ChatGPT

Visão Geral da Arquitetura

O sistema de gestão de igrejas é organizado como um conjunto de microsserviços .NET que interagem entre si, usando **.NET Aspire** para orquestração local e facilidades de infraestrutura. Cada microsserviço é uma aplicação ASP.NET Core com responsabilidades específicas. Os serviços principais são: **Auth.API** (autenticação/identidade), **Church.API** (dados da igreja), **Member.API** (gerenciamento de membros), **Content.API** (conteúdo e publicações), **Events.API** (eventos e agendas), **Finance.API** (finanças/doações) e **Counseling.API** (aconselhamento e agendamentos). Essa abordagem segue o princípio de *domain-driven design*, onde cada serviço mantém seu próprio domínio, banco de dados e lógica de negócio, garantindo desacoplamento.

Todos os projetos seguem a mesma estrutura genérica (camadas *API*, *Application*, *Domain*, *Infrastructure*) dentro de uma solução única. Além disso, há um projeto compartilhado **ServiceDefaults**, contendo configurações e extensões comuns (resiliência, logging, serviço de descoberta, etc.), e um projeto de orquestração **AppHost** (Aspire AppHost), que executa todos os serviços em desenvolvimento. O AppHost provê *service discovery*, injeção de variáveis de ambiente e inicialização consistente de recursos (bancos de dados, caches, filas) [1] [2]. Durante o desenvolvimento, o Aspire lança um painel (dashboard) visual mostrando todos os serviços e recursos em execução. Por exemplo, a imagem abaixo ilustra o painel de recursos, com um container de Redis (`cache`) e dois serviços em execução: `apiservice` e `webfrontend`, cada um expondo endpoints HTTP locais.

Figura – Dashboard .NET Aspire exibindo recursos (serviços e containers) ativos durante o desenvolvimento local.

Cada serviço é exposto via HTTP REST (controladores ASP.NET Core) e também pode publicar/consumir mensagens assíncronas usando filas RabbitMQ. Internamente, a comunicação síncrona entre APIs usa HttpClient com *service discovery* automático configurado pelo Aspire [1] [3], evitando hardcode de URLs. Já a integração assíncrona é feita via mensageria RabbitMQ (fila de eventos), desacoplando fluxos de trabalho (por exemplo, publicar eventos **MemberRegistered** ou **DonationReceived** para outros serviços reagirem). O .NET Aspire provê integração hospedagem/cliente do RabbitMQ: no AppHost podemos usar `builder.AddRabbitMQ("messaging")` para subir um container RabbitMQ e referenciá-lo nos microsserviços [4]. Cada serviço pode então usar a biblioteca de cliente (por exemplo, `Aspire.RabbitMQ.Client` ou MassTransit) para enviar/receber mensagens, com as credenciais e conexão sendo configuradas automaticamente pelo Aspire.

Internamente, usamos um banco de dados isolado por serviço (por exemplo, PostgreSQL) e um cache Redis compartilhado para alta performance. Exemplos de configurações e detalhes de infraestrutura são discutidos nos tópicos abaixo. Neste documento, mantemos todos os nomes de projetos, serviços e tabelas em inglês, deixando apenas textos de interface no frontend em português. Os conceitos de *observabilidade* (telemetria, logs, tracing) e de *CI/CD* (GitHub Actions, Docker, Kubernetes) também serão detalhados, visando tornar este documento uma base técnica completa para o desenvolvimento e operação do sistema.

Microsserviços Principais

O sistema é dividido em sete microsserviços principais, cada um com uma responsabilidade bem definida:

- **Auth.API**: serviço de autenticação e autorização. Provê endpoints para login, geração de tokens JWT e gestão de usuários. Pode usar ASP.NET Identity ou similar para persistência de credenciais. Emita tokens que outros serviços validarão.
- **Church.API**: gerencia informações das igrejas (nomes, endereços, filiais, etc.). Serve dados organizacionais que podem ser usados por outros serviços (ex.: vincular membros a uma igreja).
- **Member.API**: gerencia membros da igreja (dados pessoais, associação a igrejas). Lida com cadastros, atualizações de perfil e relacionamentos de membros a igrejas e eventos.
- **Content.API**: gerencia conteúdo estático ou dinâmico (como notícias, artigos, blogs, mídia). Fornece CRUD de conteúdo que aparece no site da igreja.
- **Events.API**: gerencia eventos, cultos e agendamentos (datas, horários, inscrições). Pode coordenar com Church.API (local) e Member.API (inscrições de membros).
- **Finance.API**: lida com finanças e doações. Possui entidades para transações, orçamentos, doadores (talvez referenciando membros).
- **Counseling.API**: gerencia sessões de aconselhamento ou suporte (agendamentos, registros de atendimentos) entre membros e conselheiros.

