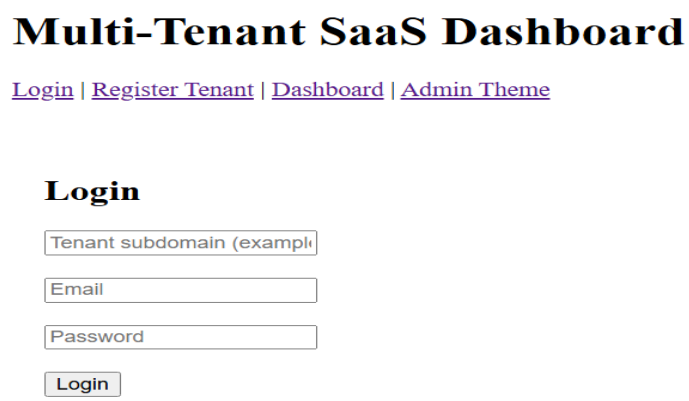
- A user can create multiple resources

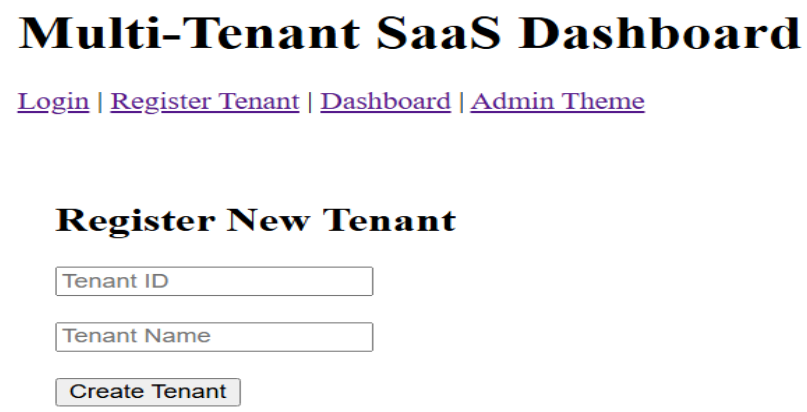
- Relationship maintained through tenantId, ensuring resource access is restricted to the same tenant
- ☑ 4. Summary of Schema Design

  The schema ensures:

  ☑ Strong data isolation between tenants
  ☑ Secure access using tenant-based authentication
  ☑ Scalable structure for adding more modules
  ☑ Easy expansion for future features
  ☑ Simple querying and filtering using tenantId

Front-End Screens





Multi-Tenant SaaS Dashboard

Login | Register Tenant | Dashboard | Admin Theme

Admin Theme Editor

Primary Color:

Secondary Color:

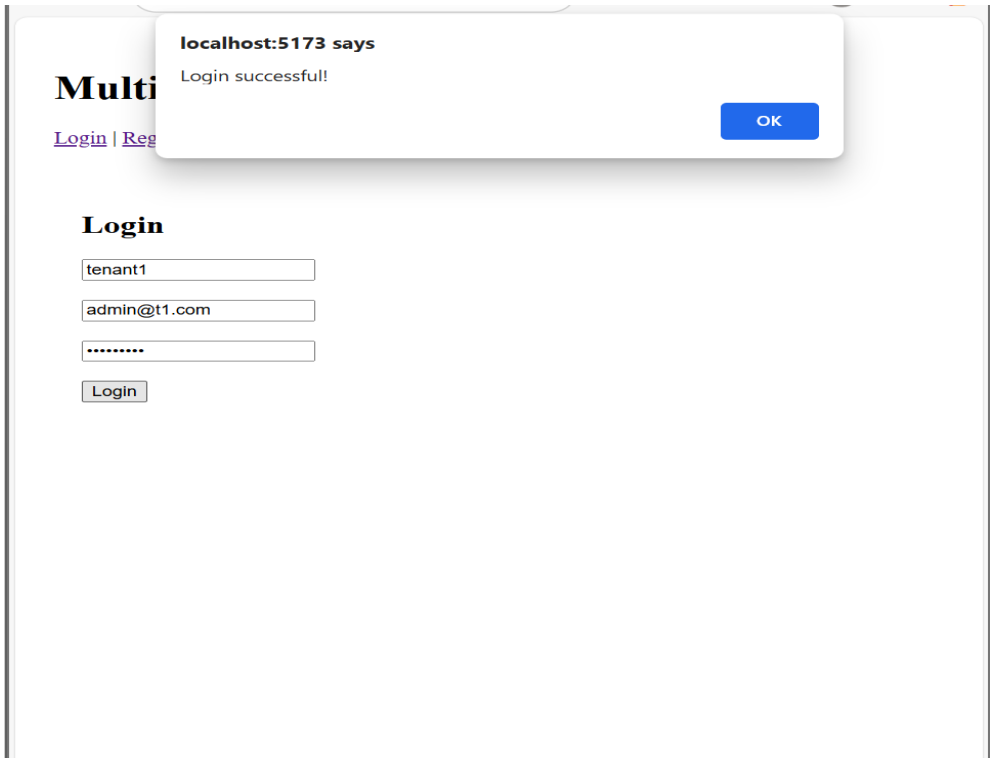
Font:

