ChatGPT

Estrutura do projeto Gateway

A solução **ChurchManagement** terá um projeto separado de *API Gateway*, organizado de forma modular. Por exemplo:

```
ChurchManagement.sln
└── Gateway/
    ├── Gateway.API/         # Projeto principal (aspnet core ou similar)
    ├── Gateway.Application/ # (Opcional) Casos de uso e lógica de negócio do
Gateway
    └── Gateway.Infrastructure/ # Configurações de infraestrutura (ex: YARP)
```

O **Gateway.API** hospeda o proxy reverso (por exemplo, um projeto ASP.NET Core com YARP) e expõe os endpoints públicos. Ele deverá referenciar outros projetos (por exemplo, Auth.API, Church.API, Event.API) ou pelo menos conhecê-los para rotear chamadas. A pasta **Gateway.Infrastructure** conterá configurações específicas, como arquivos JSON de roteamento ou transformações do YARP. A pasta **Gateway.Application** pode conter serviços de aplicação específicos do gateway, como classes de autorização customizadas, se necessário.

Abordagem 1: ASP.NET Core + YARP

Nesta abordagem, o Gateway é um projeto ASP.NET Core que usa o **YARP (Yet Another Reverse Proxy)** para rotear requisições aos microserviços. Os principais pontos incluem:

- **Rotas e Clusters:** Configure em *appsettings.json* (ou em código) as rotas (`Routes`) e clusters (`Clusters`) do YARP. Por exemplo, no `appsettings.json` sob `"ReverseProxy"` define-se algo como:

```
{
  "ReverseProxy": {
    "Routes": {
      "auth-route": {
        "ClusterId": "auth-cluster",
        "Match": { "Path": "/auth/{**catch-all}" },
        "AuthorizationPolicy": "anonymous"
      },
      "church-route": {
        "ClusterId": "church-cluster",
        "Match": { "Path": "/church/{**catch-all}" },
        "AuthorizationPolicy": "default"
      },
      "event-route": {
        "ClusterId": "event-cluster",
        "Match": { "Path": "/event/{**catch-all}" },
        "AuthorizationPolicy": "default"
```

```
      }
    },
    "Clusters": {
      "auth-cluster": {
        "Destinations": {
          "authDestination": { "Address": "http://localhost:5001/" }
        }
      },
      "church-cluster": {
        "Destinations": {
          "churchDestination": { "Address": "http://localhost:5002/" }
        }
      },
      "event-cluster": {
        "Destinations": {
          "eventDestination": { "Address": "http://localhost:5003/" }
        }
      }
    }
  }
}
```

Cada rota aponta para um *ClusterId* que contém o endereço do serviço de destino. Neste exemplo, `/auth/**` não exige autenticação (`AuthorizationPolicy: "anonymous"`), enquanto `/church/**` e `/event/**` exigem usuário autenticado (`"default"`) [1]. O YARP permite definir `"AuthorizationPolicy"` por rota no JSON [1], que corresponde a políticas definidas em código.

- **Autenticação JWT:** O Gateway valida o *token JWT* emitido pela Auth.API. No `Program.cs`, adiciona-se a autenticação do tipo Bearer. Por exemplo:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddReverseProxy()
       .LoadFromConfig(builder.Configuration.GetSection("ReverseProxy")); //
carrega rotas/clusters [2]

// Configura autenticação JWT
builder.Services.AddAuthentication("Bearer")
       .AddJwtBearer("Bearer", options =>
       {
           options.Authority = "https://auth.exemplo.com"; // URL do Auth.API
           options.TokenValidationParameters = new TokenValidationParameters
           {
               ValidateAudience = false
           };
       });
builder.Services.AddAuthorization();

var app = builder.Build();
app.UseAuthentication();
app.UseAuthorization();
```

```
app.MapReverseProxy().RequireAuthorization(); // exige token válido [3] [4]
app.Run();
```

O trecho acima configura o YARP a partir da seção `"ReverseProxy"` [2] e habilita JWT Bearer. As chamadas só são roteadas se `RequireAuthorization()` passar (token válido) [4]. *Importante:* `app.MapReverseProxy().RequireAuthorization()` exige que todas as rotas do proxy só funcionem com usuário autenticado. Cada rota individual também pode declarar sua política (`"default"` ou `"anonymous"`) no JSON [1].

- **Políticas por rota:** Além de `"default"` (usuário autenticado) e `"anonymous"`, você pode criar políticas customizadas (por exemplo, exigindo claims/roles específicas) e referenciá-las nas rotas. Exemplo de política customizada no `Program.cs`:

```
services.AddAuthorization(options =>
{
    options.AddPolicy("IsAdmin", policy =>
        policy.RequireAuthenticatedUser().RequireClaim("role", "Admin"));
});
```

E no JSON do YARP:

```
"some-route": {
  "ClusterId": "some-cluster",
  "AuthorizationPolicy": "IsAdmin",
  "Match": { "Path": "/admin/{**catch-all}" }
}
```

Isso faz o YARP aplicar a política “IsAdmin” naquela rota [5].

- **Fluxo de credenciais:** O YARP, por padrão, encaminha cabeçalhos de autorização (como cookies ou Bearer) aos serviços finais. Assim, depois de validar o JWT no Gateway, você pode encaminhar o mesmo token para os microsserviços (para que eles saibam quem é o usuário). Nada adicional é preciso além de deixar os headers passarem (o YARP já transfere tokens por padrão) [6].

Exemplo de configuração completa

Em `Gateway.API/Program.cs`, poderíamos ter:

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

// 1. Adiciona o reverse proxy YARP com config de rotas/clusters
builder.Services.AddReverseProxy()
       .LoadFromConfig(builder.Configuration.GetSection("ReverseProxy")); //
YARP lê "ReverseProxy" do appsettings [2]

// 2. Configura autenticação JWT (Auth.API como autoridade)
builder.Services.AddAuthentication("Bearer")
       .AddJwtBearer("Bearer", options =>
```

```
        {
            options.Authority = "https://auth.exemplo.com";
            options.TokenValidationParameters.ValidateAudience = false;
        });
builder.Services.AddAuthorization();

var app = builder.Build();

// 3. Usa middlewares de autenticação/autorização
app.UseAuthentication();
app.UseAuthorization();

// 4. Mapeia o proxy YARP (filtrando por autorização)
app.MapReverseProxy().RequireAuthorization();

app.Run();
```

Em `Gateway.API/appsettings.json`, definimos as rotas e destinos (exemplo adaptado):

```
{
  "ReverseProxy": {
    "Routes": {
      "auth-route": {
        "ClusterId": "auth-cluster",
        "AuthorizationPolicy": "anonymous",
        "Match": { "Path": "/auth/{**catch-all}" }
      },
      "church-route": {
        "ClusterId": "church-cluster",
        "Match": { "Path": "/church/{**catch-all}" },
        "AuthorizationPolicy": "default"
      },
      "event-route": {
        "ClusterId": "event-cluster",
        "Match": { "Path": "/event/{**catch-all}" },
        "AuthorizationPolicy": "default"
      }
    },
    "Clusters": {
      "auth-cluster": {
        "Destinations": {
          "authDest": { "Address": "http://localhost:5001/" }
        }
      },
      "church-cluster": {
        "Destinations": {
          "churchDest": { "Address": "http://localhost:5002/" }
        }
      },
      "event-cluster": {
```

```
        "Destinations": {
          "eventDest": { "Address": "http://localhost:5003/" }
        }
      }
    }
  }
}
```

Nesse exemplo, o endpoint `/auth/**` do Gateway encaminha para **Auth.API** sem exigir token; `/church/**` e `/event/**` vão para seus serviços com o token JWT válido [1] [7].

Abordagem 2: NGINX + validação JWT com Lua

Como alternativa mais leve, podemos usar **NGINX** (ou OpenResty) como proxy reverso, validando o JWT via Lua (sem usar .NET). Nesse caso:

- **Instalação:** Use o NGINX tradicional ou *OpenResty* (NGINX com suporte a Lua). Em um VPS Ubuntu 24.04, pode-se instalar o pacote `openresty` ou compilar o NGINX com o módulo Lua JIT. Também instale o módulo lua-resty-jwt (via luarocks ou clonando o código).