Cada microsserviço segue uma **arquitetura em camadas** inspirada em Clean Architecture: há um projeto ASP.NET Core para a camada *API* (Controllers e Program.cs), um projeto *Application* para casos de uso (Commands/Queries e Handlers), um projeto *Domain* para entidades e lógica central, e um projeto *Infrastructure* para detalhes como contexto EF Core, repositórios, configuração de banco e de clientes externos. Por exemplo, para **Member.API** podemos ter os projetos `Member.API`, `Member.Application`, `Member.Domain`, `Member.Infrastructure`. Essa separação facilita testes unitários de cada camada e clareza de dependências.

Cada serviço tem seu próprio banco de dados (seguindo o princípio de base de dados por serviço), usando PostgreSQL (gratuito, compatível com .NET via Npgsql) em produção. Para testes e cenários locais simples pode-se usar SQLite in-memory. Usamos Entity Framework Core ou outra ORM leve na camada Infrastructure para acesso a dados. Em desenvolvimento com .NET Aspire, podemos usar `builder.AddPostgres("postgres")` no AppHost para provisionar um container Postgres e criar bancos de dados para cada serviço [5]. Por exemplo, no AppHost:

```
var builder = DistributedApplication.CreateBuilder(args);
var postgres = builder.AddPostgres("postgres").WithPgAdmin();
var authDb   = postgres.AddDatabase("AuthDB");
var churchDb = postgres.AddDatabase("ChurchDB");
builder.AddProject<Projects.Auth.API>("auth-api")
       .WithReference(authDb);
builder.AddProject<Projects.Church.API>("church-api")
       .WithReference(churchDb);
```

Isso cria um container Postgres (com pgAdmin) e disponibiliza conexões *AuthDB* e *ChurchDB* para os serviços. Os demais serviços seriam adicionados de forma similar.

Estrutura de Projetos e Pastas

A estrutura geral da solução deve seguir este padrão (exemplo adaptado de [21]):

```
ChurchManagement/
├── Auth.API/             # Projeto ASP.NET Core para Auth
│   ├── Controllers/
│   ├── Program.cs
│   ├── appsettings.json
│   └── ...
├── Auth.Application/     # Casos de uso, comandos e handlers
│   ├── Commands/
│   ├── Queries/
│   └── ...
├── Auth.Domain/          # Entidades e lógica de domínio
│   └── ...
├── Auth.Infrastructure/  # Conexão EF Core, repositórios, migrações
│   ├── Migrations/
│   └── AuthDbContext.cs
├── Church.API/           # Similar estrutura para Church
│   ├── Controllers/
│   └── ...
├── Church.Application/
├── Church.Domain/
├── Church.Infrastructure/
├── Member.API/          # E assim por diante para cada serviço (Member,
Content, Events, Finance, Counseling)
│   └── ...
├── Member.Application/
├── Member.Domain/
├── Member.Infrastructure/
├── Content.API/
├── Content.Application/
├── Content.Domain/
├── Content.Infrastructure/
├── Events.API/
├── Events.Application/
├── Events.Domain/
├── Events.Infrastructure/
├── Finance.API/
├── Finance.Application/
├── Finance.Domain/
├── Finance.Infrastructure/
├── Counseling.API/
├── Counseling.Application/
├── Counseling.Domain/
├── Counseling.Infrastructure/
├── ServiceDefaults/     # Projeto .NET (Class Library) com configurações
comuns
```

```
│   ├── ServiceDefaults.csproj
│   └── Extensions.cs   # Métodos de extensão para AddServiceDefaults etc.
└── ChurchManagement.AppHost/  # Orquestrador Aspire (console app)
    ├── ChurchManagement.AppHost.csproj
    └── Program.cs
```

Nesse esquema, a pasta **ServiceDefaults** contém um projeto compartilhado (classe biblioteca) usado por todas APIs. Ele define configurações e métodos de extensão comuns (ex.: métodos `AddServiceDefaults`, `ConfigureOpenTelemetry`, `AddDefaultHealthChecks` [6]). Todas as APIs referenciam esse projeto (via Project Reference) e chamam `builder.AddServiceDefaults()` em seu `Program.cs` para ativar as configurações padrão (telemetria, logging, health checks, serviço de descoberta, políticas de resiliência) [6]. O projeto **AppHost** é um executável que gerencia a orquestração local: seu arquivo `.csproj` usa o SDK `Aspire.AppHost.Sdk` e referencia `Aspire.Hosting.AppHost` [7]. Veja exemplo de `.csproj` do AppHost:

```
<Project Sdk="Microsoft.NET.Sdk">
  <Sdk Name="Aspire.AppHost.Sdk" Version="9.1.0" />
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net9.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Aspire.Hosting.AppHost" Version="9.1.0" />
  </ItemGroup>
</Project>
```

Este arquivo configura o AppHost como um host de orquestração Aspire [7]. Os projetos de API (por exemplo, `Auth.API.csproj`) usam o SDK padrão do ASP.NET Core, mas têm referências ao projeto ServiceDefaults e possivelmente a pacotes do Aspire (por exemplo, `Aspire.Hosting.AppHost`). Cada projeto API contém seu `Program.cs`, controllers, configuração e demais recursos web.