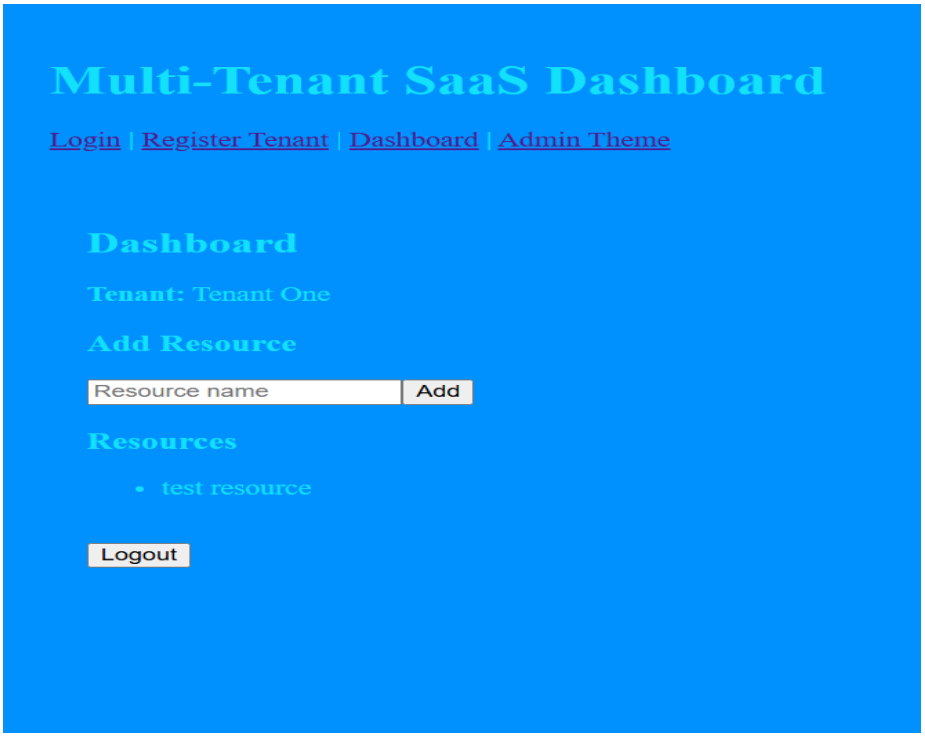
Logo URL:

