The Weeding Company

ADDITIONAL QUESTION

What’s good about your current design

- **Clear separation**: a *master* DB for global metadata (orgs + admins) and *tenant* collections for each org. This keeps global data centralized and tenant data isolated.
- **Dynamic tenant creation**: creating `org_<name>` collections programmatically is simple and works for many use cases.
- **JWT auth**: stateless tokens are easy to scale and work well for APIs.
- **BCrypt for passwords**: good practice — passwords are hashed, not stored plain.

Main trade-offs and limitations (what to watch out for)

1. **Single MongoDB instance (single database)**
    - *Pro:* Easy to implement and operate for small projects.
    - *Con:* Single point of failure and potential performance bottleneck as tenants and data grow.
2. **One collection per org in the same DB**
    - *Pro:* Logical separation, easy to query per-tenant.
    - *Con:* If you have thousands of orgs, thousands of collections can stress the server and increase metadata overhead. Backup/restore of a single tenant is harder.
3. **Auth & RBAC handled manually in controllers**
    - *Pro:* Simple and explicit.
    - *Con:* Harder to enforce consistently as you add endpoints. Also repetitive code. A centralized JWT filter/middleware is preferable.

Practical suggestions to improve

Short-term

1. **Centralized JWT filter**
   - Add a Spring Security filter that validates tokens and attaches claims to request context — controllers read from context rather than parsing the token repeatedly.
2. **Create standard template + validation**
   - Use MongoDB schema validation or a JSON Schema per tenant to avoid garbage data.

Medium-term (scale & safety)

1. **Shard or split tenants**
   - For many tenants or large tenants, consider:
     - grouping multiple tenants per database (DB-per-100-tenants), or
     - using separate databases for very large tenants (db-per-tenant for big customers).
2. **Move to replica sets + backup strategy**
   - Use MongoDB replica sets for high availability and scheduled backups for disaster recovery.

Longer-term (enterprise)

1. **Multi-region deployments**
   - Use geo-aware routing and regional DB replicas for low latency.
2. **Policy / billing / audit**
   - Add audit logs per tenant, quota management, and billing hooks.

Summary

The current design is a good starting point: it keeps global metadata in a master database and creates separate tenant collections dynamically. That makes the code simple and easy to test. For production, I'd add a centralized JWT filter, schema validation, and move MongoDB to a replica set. If we expect thousands of tenants or very large customers, consider a hybrid model (shared collections for small tenants, dedicated DB or cluster for big tenants). These steps give us better availability, clearer isolation, and easier operational control.