Por exemplo, a solução de exemplo do Aspire exibiu uma estrutura similar: um projeto *ApiService* (depende de ServiceDefaults), um projeto *AppHost* (depende de ApiService e de um *Web*), e um projeto *ServiceDefaults* [8]. Adaptando para nosso caso, cada microsserviço corresponde a um projeto API independente, e o AppHost faz referência a todos eles para iniciar a aplicação distribuída.

Configurações de Ambiente

Cada microsserviço possui um arquivo `appsettings.json` (e variantes por ambiente) com configurações próprias, como *ConnectionStrings*, parâmetros de serviço e RabbitMQ. Por exemplo, um `appsettings.json` típico de **Auth.API** poderia ser:

```
{
  "ConnectionStrings": {
    "AuthDb":
"Host=localhost;Port=5432;Database=AuthDb;Username=postgres;Password=YourPassword"
  },
  "RabbitMQ": {
```

```
    "Host": "localhost",
    "User": "guest",
    "Password": "guest"
  },
  "Redis": {
    "ConnectionString": "localhost:6379"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning"
    }
  }
}
```

Nesse arquivo definimos a string de conexão do banco (`AuthDb`), credenciais de RabbitMQ e Redis, e níveis de log. O .NET Aspire injeta automaticamente as configurações corretas em ambiente local (por exemplo, IP e porta do container Redis) quando usamos métodos como `AddRedis` no AppHost [1]. Usamos nomes em inglês para seções e chaves, mas valores de texto (mensagens, labels) no frontend ficam em português conforme requisito do cliente.

ServiceDefaults e Configurações Compartilhadas

O projeto **ServiceDefaults** pode incluir um arquivo de configuração compartilhado (por exemplo, `appsettings.Defaults.json`) que provê valores padrão para todas as APIs. Esse arquivo deve ser definido como *Copy to Output* e inserido antes dos demais em `Configuration.Sources` (veja [59] para exemplo de implementação). Por exemplo, em `ServiceDefaults/appsettings.Defaults.json` podemos definir configurações comuns:

```
{
  "Logging": {
    "LogLevel": {
      "Microsoft": "Warning",
      "Default": "Information"
    }
  },
  "GlobalSettings": {
    "CacheDurationSeconds": 300
  }
}
```

Esses valores serão carregados em todas as APIs (via `builder.Configuration.AddJsonFileBeforeOtherJson("appsettings.Defaults.json", false, true)`) [9]. Assim, podemos centralizar parâmetros como níveis de log ou tempo de expiração padrão. Cada serviço ainda pode sobrescrever esses valores em seu próprio `appsettings.json`.

Configuração do AppHost

O AppHost (`ChurchManagement.AppHost/Program.cs`) contém a configuração de orquestração. Nele usamos o **builder** de aplicação distribuída do Aspire para declarar recursos. Por exemplo:

```
var builder = DistributedApplication.CreateBuilder(args);

// Provisiona um container Redis e o associa aos serviços que precisarem de
cache
var redis = builder.AddRedis("cache");

// Provisiona Postgres e bancos
var postgres = builder.AddPostgres("postgres").WithPgAdmin();
var authDB   = postgres.AddDatabase("AuthDB");
var churchDB = postgres.AddDatabase("ChurchDB");

// Provisiona RabbitMQ para mensageria
var rabbit  = builder.AddRabbitMQ("messaging");

// Registra projetos e dependências
builder.AddProject<Projects.Auth.API>("auth-api")
       .WithReference(redis)
       .WithReference(authDB)
       .WithReference(rabbit);

builder.AddProject<Projects.Church.API>("church-api")
       .WithReference(redis)
       .WithReference(churchDB)
       .WithReference(rabbit);

// Adicionar os demais serviços de forma similar...
// builder.AddProject<Projects.Member.API>("member-api")...
// ...

builder.Build().Run();
```

Esse código adiciona recursos (Redis, Postgres, RabbitMQ) e em seguida adiciona cada projeto API, referenciando os recursos necessários com `.WithReference()`. Por exemplo, `Auth.API` recebe acesso ao Redis, seu banco *AuthDB* e ao RabbitMQ. O AppHost então inicia todos os serviços e containers necessários, configurando variáveis de ambiente e esperando (WaitFor) até que estejam prontos [10] [4]. Em execução local, cada serviço fica acessível em uma porta localhost, e o painel do Aspire mostra cada "resource" (projeto ou container) com seu estado e endpoint.