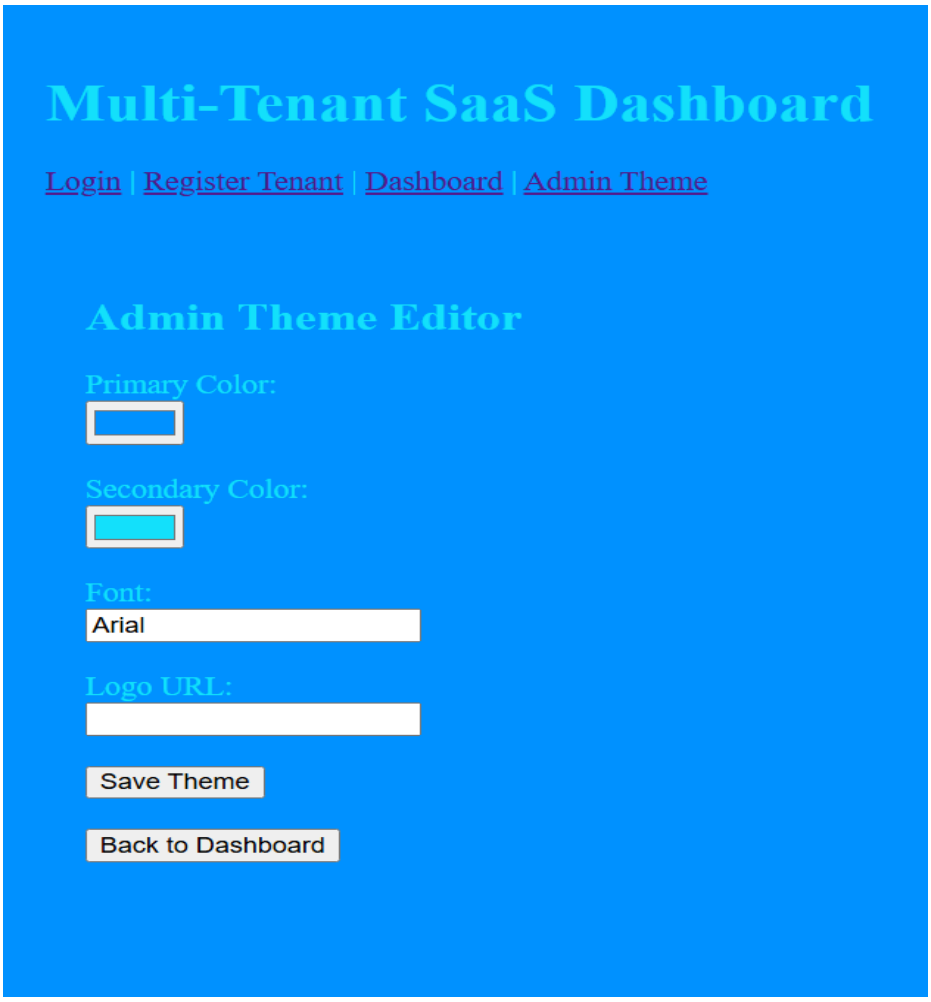
Save Theme

Back to Dashboard

Output Screens and Reports




Multi-Tenant SaaS Dashboard
Login | Register Tenant | Dashboard | Admin Theme
Dashboard
Tenant: Tenant One
Add Resource
Resource name
Add
Resources
test resource
Logout


Multi-Tenant SaaS Dashboard
Login | Register Tenant | Dashboard | Admin Theme
Admin Theme Editor
Primary Color:
Secondary Color:
Font:
Arial
Logo URL:
Save Theme
Back to Dashboard

Limitation & Future Scope

☑ Limitations

1. Basic UI and Features
   The current system includes only essential modules such as tenant registration, login, and resource management. Advanced SaaS features are not yet implemented.
2. No Role-Based Access Control (RBAC)
   Only basic roles exist (Admin/User). There is no fine-grained permission system for different user levels.
3. Limited Frontend UI Customization
   Tenants can choose a theme, but deeper UI customization (colors, layout changes, branding options) is not fully developed.
4. No Advanced Security Measures
   Features like OTP login, MFA, rate limiting, and audit logs are not included.
5. No Billing or Subscription Management
   SaaS systems usually include subscription plans, payment gateways, and usage tracking, which are not part of this version.
6. No API Rate Limiting / Throttling
   Heavy usage by one tenant could affect other tenants if rate limiting is not applied.
7. Data Backup and Recovery Not Implemented
   Automated backups and restore features are missing.
8. Limited Analytics
   There are no dashboards showing tenant activity, user statistics, or resource usage.

☑ Future Scope

1. Full Role-Based Access Control (RBAC)
   Implementing granular access permissions for Admin, Manager, and Regular Users.
2. Advanced Customization for Tenants

- Custom themes
- Fonts
- Custom dashboard layouts
- Tenant branding (colors, banners)

3. Subscription and Billing Integration

- Stripe/Razorpay integration
- Automatic billing
- Tenant plans (Free, Basic, Premium)

4. Multi-Factor Authentication (MFA)
   Enhancing login security through OTP, email verification, or authenticator apps.
5. API Rate Limiting
   Ensuring fair resource usage for each tenant.
6. Automated Backups & Disaster Recovery
   Daily backups, restore points, and high-availability deployment.
7. Advanced Analytics Dashboard

- Resource usage
- User activity logs
- Tenant-specific reports
- Graphs and insights

8. Microservices Architecture
   Converting the backend into microservices for scalability.
9. Containerization & Cloud Deployment
   Deploying on Docker + Kubernetes for automatic scaling and high performance.
10. Add Notification System
    Email or in-app notifications for tenant alerts.

