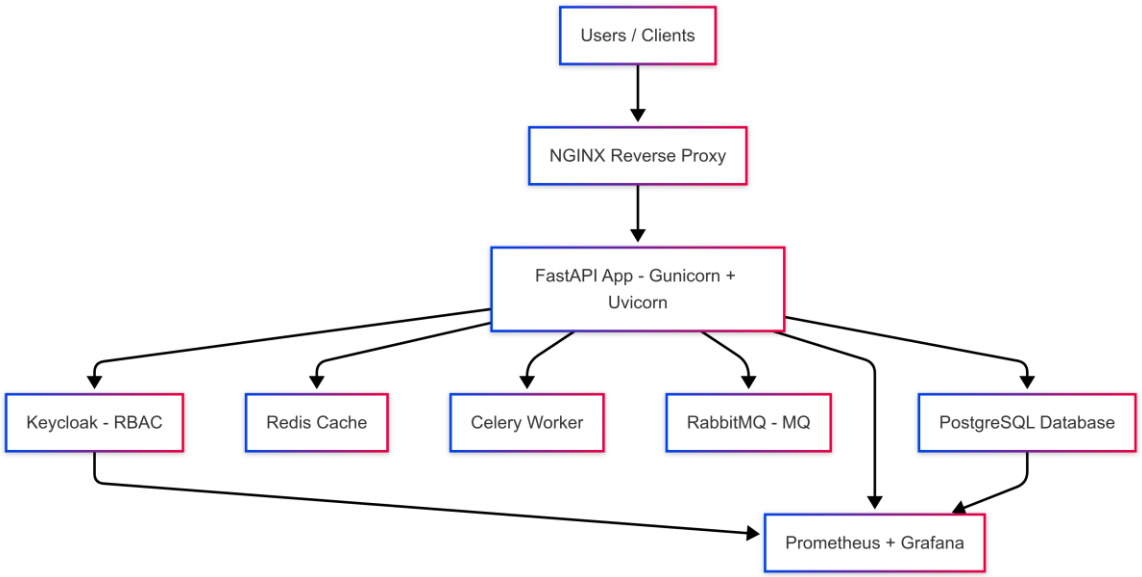
Task Management System - Application Architecture Design

Task Management System - Application Architecture Design

1. Overview

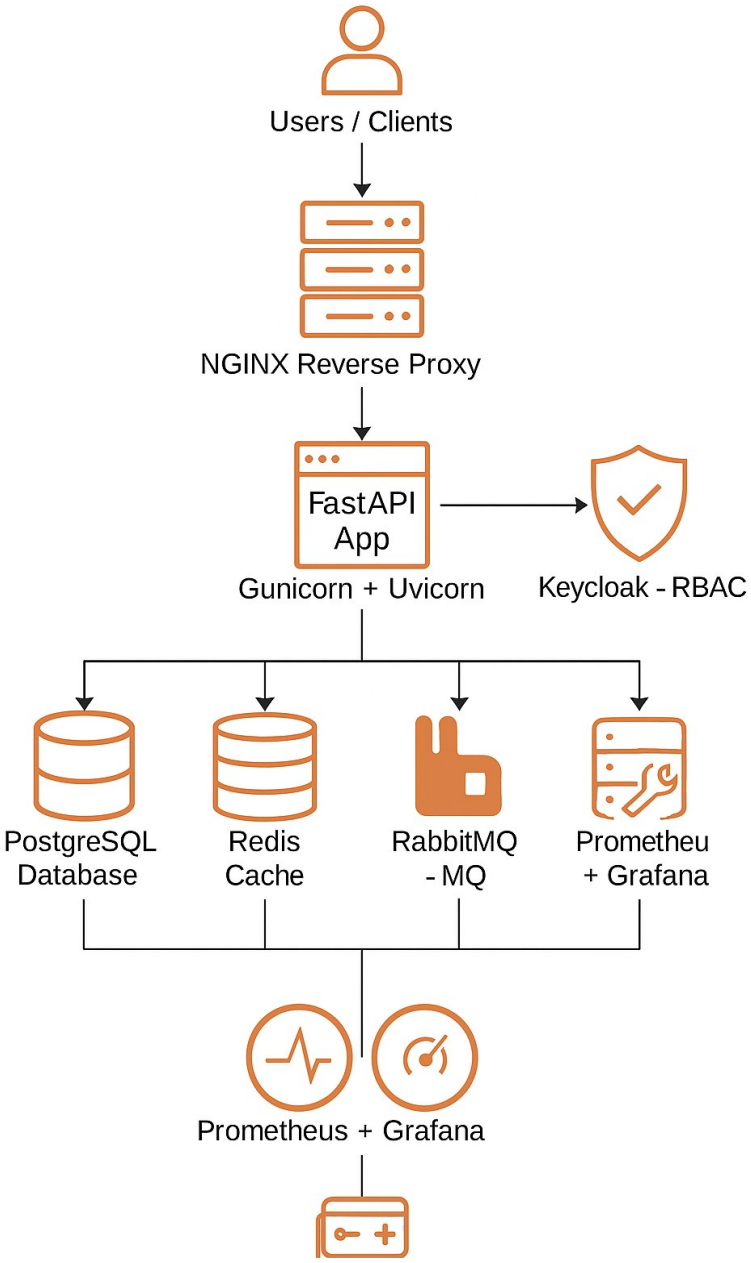
FastAPI-based **Task Management System**