Exemplo de Arquivo do AppHost

O `.csproj` do AppHost já foi mostrado acima. Além disso, o AppHost pode incluir um arquivo `aspire.apphost.json` ou similar (dependendo da versão do .NET Aspire) que define configurações do orquestrador. Em geral, a configuração primária está no código C# acima e no `aspire.yml` ou `azure.yaml` se fizermos deploy em nuvem com ferramentas como `azd`. Por exemplo, a CLI Azure Developer (`azd`) gera um `azure.yaml` descrevendo cada serviço do Aspire e seus recursos na

nuvem [11], mas isso foge ao escopo local. Para nosso caso de desenvolvimento local, os recursos do AppHost são configurados via o código mostrado.

Comunicação entre Serviços (REST e Mensageria)

REST via HttpClient

Cada serviço expõe endpoints HTTP REST e pode consumir outros serviços via `HttpClient`. Graças à *service discovery* do Aspire, não usamos URLs fixos. Ao registrar cada projeto no AppHost com um nome (ex.: `"auth-api"`), o Aspire garante que podemos injetar um `HttpClient` (ou usar `IHttpClientFactory`) com o nome do serviço. A extensão `builder.Services.AddServiceDiscovery()` e `http.AddServiceDiscovery()` (chamadas dentro de `AddServiceDefaults`) configuram o HttpClient para usar o DNS de serviço correto [3] [6]. Por exemplo, na `Church.API` poderíamos chamar um endpoint do Auth via `httpClient.GetAsync("https://auth-api/api/users")`; o Aspire substituirá `auth-api` pelo endereço real do container local. Essa descoberta automática simplifica a comunicação: o desenvolvedor não precisa codificar portas ou domínios, apenas o nome lógico do serviço.

Mensageria com RabbitMQ

Para comunicação assíncrona, usamos *RabbitMQ*. No AppHost definimos um recurso RabbitMQ (por exemplo `builder.AddRabbitMQ("messaging")`) [4]. Isso inicia um broker RabbitMQ local. Em cada serviço cliente (Application/Infrastructure), podemos usar a biblioteca de cliente do RabbitMQ (`RabbitMQ.Client`) ou um framework como MassTransit. O Aspire também oferece uma integração de cliente (`Aspire.RabbitMQ.Client`) que registra um `IConnection` automaticamente [12]. Assim, a configuração de host/porta do RabbitMQ chega às aplicações via `IConfiguration`. Os handlers de mensagens nos serviços podem então publicar eventos em exchanges e filas do RabbitMQ. Por exemplo, após cadastrar um membro, `Member.API` pode publicar uma mensagem "MemberCreated" que outros serviços (Eventos, Finanças) consumirão.

Em resumo, adotamos dois padrões de integração: **REST síncrono** para consultas e comandos imediatos, e **fila RabbitMQ** para eventos assíncronos e workflow desacoplados. O Aspire cuida de orquestrar o acesso a esses recursos, simplificando o bootstrap de contêineres de mensageria e injetando as configurações nos serviços [4] [12].

Arquivos de Configuração de Exemplo

`appsettings.json`

Em cada serviço API, o `appsettings.json` define conexões de banco, filas, caches e níveis de log. Exemplo (simplificado) para **Auth.API**:

```
// Auth.API/appsettings.json
{
  "ConnectionStrings": {
    "AuthDb":
"Host=localhost;Port=5432;Database=AuthDb;Username=postgres;Password=Pass"
  },
  "RabbitMQ": {
```

```
    "Host": "localhost",
    "User": "guest",
    "Password": "guest"
  },
  "Redis": {
    "ConnectionString": "localhost:6379"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning"
    }
  }
}
```

Em desenvolvimento local com Aspire, essas configurações podem ser ajustadas automaticamente (ex.: o container Redis fornece a porta correta). Em produção, esses valores apontariam para instâncias reais (e.g. em Kubernetes ou nuvem).