GitHub URL

https://github.com/drishtisalar172/multi-tenant-saas

Presentation Slides

Multi-Tenant SaaS Application

- A Full Stack Project using Node.js, Express, MongoDB, and React

Project Overview

- Multi-tenant SaaS platform
- Each tenant has isolated data
- Tenant-specific themes
- Backend: Node.js + Express
- Frontend: React (Vite)

Objectives

- Implement multi-tenant architecture
- Secure tenant-based authentication
- Dashboard with isolated data
- Dynamic theming per tenant
- REST API integration

System Architecture

- React Frontend
- Express.js Backend
- MongoDB Database
- JWT Authentication
- Middleware for Tenant Resolver

Key Features

- Tenant Registration
- Tenant Login
- Dashboard Access
- Resource Isolation
- Theme Customization

Database Schema

- TENANT (tenantId, name, logo, theme)
- USER (id, email, password, tenantId)
- RESOURCE (id, title, description, tenantId)

Limitations

- Basic UI
- No advanced analytics
- No role-based access
- Theme customization limited

Future Scope

- • RBAC roles
- • Advanced dashboards
- • Email notifications
- • Full theme editor

Conclusion

- This project successfully demonstrates isolated multi-tenant architecture with secure authentication and customizable themes.

Project Code

backend/.env

```
PORT=5000
MONGO_URI=mongodb://127.0.0.1:27017/multitenant_saas
JWT_SECRET=supersecretkey
```

backend/package.json

```
{
  "name": "multitenant-backend",
  "version": "1.0.0",
```

```
  "type": "module",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "seed": "node utils/seed.js"
  },
  "dependencies": {
    "bcryptjs": "^2.4.3",
    "cors": "^2.8.5",
    "dotenv": "^16.4.0",
    "express": "^4.19.2",
    "jsonwebtoken": "^9.0.2",
    "mongoose": "^8.3.0",
    "morgan": "^1.10.0",
    "uuid": "^9.0.1"
  }
}
```

backend/server.js

```
import express from "express";
import cors from "cors";
import morgan from "morgan";
import dotenv from "dotenv";
import "./config/db.js";
import tenantResolver from "./middleware/tenantResolver.js";

dotenv.config();

const app = express();
app.use(cors());
app.use(express.json());
app.use(morgan("dev"));

// Attach tenant context (from subdomain, header, or path)
app.use(tenantResolver);
```

```
// Routes
import authRoutes from "./routes/auth.js";
import tenantRoutes from "./routes/tenants.js";
import resourceRoutes from "./routes/resources.js";

app.use("/api/auth", authRoutes);
app.use("/api/tenants", tenantRoutes);
app.use("/api/resources", resourceRoutes);

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`API running on http://localhost:${PORT}`));


backend/config/db.js
import mongoose from "mongoose";
import dotenv from "dotenv";
dotenv.config();

const uri = process.env.MONGO_URI;

mongoose
  .connect(uri, { dbName: "multitenant_saas" })
  .then(() => console.log("☑ MongoDB connected"))
  .catch((e) => console.error("Mongo error:", e.message));


backend/models/Tenant.js
import mongoose from "mongoose";

const tenantSchema = new mongoose.Schema(
  {
    tenantId: { type: String, unique: true, required: true, index: true },
    name: { type: String, required: true },
    theme: {
```

```
      primaryColor: { type: String, default: "#4f46e5" },
      secondaryColor: { type: String, default: "#111827" },
      font: { type: String, default: "system-ui" },
      logo: { type: String, default: "" }
    },
    modules: [{ type: String }]
  },
  { timestamps: true }
);

export default mongoose.model("Tenant", tenantSchema);
```

