📘 Product Requirements Document (PRD)

Multi-Account System (SUPPLIER / COMPANY / VEHICLE)

✂️ Overview

Hum ek multi-tenant SaaS product bana rahe hain jisme koi bhi user signup kar sakta hai.

System me 3 primary account types honge: - SUPPLIER - COMPANY - VEHICLE

Har account ka apna isolated system hoga.

👉 Basic principle: - Ek account = apna data - Dusre account ka data accessible nahi hoga - Har account ke andar OWNER + STAFF users honge

---

✒️ Core Concepts (Important)

🔗 2.1 Account (Tenant)

Account ek independent system hai.

Account properties: - accountId (unique) - accountType (SUPPLIER / COMPANY / VEHICLE) - email - name - status (active/inactive)

🔗 2.2 User Types

Har account me 2 type ke users honge:

1. OWNER (Main user / Admin)
2. STAFF (Sub users)

🛏️ Email Rules (Modified Logic 📀)

🔗 3.1 Account Level Email Rule

Ek email se: - SUPPLIER account ban sakta hai - COMPANY account ban sakta hai - VEHICLE account ban sakta hai

BUT 👇

👊🏿 Same email + same accountType allowed nahi hai.

Example:

Email	Account Type	Allowed?
a@g.com	SUPPLIER	👍🏽 Allowed
a@g.com	SUPPLIER	👆 Not Allowed
a@g.com	COMPANY	👍🏽 Allowed
a@g.com	VEHICLE	👍🏽 Allowed
a@g.com	COMPANY (again)	👆 Not Allowed

👉 Rule:

```
One Email = One Account per Account Type
```

🔗 3.2 Staff Email Rule

👊🏿 Staff ke liye bhi same rule:

- Same email ek hi account me repeat nahi ho sakta
- Same email dusre account me allowed ho sakta hai

Example:

Email	AccountId	Allowed?
staff@g.com	SupplierAccount1	👍🏽 Allowed
staff@g.com	SupplierAccount1	👆 Not Allowed
staff@g.com	CompanyAccount1	👍🏽 Allowed

🌂 Role System

🔗 4.1 Roles

Role	Description
OWNER	Account ka main admin
STAFF	Limited access user

🔗 4.2 OWNER Permissions

OWNER ke paas full access hoga: - Account data manage - Staff create / update / delete - Supplier / Company / Vehicle / Trip CRUD - Permissions assign to staff

🔗 4.3 STAFF Permissions

STAFF ke permissions OWNER decide karega:

Modules: - Supplier - Company - Vehicle - Trip

Actions: - create - read - update - delete

Example permission structure:

```
{
  "supplier": { "create": true, "read": true, "update": false, "delete":
false },
  "company": { "create": false, "read": true, "update": false, "delete":
false },
  "vehicle": { "create": true, "read": true, "update": true, "delete": false },
  "trip": { "create": true, "read": true, "update": true, "delete": true }
}
```

🌡️ Account Type Behavior

🔗 5.1 SUPPLIER Account

SUPPLIER OWNER can: - Staff create / update / delete - Supplier / Company / Vehicle / Trip CRUD - Apna account edit (update)

SUPPLIER OWNER cannot: - Apna account delete

SUPPLIER STAFF can: - Only allowed permissions

🔗 5.2 COMPANY Account

COMPANY OWNER can: - Staff create / update / delete - Company / Supplier / Vehicle / Trip CRUD - Apna account edit

COMPANY OWNER cannot: - Apna account delete

COMPANY STAFF can: - Only allowed permissions

🔗 5.3 VEHICLE Account

VEHICLE OWNER can: - Staff create / update / delete - Vehicle / Supplier / Company / Trip CRUD - Apna account edit

VEHICLE OWNER cannot: - Apna account delete

VEHICLE STAFF can: - Only allowed permissions

🎒 Data Isolation Rule (Most Important 🔐)

👉 Har table me accountId mandatory hoga.

Example schemas:

Supplier Table

- accountId
- supplierName
- details

Company Table

- accountId
- companyName
- details

Vehicle Table

- accountId
- vehicleNumber
- details

Trip Table

- accountId
- tripDetails

Rule:

```
User sirf apne accountId ka data access kar sakta hai.
```

Example query:

```
Model.find({ accountId: req.user.accountId })
```

👉 Result: - Supplier account ka data = sirf supplier ko dikhega - Company account ka data = sirf company ko dikhega - Vehicle account ka data = sirf vehicle ko dikhega

---

🎓 Authentication & Authorization Flow

🔗 7.1 Register Flow

1. User selects accountType (SUPPLIER / COMPANY / VEHICLE)
2. System checks:
3. Email + accountType already exists?
4. If not exists:
5. Create Account
6. Create OWNER user

🔗 7.2 Login Flow

1. User login with email + password + accountType
2. System validates credentials
3. JWT token generated with:
4. userId
5. accountId
6. role
7. permissions

🔗 7.3 Staff Flow

1. OWNER creates staff
2. Assign permissions
3. Staff login with same flow

🎩 Database Design (Summary)

Tables / Collections:

1. Accounts
2. Users
3. Suppliers
4. Companies
5. Vehicles

6. Trips
7. RefreshTokens (optional)

🎽 Future Scalability (Optional)

Possible future features: - Super Admin Panel - Subscription Plans - Audit Logs - Activity History - Multi-branch accounts - Role Templates - Module-wise access control

