ChatGPT

Arquitetura Atualizada do Sistema de Gestão de Igrejas

Visão Geral da Arquitetura

O sistema segue arquitetura de microsserviços (Auth, Church, Event, etc.), agora executando em contêineres via Docker Compose. Cada serviço é isolado em seu próprio repositório/contexto (ex: pastas `Auth/`, `Church/`, etc.) contendo código, Dockerfile e configurações específicas. Isso facilita builds isolados e o deploy modular de cada contêiner. O front-end Vue.js é servido por um container NGINX, consumindo as APIs dos microsserviços internos. O roteamento de requisições externas passa pelo NGINX (porta 80) diretamente no VPS. Em resumo, toda a solução foi reorientada para um ambiente Docker em VPS Linux, simplificando o deploy e a escalabilidade.

Microsserviços e Estrutura de Pastas

- **Subdiretórios por contexto:** Cada domínio funcional tem sua pasta raiz (ex: `Auth/`, `Church/`, `Event/`, `Shared/` etc.), contendo APIs, modelos de domínio e Dockerfile. Por exemplo, `Auth/` abriga `Auth.API/`, `Auth.Domain/`, etc. Isso isola os serviços e permite builds de imagens independentes.
- **Build e Deploy:** Cada Dockerfile usa o diretório do serviço como *build context*. No Docker Compose, referenciamos essas pastas para construir as imagens (ex.: `build: { context: ./Auth/Auth.API }`). Assim, ao executar `docker compose up -d`, cada serviço é containerizado separadamente seguindo essa estrutura de pastas.

Banco de Dados (PostgreSQL)

Substituímos completamente o MySQL por **PostgreSQL** como banco de dados principal. As vantagens incluem:
- **Integração .NET:** Há um provedor ADO.NET (Npgsql) oficial que permite às aplicações C# acessarem o PostgreSQL de forma nativa [1]. Existe também um *provider* do Entity Framework Core específico para Postgres, expondo recursos avançados deste SGBD [1].
- **Tipos de dados complexos:** Postgres suporta tipos complexos (arrays, tipos compostos, hstore, `JSONB`, `XML`, etc.), permitindo modelar dados ricos sem perder performance [2]. Isso facilita armazenar documentos ou dados não-relacionais integrados ao banco.
- **Extensibilidade e Índices:** Postgres é altamente extensível (ex.: extensões como PostGIS, TimescaleDB) e oferece índices avançados (GIN/GiST) que aceleram buscas textuais ou em JSON [2].
- **Conformidade e Transações:** Cumpre rigorosamente padrões ANSI SQL e oferece ACID completo com MVCC (controle de concorrência multi-versão). Isso garante integridade mesmo sob alta concorrência [3].
- **Performance:** Embora cada caso varie, Postgres escala bem em leituras e suporta particionamento e paralelismo. Em especial, gerencia eficientemente cargas concorrentes altas graças ao MVCC [3].

Para **persistência de dados**, cada serviço que usa banco (Auth, Church, Event) aponta para um servidor PostgreSQL único ou replica. No Docker Compose, usamos volumes nomeados para o Postgres (ex.: `postgres_data:/var/lib/postgresql/data`) garantindo que os dados sobrevivam a reinícios do

container [4] [5]. Para backups, substituímos ferramentas MySQL (ex.: mysqldump) por equivalentes no Postgres (ex.: `pg_dump`, `pg_basebackup`). Não há mais referência a InnoDB ou engines MySQL, pois o armazenamento agora usa o formato nativo do PostgreSQL.