backend/models/User.js

```
import mongoose from "mongoose";

const userSchema = new mongoose.Schema(
  {
    tenantId: { type: String, required: true, index: true },
    name: { type: String, required: true },
    email: { type: String, required: true, index: true },
    password: { type: String, required: true },
    role: { type: String, enum: ["admin", "user"], default: "admin" }
  },
  { timestamps: true }
);

// Compound index for quick tenant scoping
userSchema.index({ tenantId: 1, email: 1 }, { unique: true });

export default mongoose.model("User", userSchema);
```

backend/models/Resource.js

```
import mongoose from "mongoose";
```

```
const resourceSchema = new mongoose.Schema(
  {
    tenantId: { type: String, required: true, index: true },
    name: { type: String, required: true }
  },
  { timestamps: true }
);

resourceSchema.index({ tenantId: 1, createdAt: -1 });

export default mongoose.model("Resource", resourceSchema);
```

backend/middleware/tenantResolver.js

```
// Resolves tenant from subdomain (tenant.localhost), header, or path /t/:tenant
export default function tenantResolver(req, res, next) {
  let tenant = null;

  // 1) Signed/Trusted header (preferred in prod behind gateway)
  if (req.headers["x-tenant"]) {
    tenant = String(req.headers["x-tenant"]).trim().toLowerCase();
  }

  // 2) Subdomain like tenant.localhost or tenant.example.com
  if (!tenant && req.hostname) {
    const parts = req.hostname.split(".");
    if (parts.length >= 3) tenant = parts[0]; // tenant.localhost or tenant.example.com
  }

  // 3) Path prefix /t/:tenant (optional pattern)
```

```
  if (!tenant && req.path.startsWith("/t/")) {
    tenant = req.path.split("/")[2];
  }

  // 4) Fallback: from auth routes that carry :tenant param
  if (!tenant && req.params?.tenant) {
    tenant = req.params.tenant;
  }

  // Attach to request
  req.tenantId = tenant || null;
  next();
}
```

backend/middleware/auth.js

```
import jwt from "jsonwebtoken";

export default function auth(required = true) {
  return (req, res, next) => {
    const header = req.headers.authorization || "";
    const token = header.startsWith("Bearer ") ? header.slice(7) : null;

    if (!token) {
      if (required) return res.status(401).json({ message: "Unauthorized" });
      return next();
    }

    try {
      const payload = jwt.verify(token, process.env.JWT_SECRET);
      req.user = payload;
      // Enforce tenant match on every request
```

```
    if (req.tenantId && req.tenantId !== payload.tenantId) {
      return res.status(403).json({ message: "Cross-tenant access denied" });
    }
    next();
  } catch {
    return res.status(401).json({ message: "Invalid token" });
  }
 };
}
```

backend/routes/auth.js

```
import express from "express";
import bcrypt from "bcryptjs";
import jwt from "jsonwebtoken";
import User from "../models/User.js";
import Tenant from "../models/Tenant.js";

const router = express.Router();

// Login with tenant in path: /api/auth/login/:tenant
router.post("/login/:tenant", async (req, res) => {
 try {
  const tenantId = req.params.tenant?.toLowerCase();
  const { email, password } = req.body;

  if (!tenantId) return res.status(400).json({ message: "Missing tenant" });

  const tenant = await Tenant.findOne({ tenantId });
  if (!tenant) return res.status(404).json({ message: "Tenant not found" });
```

```
    const user = await User.findOne({ tenantId, email });
    if (!user) return res.status(400).json({ message: "Invalid email or
password" });

    const ok = await bcrypt.compare(password, user.password);
    if (!ok) return res.status(400).json({ message: "Invalid email or password"
});

    const token = jwt.sign(
      { userId: user._id, tenantId, role: user.role },
      process.env.JWT_SECRET,
      { expiresIn: "8h" }
    );

    res.json({
      token,
      tenant: { tenantId, name: tenant.name, theme: tenant.theme },
      user: { name: user.name, email: user.email, role: user.role }
    });
  } catch (e) {
    res.status(500).json({ message: "Auth error" });
  }
});

export default router;
```

