Do you think this is a good architecture with a scalable design? What can be the trade-offs with the tech stack and design choices? Please feel free to explain briefly if you can design something better. We would love to hear that.

Answer :

Tech Stack Used

For this assignment, I used:

Backend

- **Node.js (Express.js):** Easy to structure APIs, fast to prototype, and fits well with JSON-based data flows.
- **MongoDB**
- **bcrypt:** For hashing admin passwords securely.
- **JWT:** For admin authentication and token-based authorization.
- **CORS & cookie-parser:** To handle browser-based API requests safely.

Frontend

- **HTML, CSS, and Vanilla JavaScript:**
  This keeps the UI simple, dependency-free, and easy to test the assignment endpoints. It also avoids unnecessary complexity that a frontend framework might introduce for a small demo.

From an engineering point of view, the architecture we used for this assignment is perfectly fine for a small to medium multi-tenant setup. The basic idea is simple: store all organization metadata in a master database and create a separate collection for each organization. It works well for quick onboarding, it's easy to maintain in the early stages.

However, when we start thinking about scalability and long-term growth, this design does come with a few trade-offs. A collection-per-customer model starts to get harder to manage once the number of organizations grows. Indexing, schema changes, backups, migrations, and even cluster performance can become more complex because everything sits inside one database even though each org is isolated at the collection level.

Trade-offs with the current approach

- **Operational overhead increases** when many collections exist inside one DB.
- **Schema updates must be repeated** across all tenant collections.
- **Reporting or analytics across tenants is harder** because data is physically separated.
- **Performance isolation is limited** a heavy customer can still impact others, since everything runs on the same DB resources.
- **Renaming organizations requires copying data**, which can become expensive over time.

My Suggestion: One Database per Customer (Same Cluster)

If I were designing this for a more production-ready system, I would personally use **one database per customer but still keep everything inside the same MongoDB cluster**.

This improves isolation significantly without adding too much complexity. Each organization gets:

/org_master_db

/org_customer1_db

/org_customer2_db

/org_customer3_db

...

Why this is better

- **Cleaner isolation:** One misconfigured query cannot leak data across tenants.
- **Easier backups and restores:** Each customer's DB can be archived or restored individually.
- **Better long-term maintainability:** If schemas or indexes change, each tenant DB can be upgraded independently.
- **Better performance boundaries:** Heavy tenants can be throttled or moved more easily.
- **Still simple to operate:** MongoDB clusters can comfortably handle many databases.

This provides a more scalable tenant-isolation model. At the same time, it avoids the operational complexity of running separate clusters for every customer.

In short

The architecture implemented here is completely fine for the assignment and demonstrates multi-tenant concepts clearly.
But if this system were to evolve into a real SaaS product, I would move toward **database-per-tenant inside the same cluster**, since it gives a stronger separation of data, easier maintenance, and better scalability while keeping costs and operational overhead manageable.