`ServiceDefaults/config.json` (appsettings.Defaults.json)

No projeto **ServiceDefaults** podemos ter um arquivo de configurações padrão compartilhadas. Por convenção, nomeamos algo como `appsettings.Defaults.json`. Por exemplo:

```
// ServiceDefaults/appsettings.Defaults.json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning"
    }
  },
  "GlobalSettings": {
    "CacheDurationSeconds": 300
  }
}
```

Esse arquivo deve ser copiado para a saída e inserido como fonte de configuração antes dos outros *appsettings*. Usamos um trecho como `builder.Configuration.AddJsonFileBeforeOtherJson("appsettings.Defaults.json", optional: false)` para garantir que esses valores sejam carregados primeiro [9]. Assim, todas as APIs herdam esses valores iniciais, podendo sobrescrevê-los localmente. Essa técnica centraliza parâmetros comuns (níveis de log, tempos de cache, etc.).

Configuração do AppHost (Exemplo)

O projeto **AppHost** não usa `appsettings` típicos, mas podemos incluir configuração no código (Program.cs). Um `.csproj` do AppHost típico já foi apresentado acima. Um exemplo de `Program.cs` (perto do código de orquestração já mostrado) serve como *Aspire.apphost configuration*.

Além disso, se usarmos Azure Developer CLI para deploy, seriam gerados arquivos como `azure.yaml` e `.azure/config.json` definindo serviços de nuvem [11]. Porém, para desenvolvimento local, a configuração principal está no código C# do AppHost ilustrado anteriormente.

Testes Unitários

Cada microsserviço deve ter um projeto de testes unitários correspondente, normalmente nomeado como `Auth.API.Tests`, `Church.API.Tests`, etc., seguindo o mesmo padrão de nomes dos serviços. Dentro de cada projeto de testes, organizamos classes de teste agrupadas por funcionalidade: testes de *controllers* (`AuthControllerTests`), de *handlers/comandos* (`CreateUserHandlerTests`), e até de componentes de domínio, se aplicável.

Seguem algumas boas práticas de teste em .NET usando xUnit, Moq e FluentAssertions:

- **Projetos separados**: Os projetos de teste ficam fora dos projetos de produção, referenciando apenas as bibliotecas necessárias (evitando dependências de infraestrutura).
- **Evitar dependências externas**: Em testes unitários não acesse bancos ou serviços reais. Use in-memory ou mocks. Por exemplo, para testar um handler de comando que precisa de repositório, usa-se um *mock* do repositório com Moq [13]. A ideia é *isolar* a lógica de negócio (princípio de dependências explícitas) [14].
- **Nomenclatura clara**: Nomeie testes com formato `{Metodo_Scenario_ResultadoEsperado}` para descrever o propósito [15]. Por exemplo, `CreateUserHandler_InvalidInput_ThrowsValidationError`. Isso torna o teste autoexplicativo.
- **Arrange-Act-Assert (AAA)**: Estruture cada teste em três partes claras: preparar ambiente (mocks, dados), executar ação, e validar resultado [16]. Separe essas fases por comentários ou espaçamento para legibilidade.
- **Uso de Moq**: Crie *mocks* de dependências (serviços, repositórios) usando o `Mock<T>`. Configure comportamentos esperados, por exemplo:

```
var mockRepo = new Mock<IUserRepository>();
mockRepo.Setup(r => r.AddAsync(It.IsAny<User>()))
        .Returns(Task.CompletedTask)
        .Verifiable();
var handler = new CreateUserHandler(mockRepo.Object);
```

- **Assert com FluentAssertions**: Utilize FluentAssertions para asserções mais legíveis: em vez de `Assert.IsType<OkResult>(result)`, faça `result.Should().BeOfType<OkResult>()`. FluentAssertions torna os testes mais expressivos (e.g. `collection.Should().HaveCount(2)` em vez de `Assert.Equal(2, collection.Count)`).
- **Verificar chamadas de mock**: Use `mock.Verify()` ou as opções `Verifiable()` e `mockRepo.Verify()` do Moq para garantir que métodos esperados foram chamados (útil em métodos `POST`, como no exemplo [25] em que verificamos `repo.AddAsync` foi chamado antes do redirecionamento [17]).
- **Evitar lógica nos testes**: Os testes devem ser simples e determinísticos, sem *ifs* ou loops complexos [18]. Use constantes ao invés de "magic strings" diretamente para facilitar manutenção [19].

Exemplos de teste: Para ilustrar, considere um endpoint GET simples de um controller (versão reduzida do exemplo oficial [25]):

```
[Fact]
public async Task Index_ReturnsView_WithTwoItems()
{
    // Arrange
    var testUsers = new List<User> { new User("A"), new User("B") };
    var mockRepo = new Mock<IUserRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
            .ReturnsAsync(testUsers);
    var controller = new HomeController(mockRepo.Object);

    // Act
    var result = await controller.Index();

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model =
Assert.IsAssignableFrom<IEnumerable<UserViewModel>>(viewResult.Model);
    model.Should().HaveCount(2);
}
```

Neste teste, criamos um mock de repositório retornando dois usuários, chamamos a ação `Index()` e então verificamos que o resultado é uma *ViewResult* contendo exatamente 2 itens. Observe o padrão *Arrange-Act-Assert* e a combinação de Moq com asserções. Um teste de POST com validação de modelo poderia usar `controller.ModelState.AddModelError(...)` para simular um modelo inválido e então checar que retorna `BadRequest` [20].