backend/routes/tenants.js

```
import express from "express";
import Tenant from "../models/Tenant.js";
import auth from "../middleware/auth.js";
```

```
const router = express.Router();

// Get current tenant info (requires token and tenant context)
router.get("/", auth(true), async (req, res) => {
  const tenant = await Tenant.findOne({ tenantId: req.user.tenantId });
  if (!tenant) return res.status(404).json({ message: "Tenant not found" });
  res.json({ tenantId: tenant.tenantId, name: tenant.name, theme: tenant.theme
});
});

// Admin: update theme
router.put("/theme", auth(true), async (req, res) => {
  const { theme } = req.body;
  const updated = await Tenant.findOneAndUpdate(
    { tenantId: req.user.tenantId },
    { $set: { theme: theme || {} } },
    { new: true }
  );
  res.json({ message: "Theme updated", theme: updated.theme });
});

// Public: register a tenant
router.post("/register", async (req, res) => {
  const { tenantId, name, themeColor } = req.body;
  if (!tenantId || !name) return res.status(400).json({ message: "tenantId and
name required" });

  const exists = await Tenant.findOne({ tenantId: tenantId.toLowerCase() });
  if (exists) return res.status(400).json({ message: "Tenant already exists" });

  const tenant = await Tenant.create({
```

```
    tenantId: tenantId.toLowerCase(),
    name,
    theme: { primaryColor: themeColor || "#4f46e5", secondaryColor: "#111827" }
  });

  res.json({ message: "Tenant registered successfully", tenant });
});

export default router;
```

backend/routes/resources.js

```
import express from "express";
import auth from "../middleware/auth.js";
import Resource from "../models/Resource.js";

const router = express.Router();

// List resources (tenant-scoped)
router.get("/", auth(true), async (req, res) => {
  const items = await Resource.find({ tenantId: req.user.tenantId }).sort({ createdAt: -1 });
  res.json(items);
});

// Create resource (tenant-scoped)
router.post("/", auth(true), async (req, res) => {
  const { name } = req.body;
  if (!name) return res.status(400).json({ message: "name required" });
  const created = await Resource.create({ tenantId: req.user.tenantId, name });
  res.json(created);
```

```
});

export default router;
```

backend/utils/seed.js

```
import bcrypt from "bcryptjs";
import "./../config/db.js";
import Tenant from "../models/Tenant.js";
import User from "../models/User.js";

async function seed() {
  await Tenant.deleteMany({});
  await User.deleteMany({});

  const tenants = [
    {
      tenantId: "tenant1",
      name: "Tenant One",
      theme: { primaryColor: "#7c3aed", secondaryColor: "#111827", font:
"system-ui", logo: "" }
    },
    {
      tenantId: "tenant2",
      name: "Tenant Two",
      theme: { primaryColor: "#059669", secondaryColor: "#0f172a", font:
"system-ui", logo: "" }
    }
  ];

  await Tenant.insertMany(tenants);

  const users = [
```

```
    {
      tenantId: "tenant1",
      name: "Admin T1",
      email: "admin@tenant1.com",
      password: await bcrypt.hash("123456", 10),
      role: "admin"
    },
    {
      tenantId: "tenant2",
      name: "Admin T2",
      email: "admin@tenant2.com",
      password: await bcrypt.hash("123456", 10),
      role: "admin"
    }
  ];

  await User.insertMany(users);
  console.log("☑ Seed done");
  process.exit(0);
}

seed().catch((e) => {
  console.error(e);
  process.exit(1);
});
```

frontend/package.json

```
{
  "name": "frontend",
  "private": true,
```

```
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "react": "^18.3.1",
    "react-dom": "^18.3.1",
    "react-router-dom": "^6.26.1"
  },
  "devDependencies": {
    "vite": "^5.4.0",
    "@eslint/js": "^9.8.0",
    "eslint": "^9.8.0"
  }
}
```

frontend/src/main.jsx

```
import React from "react";
import ReactDOM from "react-dom/client";
import { BrowserRouter } from "react-router-dom";
import App from "./App.jsx";

ReactDOM.createRoot(document.getElementById("root")).render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
);
```

frontend/src/App.jsx

```
import { Routes, Route, Link } from "react-router-dom";
import Login from "./pages/Login.jsx";
import RegisterTenant from "./pages/RegisterTenant.jsx";
import Dashboard from "./pages/Dashboard.jsx";
import AdminTheme from "./pages/AdminTheme.jsx";

export default function App() {
  return (
    <div style={{ padding: 20 }}>
      <h1>Multi-Tenant SaaS Dashboard</h1>

      <nav style={{ marginBottom: 16 }}>
        <Link to="/login">Login</Link> |{" "}
        <Link to="/register-tenant">Register Tenant</Link> |{" "}
        <Link to="/dashboard">Dashboard</Link> |{" "}
        <Link to="/admin-theme">Admin Theme</Link>
      </nav>

      <Routes>
        <Route path="/login" element={<Login />} />
        <Route path="/register-tenant" element={<RegisterTenant />} />
        <Route path="/dashboard" element={<Dashboard />} />
        <Route path="/admin-theme" element={<AdminTheme />} />
      </Routes>
    </div>
  );
}
```