- high traffic,
- concurrent users,
- and scalable operations.
- It emphasizes
    - modularity,
    - high availability (HA),
    - performance,
    - observability,
    - and secure role-based access control (RBAC) using Keycloak



---

2. High-Level Architecture Diagram



---

3. Components and Design

3.1 FastAPI App

- **Asynchronous**: Uses `async def` where needed.
- **Gunicorn + Uvicorn Workers**: To handle concurrent requests.
- **Dockerized**: Enables container-based deployments.
- **Endpoints**: CRUD for tasks, health checks, etc.

3.2 PostgreSQL

- Hosted in HA mode using cloud-native setups or via `Patroni`/`TimescaleDB`.
- Use UUID for task IDs for distribution friendliness.
- Write-heavy operations scaled via **read replicas**.
- ORM: SQLAlchemy + Alembic for migrations.

3.3 Keycloak for RBAC

- Keycloak Docker container.
- Realm: `task-manager`.
- Clients: `fastapi-backend`.
- Roles: `admin`, `user`, etc.
- Users authenticate via OAuth2/OpenID Connect.
- FastAPI uses `fastapi-keycloak` or JWT middleware for verifying scopes and roles.

3.4 Redis (Caching Layer)

- Cache frequently accessed tasks (e.g., GET by ID).
- Use TTL-based expiration for cache invalidation.

3.5 Celery + RabbitMQ

- Offload background jobs like audit logs, analytics, or notifications.
- Scales independently from web workers.

3.6 NGINX

- Reverse proxy.
- SSL termination.
- Load balancing across Gunicorn workers or multiple FastAPI pods.

3.7 Prometheus + Grafana

- **Prometheus**: Scrapes metrics from FastAPI app.
- **Grafana**: Visualizes FastAPI health, latency, DB connections, cache hit/miss, etc.
- Logs centralized via Loki or Fluentd to Elasticsearch.

---

4. Scalability Plan

- FastAPI behind NGINX, auto-scaled with Kubernetes HPA.

- PostgreSQL with read replicas and WAL archiving.

- Redis horizontally scalable via clustering.

- RabbitMQ clustered or replaced with Kafka if scale increases.

---

5. Caching Strategy

- Use Redis to cache GET `/tasks/{id}` responses.

- Use `task_id` as cache key.

- Invalidate/update cache on `GET UPDATE` and `DELETE`.

- Use `async` Redis client like `aioredis`.

---

6. Message Queuing (Async Tasks)

- Celery workers perform async logging, notifications, analytics.

- RabbitMQ used as the message broker.

- Retry mechanisms and DLQs (dead-letter queues) for failed tasks.

---

7. Security Considerations

- OAuth2-based token validation using Keycloak.

- Role-specific access with scopes (e.g., admin-only delete).

- HTTPS enforced at NGINX.

- SQL injection and XSS protections via ORM + FastAPI validation.

---

8. Deployment Strategy

- Docker Compose for local setup.

- Kubernetes-ready Helm charts for production.

- Environment variables via `.env` + Docker secrets.

- CI/CD pipeline using GitHub Actions.

---

9. Health Checks & Observability

- `/health` endpoint to validate DB connection.

- `/metrics` endpoint exposed for Prometheus.

- Alerting rules for API failures, high response time, low DB availability.

---

10. Future Improvements

- Integrate Sentry for exception monitoring.

- Use Kafka for event streaming.

- Add API rate limiting via Redis token bucket.

- RBAC per-resource instead of per-endpoint.