Os testes de *handlers* (casos de uso) seguem lógica análoga: instanciam o handler, usam mocks de dependências, executam o comando/consulta e verificam efeitos ou retornos. O importante é manter alta cobertura de comportamento e não de implementação, testando cada caso de uso e regra de negócio.

Banco de Dados

Cada microsserviço deve ter seu próprio banco de dados. Recomendamos **PostgreSQL** como SGBD principal (é gratuito, robusto e possui bom suporte em .NET via Npgsql) [21]. No ambiente de desenvolvimento local, o .NET Aspire pode provisionar um container PostgreSQL facilmente (como mostrado no AppHost), e cada serviço configura sua *ConnectionString* adequadamente. Para testes de unidade que precisam de acesso a dados, é comum usar **SQLite in-memory** ou substitutos leves (ou simplesmente mocks dos repositórios).

É possível usar outros bancos gratuitos como MySQL ou SQL Server Developer Edition, mas preferimos Postgres pela integração nativa do Aspire (ex.: `builder.AddPostgres("postgres")`) e pela comunidade de desenvolvedores .NET. De qualquer forma, o padrão de arquitetura não depende do SGBD: basta ajustar o *provider* do EF Core e as *connection strings*. O Aspire até provê integrações para outros bancos (ex.: `Aspire.Hosting.SqlServer` para SQL Server) [22], mas em nosso caso Postgres e SQLite atendem aos requisitos de custo.

CI/CD e Deploy

Para automação de integração contínua e deploy, sugerimos usar **GitHub Actions** combinados com Docker e Kubernetes. Cada repositório (ou monorepo com múltiplos serviços) pode ter um workflow GitHub Actions que dispara em cada push/pull request: ele faz `dotnet restore`, `dotnet build`, `dotnet test`, e então constrói imagens Docker para cada serviço e as publica num registro (ex.: Docker Hub, GitHub Container Registry ou Azure Container Registry).

Em seguida, um passo de deploy aplica as imagens a um cluster Kubernetes. Por exemplo, podemos usar um cluster local (minikube, Docker Desktop K8s) ou um gratuito na nuvem (AKS, EKS, GKE têm opções gratuitas de uso limitado) para testes. A Microsoft oferece um tutorial de pipeline que **builda e deploya automaticamente para Azure Kubernetes Service (AKS)** usando Actions [23]. Em essência, o pipeline automatiza:
1. **Compilação**: Restaurar pacotes, compilar e testar cada serviço (.NET 9.0) e a solução como um todo.
2. **Build de Imagens**: Para cada serviço, executar `docker build` usando um Dockerfile (ex. baseado em `mcr.microsoft.com/dotnet/aspnet:9.0` e `sdk:9.0`) e `docker push` para registro.
3. **Deploy no Kubernetes**: Aplicar manifests (Deployment, Service, ConfigMap, Secret, etc.) ou usar `helm` charts. Um possível arquivo `k8s/deployment.yml` declararia pods de cada serviço conectados entre si, com *service discovery* do próprio K8s (via *Services*).
4. **Rollouts e Rollbacks**: O Actions pode permitir reverter a versão da imagem ou do deployment se houver falha, como descrito no módulo de microsserviços e DevOps [24].

Em resumo, a estrutura CI/CD sugere utilizar GitHub Actions para testes automáticos e geração de containers, um registro de imagens gratuito, e Kubernetes como plataforma de produção (ou mesmo testes integrados). O .NET Aspire facilita a orquestração em dev, mas em produção real confiaríamos no K8s ou em Serviços de Contêiner equivalentes (the Aspire orchestration is not meant to replace production-grade systems like Kubernetes [25]). Por fim, a observabilidade (logs, métricas, traces) pode ser integrada ao pipeline para monitorar builds e deployments.

Service Discovery, Observabilidade e Logs

Service Discovery

O *service discovery* do .NET Aspire permite que serviços encontrem uns aos outros em tempo de execução sem precisar de configuração manual de URLs. O AppHost injeta as informações de rede corretas de cada serviço referenciado [1]. Ao chamar `builder.Services.AddServiceDiscovery()` (fato feito por padrão em `AddServiceDefaults` [6]) e usar `http.AddServiceDiscovery()`, qualquer `HttpClient` configurado assim irá resolver automaticamente o nome do serviço (como `"member-api"`) para o endereço de rede local apropriado. Por exemplo, em **Events.API**, podemos usar `httpClient.GetAsync("https://member-api/api/members/123")` e o Aspire substituirá `"member-api"` pelo host:porta do container **Member.API**. [3] Esse mecanismo elimina a necessidade de manter manualmente URLs de serviços; basta usar o nome lógico dado no AppHost.