Deploy com Docker Compose em Linux VPS

O ambiente de produção é um VPS Ubuntu 24.04 (2 vCPU, 12 GB RAM). O sistema é orquestrado por **Docker Compose**, com um arquivo `docker-compose.yml` unificando todos os serviços (microsserviços .NET, banco PostgreSQL, Redis, RabbitMQ, NGINX+Vue). No Compose definimos *resources* para cada serviço, aplicando **limites de memória** via `deploy.resources.limits.memory` (ex.: `memory: 512M`) [6]. Isso evita que um container exija toda a RAM disponível e garante estabilidade. Também usamos **volumes persistentes** para o Postgres e RabbitMQ (ex.: `postgres_data`, `rabbitmq_data`) montados nos diretórios de dados internos, assegurando persistência de dados [4] [5]. O NGINX é configurado para expor a porta 80 do host (por exemplo, `ports: ["80:80"]`), servindo o front-end Vue compilado [7]. Após configurar, sobe-se tudo com `docker compose up -d`, que builda as imagens e inicia todos os contêineres em background.

```
version: '3.8'
services:
  auth:
    build:
      context: ./Auth/Auth.API
      dockerfile: Dockerfile
    container_name: auth-service
    environment:
      -
ConnectionStrings__AuthDB=Host=postgres;Database=AuthDB;Username=admin;Password=secret
    deploy:
      resources:
        limits:
          memory: 512M
    depends_on:
      - postgres

  church:
    build:
      context: ./Church/Church.API
      dockerfile: Dockerfile
    container_name: church-service
    environment:
      -
ConnectionStrings__ChurchDB=Host=postgres;Database=ChurchDB;Username=admin;Password=secret
    deploy:
      resources:
        limits:
          memory: 512M
    depends_on:
      - postgres

  event:
```

```
    build:
      context: ./Event/Event.API
      dockerfile: Dockerfile
    container_name: event-service
    environment:
      -
ConnectionStrings__EventDB=Host=postgres;Database=EventDB;Username=admin;Password=secret
    deploy:
      resources:
        limits:
          memory: 512M
    depends_on:
      - postgres

  postgres:
    image: postgres:16
    container_name: postgres-db
    restart: always
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secret
    volumes:
      - postgres_data:/var/lib/postgresql/data
    deploy:
      resources:
        limits:
          memory: 1024M

  redis:
    image: redis:7-alpine
    container_name: redis-cache
    restart: always
    deploy:
      resources:
        limits:
          memory: 256M

  rabbitmq:
    image: rabbitmq:3-management
    container_name: rabbitmq-mq
    restart: always
    ports:
      - "5672:5672"
      - "15672:15672"  # management UI
    environment:
      RABBITMQ_DEFAULT_USER: user
      RABBITMQ_DEFAULT_PASS: pass
    volumes:
      - rabbitmq_data:/var/lib/rabbitmq/
      - rabbitmq_log:/var/log/rabbitmq/
    deploy:
```

```
      resources:
        limits:
          memory: 512M

  frontend:
    build:
      context: ./Frontend
      dockerfile: Dockerfile
    container_name: vue-frontend
    depends_on:
      - auth
      - church
      - event

  nginx:
    image: nginx:alpine
    container_name: nginx-proxy
    ports:
      - "80:80"
    volumes:
      - ./nginx/default.conf:/etc/nginx/conf.d/default.conf
      - frontend_dist:/usr/share/nginx/html
    depends_on:
      - frontend

volumes:
  postgres_data:
  rabbitmq_data:
  rabbitmq_log:
  frontend_dist:
```

- **Limites de memória:** Cada serviço tem `deploy.resources.limits.memory` definido (ex: 512M ou 1024M) para evitar consumo excessivo e garantir performance estável [6] [8].
- **Volumes persistentes:** Declaramos volumes nomeados (`postgres_data`, `rabbitmq_data`, etc.) e os montamos nos caminhos de dados internos. Isso preserva o estado mesmo que os contêineres sejam recriados [4] [5].
- **Porta do NGINX:** Expondo a porta 80, o NGINX serve o conteúdo compilado do Vue no host. No exemplo acima, usa-se `ports: ["80:80"]` no serviço NGINX [7].
- **Subida dos contêineres:** Com o arquivo `docker-compose.yml` pronto, o sistema é iniciado via comando simples:

  ```
  docker compose up -d
  ```

  Isso constroi as imagens (se necessário) e levanta todos os serviços em segundo plano. Após isso, a aplicação fica acessível em `http://<IP-do-VPS>/` através do NGINX.

Infraestrutura (Linux VPS com Docker)

O deploy final ocorre em um servidor Ubuntu 24.04 (2 vCPU, 12 GB RAM). Instalamos Docker e Docker Compose (plugin). O sistema de arquivos do host não precisa conhecer detalhes internos dos

containers – o Docker Compose gerencia redes internas, volumes e ligações entre serviços. Configuramos *restart policies* (ex.: `restart: always`) para manter os serviços ativos após falhas. O NGINX (`nginx-proxy`) direciona as requisições HTTP para o front-end Vue construído, e o front-end consome as APIs dos microsserviços via rede interna do Docker. Em suma, toda a infra é orientada a contêineres, simplificando escalonamento futuro (bastaria ajustar réplicas no Compose) e garantindo que cada componente rode em ambiente isolado e consistente.

Fontes: Documentação oficial e artigos recentes sobre Docker Compose e PostgreSQL foram usados para fundamentar esta configuração [1] [2] [6] [4] [7].

---

[1] Npgsql - .NET Access to PostgreSQL | Npgsql Documentation
https://www.npgsql.org/

[2] [3] PostgreSQL Advantages and Disadvantages 2024 : Aalpha
https://www.aalpha.net/blog/pros-and-cons-of-using-postgresql-for-application-development/

[4] [5] How to Run RabbitMQ in Docker Compose | by Kaloyan Manev | Medium
https://medium.com/@kaloyanmanev/how-to-run-rabbitmq-in-docker-compose-e5baccc3e644

[6] Compose Deploy Specification | Docker Docs
https://docs.docker.com/reference/compose-file/deploy/

[7] Deploying a Vue.js Application with Docker and Nginx | by Prasuna Mudawari | Cloud Engineer | Medium
https://medium.com/@prasunamudawari/deploying-a-vue-js-application-with-docker-and-nginx-387fef1a27f2

[8] How to Configure Memory Limits in Docker Compose: A Comprehensive Guide - GeeksforGeeks
https://www.geeksforgeeks.org/configure-docker-compose-memory-limits/