---

📀 Final Principle (Golden Rule)

```
1 Account = 1 Isolated System
1 Email = 1 Account per Account Type
OWNER = Full Control
STAFF = Controlled Access
No Data Sharing Between Accounts
```

🧰 OPTION 1: Technical Design (Database & System Design)

✂️ Database Collections / Tables

1. Accounts

Fields: - _id (ObjectId) - accountType (SUPPLIER | COMPANY | VEHICLE) - name - email - status (active/inactive) - createdAt, updatedAt

Unique Index: - (email + accountType) must be unique

---

2. Users (Owner + Staff)

Fields: - _id - accountId (ref: Accounts) - name - email - password - role (OWNER | STAFF) - permissions (JSON) - isActive - createdAt, updatedAt

Unique Index: - (email + accountId) must be unique

---

3. Supplier Data

Fields: - _id - accountId - supplierName - details

---

4. Company Data

Fields: - _id - accountId - companyName - details

---

5. Vehicle Data

Fields: - _id - accountId - vehicleNumber - details

---

6. Trip Data

Fields: - _id - accountId - tripDetails - supplierId - companyId - vehicleId

---

✒️ Core Technical Rules

1. Har collection me accountId mandatory
2. Query me always accountId filter
3. Unique constraints strictly follow honge
4. Permissions JSON based honge
5. JWT token me accountId mandatory

💣 OPTION 2: Backend Architecture (System Flow)

✂️ Folder Structure (Recommended)

```
src/
 ├── config/
 ├── controllers/
 ├── models/
 ├── routes/
 ├── middlewares/
 ├── services/
 ├── utils/
 ├── app.js
 └── server.js
```

✒️ Authentication Flow

Register Flow

1. User selects accountType
2. Check (email + accountType) exists?
3. If not exists → create Account
4. Create OWNER user
5. Return tokens

Login Flow

1. Input: email + password + accountType
2. Find user by accountId + email
3. Validate password
4. Generate JWT token
5. Return user + permissions

Staff Flow

1. OWNER creates staff
2. Assign permissions
3. Staff login
4. Permissions checked via middleware

🛏️ Authorization Architecture

Middleware Layers

1. Auth Middleware (JWT verify)
2. Account Middleware (accountId check)
3. Role Middleware (OWNER / STAFF)
4. Permission Middleware (CRUD access)

Example:

```
router.post(
  "/vehicle",
  authMiddleware,
  permissionMiddleware("vehicle", "create"),
```

```
  createVehicle
);
```

🌂 Multi-Tenant Isolation

Rule:

```
Every API must filter data by accountId
```

Example:

```
const vehicles = await Vehicle.find({ accountId: req.user.accountId });
```

💻 OPTION 3: Real Code Blueprint (Implementation)

✂️ Account Registration API

```
export const registerAccount = async (req, res) => {
  const { name, email, password, accountType } = req.body;

  const existingAccount = await Account.findOne({ email, accountType });
  if (existingAccount) {
    return res.status(400).json({ message: "Account already exists" });
  }

  const hashedPassword = await bcrypt.hash(password, 12);

  const account = await Account.create({ name, email, accountType });

  const owner = await User.create({
    accountId: account._id,
    name,
    email,
    password: hashedPassword,
    role: "OWNER",
    permissions: {
      supplier: { create: true, read: true, update: true, delete: true },
```

```
      company: { create: true, read: true, update: true, delete: true },
      vehicle: { create: true, read: true, update: true, delete: true },
      trip: { create: true, read: true, update: true, delete: true },
    }
  });

  res.status(201).json({ message: "Account created", account, owner });
};
```

✒️ Staff Creation API

```
export const createStaff = async (req, res) => {
  if (req.user.role !== "OWNER") {
    return res.status(403).json({ message: "Only owner can create staff" });
  }

  const { name, email, password, permissions } = req.body;

  const existingStaff = await User.findOne({
    email,
    accountId: req.user.accountId
  });

  if (existingStaff) {
    return res.status(400).json({ message: "Staff already exists" });
  }

  const hashedPassword = await bcrypt.hash(password, 12);

  const staff = await User.create({
    accountId: req.user.accountId,
    name,
    email,
    password: hashedPassword,
    role: "STAFF",
    permissions
  });

  res.status(201).json({ message: "Staff created", staff });
};
```

🛏️ Permission Middleware

```
export const checkPermission = (module, action) => {
  return (req, res, next) => {
    const permission = req.user.permissions[module];

    if (!permission || !permission[action]) {
      return res.status(403).json({ message: "Access denied" });
    }

    next();
  };
};
```

🌂 Data Isolation Example

```
export const getVehicles = async (req, res) => {
  const vehicles = await Vehicle.find({ accountId: req.user.accountId });
  res.json(vehicles);
};
```

🔗 Summary

Tumhara system 3 layers pe kaam karega:

🤩 Technical Design → Database + Rules 🤪 Backend Architecture → Flow + Middleware 🤫Real Code Blueprint → Actual implementation

--- (Golden Rule)

```
1 Account = 1 Isolated System
1 Email = 1 Account per Account Type
OWNER = Full Control
STAFF = Controlled Access
No Data Sharing Between Accounts
```

🔗 This document will be used as:

- Product blueprint
- Backend architecture guide
- Future reference
- Team documentation

💡 Note

Agar future me tum chaho to: - Is document ko technical architecture me convert kiya ja sakta hai - API design banaya ja sakta hai - Database ER diagram banaya ja sakta hai - Production-ready system design kiya ja sakta hai

---

✍️ Created for your product vision 😰