Observabilidade e Métricas

O .NET Aspire já vem com suporte integrado a **OpenTelemetry**. Chamando `builder.ConfigureOpenTelemetry()` (faz parte de `AddServiceDefaults`) habilitamos logging estruturado, métricas e tracing automáticos [6]. Isso coleta dados como duração de requests HTTP, chamadas de banco de dados, loops da GC, etc. Cada serviço expõe endpoints de *health check* (`/`

`health`, `/alive`) por padrão, graças a `AddDefaultHealthChecks` [6]. O painel Aspire (executado em `F5` no AppHost) apresenta várias seções de monitoramento:

- **Console Logs**: exibe o output de console de cada serviço (stdout) agregado em tempo real [26]. Útil para debug textual básico.
- **Structured Logs**: exibe logs estruturados (ex. via ILogger) filtráveis. Permite consultar eventos de log usando filtros de nível, texto ou labels específicas [27].
- **Traces**: mostra o rastreamento distribuído das requisições. Cada HTTP request aparece em um diagrama de fluxo, evidenciando o caminho por múltiplos serviços e tempo gasto em cada etapa [28] [29]. Isso ajuda a localizar gargalos ou erros em chamadas encadeadas (por exemplo, *Church.API → Member.API → Finance.API*). A figura abaixo exemplifica o rastreamento ("Traces") de uma chamada GET via *webfrontend* que consulta *apiservice*, após cache:

Figura – Página de Tracing *do painel .NET Aspire mostrando uma requisição passando por múltiplos serviços.*

- **Métricas**: apresenta gráficos de métricas embutidas, como contadores de requisições, tempos de processamento, número de conexões abertas, uso de recursos, etc [30]. Por exemplo, abaixo vemos uma métrica de *HTTP client open connections*. Esses dados permitem medir a saúde do sistema e identificar instantes de alta carga (picos nas métricas podem indicar lentidão). A figura ilustra o painel de métricas para conexões HTTP ativas:

Figura – Página de Metrics *do painel .NET Aspire com métricas de conexões HTTP abertas.*

Graças a `ConfigureOpenTelemetry()`, tudo isso funciona sem configuração adicional: o Aspire já habilita coleta de telemetria ASP.NET Core, gRPC e HttpClient [6]. Em ambientes de nuvem ou produção, esses dados poderiam ser exportados a sistemas como Prometheus ou Application Insights configurados por variáveis (`OTEL_EXPORTER_OTLP_ENDPOINT`, etc.) [31] [32].

Logs

Os serviços devem usar `ILogger<T>` para gerar logs no nível adequado (Information, Warning, Error etc.). O Aspire formata e coleta esses logs automaticamente. Em nossos testes locais, os logs aparecem no painel mencionado, mas em produção podemos direcioná-los a arquivos ou ferramentas de log centralizado (ELK, Seq, etc.). O importante é que o ambiente de testes Aspire permite ver na hora cada log gerado pelo serviço. As melhores práticas de logs incluem mensagens claras e estruturas (JSON) para fácil pesquisa.

Em síntese, o .NET Aspire fornece *observabilidade* incorporada: ao usar `AddServiceDefaults()`, já ganhamos logging estruturado e tracing distribuído integrado [6] [29]. O painel Aspire possibilita inspecionar logs (console e estruturados) e traçar requisições entre microsserviços de forma visual, facilitando debugging e monitoramento.

Cache Distribuído com Redis

Para otimizar performance, adotamos cache distribuído via **Redis**. O Redis pode ser usado para *caching* de consultas frequentes ou armazenar sessões/estado que precisa ser compartilhado. O .NET Aspire possui integração completa com Redis. No AppHost adicionamos um recurso Redis com `builder.AddRedis("cache")` [33] [34], que inicia um container Redis. Em seguida referenciamos esse recurso nos microsserviços que usarão o cache, por exemplo:

```
var cache = builder.AddRedis("cache");
builder.AddProject<Projects.Auth.API>("auth-api")
       .WithReference(cache);
builder.AddProject<Projects.Church.API>("church-api")
       .WithReference(cache);
// ...
```