frontend/src/pages/Login.jsx

```
import { useState } from "react";

export default function Login() {
  const [tenant, setTenant] = useState("");
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");

  const handleLogin = async () => {
    try {
      if (!tenant) return alert("Please enter tenant ID");

      const url = `http://localhost:5000/api/auth/login/${tenant}`;
      const res = await fetch(url, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ email, password })
      });

      const data = await res.json();
      if (data.token) {
        localStorage.setItem("token", data.token);
        localStorage.setItem("tenant", tenant);
        window.location.href = "/dashboard";
      } else {
        alert(data.message || "Login failed");
      }
    } catch {
      alert("Network error");
    }
```

```
  };

  return (
    <div style={{ padding: 20 }}>
      <h2>Login</h2>
      <input
        placeholder="Tenant ID (e.g., tenant1)"
        value={tenant}
        onChange={(e) => setTenant(e.target.value)}
      /><br /><br />
      <input
        placeholder="Email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
      /><br /><br />
      <input
        placeholder="Password"
        type="password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
      /><br /><br />
      <button onClick={handleLogin}>Login</button>
    </div>
  );
}
```

frontend/src/pages/RegisterTenant.jsx

```
import { useState } from "react";

export default function RegisterTenant() {
```

```
const [tenantId, setTenantId] = useState("");
const [name, setName] = useState("");
const [themeColor, setThemeColor] = useState("#4f46e5");

const handleCreate = async () => {
  if (!tenantId || !name) return alert("Tenant ID and Name are required");
  try {
    const res = await fetch("http://localhost:5000/api/tenants/register", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ tenantId, name, themeColor })
    });
    const data = await res.json();
    alert(data.message || "Done");
  } catch {
    alert("Error creating tenant");
  }
};

return (
  <div style={{ padding: 20 }}>
    <h2>Register Tenant</h2>
    <input
      placeholder="Tenant ID (e.g., tenant3)"
      value={tenantId}
      onChange={(e) => setTenantId(e.target.value)}
    /><br /><br />
    <input
      placeholder="Tenant Name"
      value={name}
```

```
        onChange={(e) => setName(e.target.value)}
      /><br /><br />
      <label>Theme Color: </label>
      <input
        type="color"
        value={themeColor}
        onChange={(e) => setThemeColor(e.target.value)}
      /><br /><br />
      <button onClick={handleCreate}>Create Tenant</button>
    </div>
  );
}
```

frontend/src/pages/Dashboard.jsx

```
import { useEffect, useState } from "react";

export default function Dashboard() {
  const [tenantInfo, setTenantInfo] = useState(null);
  const [resources, setResources] = useState([]);
  const [newResource, setNewResource] = useState("");

  const token = localStorage.getItem("token");
  const tenant = localStorage.getItem("tenant");

  if (!token || !tenant) window.location.href = "/login";

  const BASE_URL = "http://localhost:5000";

  useEffect(() => {
    fetch(`${BASE_URL}/api/tenants`, {
      headers: { Authorization: `Bearer ${token}` }
```

```
  })
    .then((r) => r.json())
    .then((d) => {
      setTenantInfo(d);
      const p = d?.theme?.primaryColor || "#eee";
      const s = d?.theme?.secondaryColor || "#111";
      document.body.style.background = p;
      document.body.style.color = s;
    });
}, []);

useEffect(() => {
  fetch(`${BASE_URL}/api/resources`, {
    headers: { Authorization: `Bearer ${token}` }
  })
    .then((r) => r.json())
    .then(setResources);
}, []);

const addResource = async () => {
  if (!newResource) return;
  const res = await fetch(`${BASE_URL}/api/resources`, {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      Authorization: `Bearer ${token}`
    },
    body: JSON.stringify({ name: newResource })
  });
  const item = await res.json();
```

```
    setResources([item, ...resources]);
    setNewResource("");
  };

  return (
    <div style={{ padding: 20 }}>
      <h2>Dashboard</h2>
      {tenantInfo && <p><b>Tenant:</b> {tenantInfo.name}</p>}
      <h3>Add Resource</h3>
      <input
        value={newResource}
        onChange={(e) => setNewResource(e.target.value)}
        placeholder="Resource name"
      />
      <button onClick={addResource}>Add</button>

      <h3>Resources</h3>
      <ul>
        {resources.map((r) => <li key={r._id}>{r.name}</li>)}
      </ul>

      <br />
      <button onClick={() => { localStorage.clear();
window.location.href="/login"; }}>
        Logout
      </button>
    </div>
  );
}
```