- **Upstream de serviços:** Defina *upstreams* ou *server blocks* para os serviços Auth, Church, Event. Por exemplo:

```
http {
    lua_shared_dict jwt_cache 10m;  # Cache local de chaves ou tokens

    upstream auth_backend {
        server localhost:5001;
    }
    upstream church_backend {
        server localhost:5002;
    }
    upstream event_backend {
        server localhost:5003;
    }

    server {
        listen 80;
        server_name gateway.exemplo.com;

        location /auth/ {
            proxy_pass http://auth_backend;
        }

        # Caminhos Church, Event passam por verificação de JWT
        location ~ ^/(church|event)/ {
            access_by_lua_block {
                local jwt = require "resty.jwt"
                local auth_header = ngx.var.http_Authorization
```

```
                if not auth_header then
                    ngx.status = ngx.HTTP_UNAUTHORIZED
                    ngx.say("Token JWT ausente")
                    return ngx.exit(ngx.HTTP_UNAUTHORIZED)
                end

                local _, _, token = string.find(auth_header, "Bearer%s+(.+)")
                if not token then
                    ngx.status = ngx.HTTP_UNAUTHORIZED
                    ngx.say("Cabeçalho Authorization inválido")
                    return ngx.exit(ngx.HTTP_UNAUTHORIZED)
                end

                -- Valida o token (usar secret ou chave pública conforme
configuração)
                local jwt_obj = jwt:verify("minha_chave_secreta", token)
                if not jwt_obj.verified then
                    ngx.status = ngx.HTTP_UNAUTHORIZED
                    ngx.say("Token JWT inválido")
                    return ngx.exit(ngx.HTTP_UNAUTHORIZED)
                end

                -- Opcional: repassar dados do usuário para o backend
                ngx.req.set_header("x-user-id", jwt_obj.payload.sub)
            }
            proxy_pass http://$1_backend;
        }
    }
}
```

Nesse exemplo, todo tráfego para `/church/**` ou `/event/**` passa pelo bloco `access_by_lua_block`, que usa `lua-resty-jwt` para verificar o cabeçalho `Authorization`. Se o token faltar ou for inválido, retorna `401 Unauthorized`. Caso contrário, a requisição é repassada (`proxy_pass`) para o upstream apropriado [8] [9]. O trecho `[ngx.req.set_header("x-user-id", jwt_obj.payload.sub)]` mostra como extrair dados do token (ex: `sub` do JWT) e repassá-los como header.

- **Validação do JWT:** A biblioteca Lua (`resty.jwt`) faz a decodificação e verificação do token usando uma chave secreta ou um par de chaves público/privado. No exemplo acima, usamos um *secret* simples (`"minha_chave_secreta"`). Em cenários reais, use uma chave segura ou obtenha as chaves públicas do serviço de autenticação. Esta validação garante que somente tokens legítimos (emitidos pela Auth.API) sejam aceitos.

- **Bloquear requisições não autorizadas:** Se o script Lua detectar problemas (token ausente, formato inválido, assinatura inválida, token expirado etc.), deve sempre retornar `ngx.exit(ngx.HTTP_UNAUTHORIZED)` e mensagem de erro. Assim, o NGINX bloqueia requisições indevidas antes de chegar aos serviços internos.

Exemplo de configuração NGINX

Conforme exemplos como [9], o `nginx.conf` pode conter:

```
http {
    lua_shared_dict jwt_cache 10m;  # Cache de validação de JWT

    server {
        listen 80;
        server_name gateway.example.com;

        location /auth/ {
            # Roteia sem checar token (rota pública para login/registro)
            proxy_pass http://localhost:5001;
        }

        # Roteia church e event através de validação JWT
        location ~ ^/(church|event)/ {
            access_by_lua_block {
                local jwt = require "resty.jwt"
                local auth_header = ngx.var.http_Authorization

                if not auth_header then
                    ngx.status = ngx.HTTP_UNAUTHORIZED
                    ngx.say("Token ausente")
                    return ngx.exit(ngx.HTTP_UNAUTHORIZED)
                end

                local _, _, token = auth_header:find("Bearer%s+(.+)")
                if not token then
                    ngx.status = ngx.HTTP_UNAUTHORIZED
                    ngx.say("Token inválido")
                    return ngx.exit(ngx.HTTP_UNAUTHORIZED)
                end

                local jwt_obj = jwt:verify("minha_chave_secreta", token)
                if not jwt_obj.verified then
                    ngx.status = ngx.HTTP_UNAUTHORIZED
                    ngx.say("Token inválido")
                    return ngx.exit(ngx.HTTP_UNAUTHORIZED)
                end

                ngx.req.set_header("x-user-id", jwt_obj.payload.sub)
            }
            # Após validação, redireciona para o backend correto
            proxy_pass http://$1_backend$request_uri;
        }
    }
}
```