O método `.WithReference(cache)` faz com que o contêiner Redis fique acessível aos serviços. Internamente, ele adiciona no `Configuration` de cada serviço uma connection string para o Redis (nomeada "cache"). Nos serviços, usamos a biblioteca **StackExchange.Redis** ou `IDistributedCache` (via `Microsoft.Extensions.Caching.StackExchangeRedis`) apontando para essa connection string. O Aspire também oferece um pacote `Aspire.StackExchange.Redis.DistributedCaching` que registra `IDistributedCache` automaticamente usando o Redis provisionado [35]. Em código, seria algo como:

```
services.AddStackExchangeRedisCache(options =>
{
    options.Configuration =
builder.Configuration.GetConnectionString("cache");
});
```

Ou, usando a integração do Aspire: adicionamos o pacote `Aspire.StackExchange.Redis.DistributedCaching`, e então chamamos `builder.Services.AddDistributedCache()` (o próprio pacote faz o trabalho). De qualquer forma, o ponto-chave é que o container Redis está disponível e cada serviço pode armazenar dados compartilhados nele (p.ex.: cache de lista de membros, tokens de sessão, etc.).

Em desenvolvimento, podemos também incluir ferramentas auxiliares com `.WithRedisInsight()` ou `.WithRedisCommander()` no AppHost para interfaces gráficas do Redis [36] [37]. Em produção, aponta-se a um servidor Redis gerenciado ou cluster (p. ex. Azure Cache for Redis).

Conclusão

Esta documentação apresentou uma **arquitetura de microsserviços** completa usando .NET Aspire como base. Detalhamos a estrutura de projetos (AppHost, ServiceDefaults, API/Application/Domain/Infrastructure), configuração dos recursos (PostgreSQL, Redis, RabbitMQ), integração entre serviços (REST com descobrimento, mensageria assíncrona), arquivos de configuração de exemplo (`appsettings.json`, defaults compartilhados), organização de testes unitários e boas práticas, e sugestões de CI/CD com Docker e Kubernetes. Todos os serviços foram nomeados em inglês (Auth.API, Church.API etc.), com textos de interface em português como exigido. A instrumentação de observabilidade (logs, tracing, métricas) foi integrada por padrão, facilitando o monitoramento. Em suma, este guia serve como base técnica para desenvolvimento e operação do sistema, integrando ferramentas modernas do ecossistema .NET e práticas recomendadas de arquitetura de microsserviços.

Fontes: A descrição técnica apoia-se na documentação oficial do .NET Aspire [2] [1] [6], em exemplos de integração com Redis e RabbitMQ [34] [4], em tutoriais Microsoft e posts especializados [26] [29] [13], além de exemplos de código Aspire [10] [5]. Todas as práticas sugeridas seguem orientações de design e testes em .NET [14] [15].

1 2 3 21 25 33 .NET Aspire overview - .NET Aspire | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/aspire/get-started/aspire-overview

4 12 .NET Aspire RabbitMQ integration - .NET Aspire | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/aspire/messaging/rabbitmq-integration

5 GitHub - a-sharifov/Dotnet.Aspire: This project showcases Aspire library with clean architecture and minimal API.
https://github.com/a-sharifov/Dotnet.Aspire

6 31 32 .NET Aspire service defaults - .NET Aspire | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/aspire/fundamentals/service-defaults

7 .NET Aspire SDK - .NET Aspire | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/aspire/fundamentals/dotnet-aspire-sdk

8 Build your first .NET Aspire solution - .NET Aspire | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/aspire/get-started/build-your-first-aspire-app

9 c# - How can I provide a default configuration for all Aspire Services? - Stack Overflow
https://stackoverflow.com/questions/79251796/how-can-i-provide-a-default-configuration-for-all-aspire-services

10 .NET Aspire orchestration overview - .NET Aspire | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/aspire/fundamentals/app-host-overview

11 22 Deploy a .NET Aspire project with a SQL Server Database to Azure - .NET Aspire | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/aspire/database/sql-server-integration-deployment

13 17 20 Test controller logic in ASP.NET Core | Microsoft Learn
https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/testing?view=aspnetcore-9.0

14 15 16 18 19 Best practices for writing unit tests - .NET | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices

23 24 Deploy a cloud-native .NET microservice automatically with GitHub Actions and Azure Pipelines - Training | Microsoft Learn
https://learn.microsoft.com/en-us/training/modules/microservices-devops-aspnet-core/

26 27 28 29 30 Building Microservices Using .NET Aspire | by Bruce De Jager | Medium
https://medium.com/@brucedej/building-microservices-using-net-aspire-57c5a171bef2

34 36 37 .NET Aspire Redis integration - .NET Aspire | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/aspire/caching/stackexchange-redis-integration

35 .NET Aspire Redis distributed caching integration - .NET Aspire | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/aspire/caching/stackexchange-redis-distributed-caching-integration