frontend/src/pages/AdminTheme.jsx

```
import { useEffect, useState } from "react";

export default function AdminTheme() {
  const token = localStorage.getItem("token");
  if (!token) window.location.href = "/login";

  const [theme, setTheme] = useState({
    primaryColor: "#4f46e5",
    secondaryColor: "#111827",
    font: "system-ui",
    logo: ""
  });

  const BASE_URL = "http://localhost:5000";

  useEffect(() => {
    fetch(`${BASE_URL}/api/tenants`, {
      headers: { Authorization: `Bearer ${token}` }
    })
      .then((r) => r.json())
      .then((d) => d?.theme && setTheme(d.theme));
  }, []);

  const save = async () => {
    const res = await fetch(`${BASE_URL}/api/tenants/theme`, {
      method: "PUT",
      headers: {
        "Content-Type": "application/json",
        Authorization: `Bearer ${token}`
```

```
    },
    body: JSON.stringify({ theme })
  });
  const data = await res.json();
  alert(data.message || "Saved");
 };

 return (
  <div style={{ padding: 20 }}>
   <h2>Admin Theme Editor</h2>

   <label>Primary Color</label><br />
   <input type="color" value={theme.primaryColor}
        onChange={(e) => setTheme({ ...theme, primaryColor: e.target.value
})} />
   <br /><br />

   <label>Secondary Color</label><br />
   <input type="color" value={theme.secondaryColor}
        onChange={(e) => setTheme({ ...theme, secondaryColor:
e.target.value })} />
   <br /><br />

   <label>Font</label><br />
   <input value={theme.font}
        onChange={(e) => setTheme({ ...theme, font: e.target.value })} />
   <br /><br />

   <label>Logo URL</label><br />
   <input value={theme.logo}
        onChange={(e) => setTheme({ ...theme, logo: e.target.value })} />
```

```
      <br /><br />

      <button onClick={save}>Save Theme</button>
    </div>
  );
}
```

References

1. MongoDB Documentation. *Database Design and Multi-Tenancy Concepts.* Retrieved from https://www.mongodb.com/docs/
2. Express.js Documentation. *Routing, Middleware, and REST API Development.* Retrieved from https://expressjs.com/
3. React.js Documentation. *Frontend Component Architecture and State Management.* Retrieved from https://react.dev/
4. Node.js Documentation. *JavaScript Runtime and Backend Development.* Retrieved from https://nodejs.org/en/docs/
5. Vite Documentation. *React Development and Build Optimization.* Retrieved from https://vitejs.dev/guide/
6. JSON Web Tokens (JWT). *Authentication and Authorization Techniques.* Retrieved from https://jwt.io/
7. MDN Web Docs. *JavaScript, HTTP Protocol, and Web Security Guidelines.* Retrieved from https://developer.mozilla.org/
8. GitHub Documentation. *Version Control and Repository Management.* Retrieved from https://docs.github.com/
9. REST API Best Practices. *API Structure, Endpoints, and Data Handling.* Retrieved from https://restfulapi.net/

Bibliography

1. Fowler, M. (2002). *Patterns of Enterprise Application Architecture.* Addison-Wesley.
2. Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems.* O'Reilly Media.
3. Subramanian, V. *Node.js: The Right Way – Practical Server-Side JavaScript.* Pragmatic Bookshelf.
4. Dayley, B. (2014). *Learning MongoDB.* Addison-Wesley.
5. Freeman, A. (2019). *Pro React.* Apress.
6. Sharma, S. (2021). *Full-Stack Web Development with React and Node.* Packt Publishing.