Neste bloco:

- `lua_shared_dict jwt_cache` aloca memória compartilhada para eventuais caches de chave ou tokens.
- O `access_by_lua_block` executa a validação do JWT antes de encaminhar a requisição.
- `[9†L72-L81]` exemplifica a captura do header `Authorization` e verificação do token com `resty.jwt`.
- Após validação, faz `proxy_pass` para o upstream correspondente [8] [9].

Essa configuração não requer containers extras (roda no NGINX normal do sistema) e é compatível com Linux. Em produção real, cuide de encriptar/armazenar com segurança a chave JWT e considere cache de chaves públicas se usar certificados.

Comunicação Frontend

Independentemente da abordagem, o *frontend (Vue/NPM)* comunica-se **apenas com o Gateway**. Todas as chamadas a APIs (autenticação, dados da igreja, eventos etc.) usam a URL do Gateway. Por exemplo, o frontend chamaria `/church/listar` no Gateway, que rotearia internamente para `Church.API`. O Gateway repassa a resposta do serviço correto de volta ao cliente. Isso simplifica CORS e regras de segurança, pois o navegador só enxerga o domínio do Gateway.

Integração com .NET Aspire (AppHost)

Para orquestração local com .NET Aspire, o **AppHost** (projeto orquestrador) deve incluir o Gateway e os microserviços como recursos. No `Program.cs` do AppHost você faria algo como:

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

// Registra cada serviço como projeto na orquestração
var authApi    = builder.AddProject<Projects.Auth.API>("Auth.Api");
var churchApi  = builder.AddProject<Projects.Church.API>("Church.Api");
var eventApi   = builder.AddProject<Projects.Event.API>("Event.Api");
var gateway    = builder.AddProject<Projects.Gateway.API>("Gateway")
                        .WithReference(authApi)
                        .WithReference(churchApi)
                        .WithReference(eventApi);

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

Isso instrui o AppHost a executar os projetos Auth.API, Church.API, Event.API e Gateway.API. Note que usamos `WithReference(...)` para indicar dependências (por exemplo, o Gateway depende que os serviços estejam prontos) [10]. O código acima segue o modelo do .NET Aspire, conforme exemplo oficial [10]. Durante o desenvolvimento, o Aspire iniciará cada serviço e o gateway na mesma topologia, facilitando testes end-to-end.

Hospedagem em VPS Linux

Toda a solução pode rodar em um **VPS Ubuntu Server 24.04** com .NET instalado:

- O projeto **Gateway.API** (YARP) é um app ASP.NET Core. Instale o **.NET Runtime** (versão compatível, ex: .NET 8 ou 9). Basta publicar o projeto (`dotnet publish`) e executar como um serviço (via `systemd` ou `nginx reverse proxy + Kestrel`).
- Se usar **YARP**, ele roda sobre Kestrel no Linux sem problemas. É recomendável configurar o Kestrel para ouvir em portas internas e, se quiser, colocar nginx/httpd na frente para SSL.
- Se usar **NGINX**, instale o pacote `nginx` (ou OpenResty). O NGINX atua diretamente como reverse proxy e validará JWT conforme descrito.
- **Consumo de recursos:** Ambas soluções são relativamente leves. Um VPS com 2 vCPU e 12GB de RAM é mais que suficiente para receber dezenas ou centenas de requisições por segundo, dependendo de carga. Como não há containers extras (rodamos o .NET diretamente e o NGINX nativo), economizamos overhead.
- **Custo:** Rodar apps .NET e NGINX no Linux não envolve licenças extras. Use ambientes de execução Open Source. Para armazenamento de tokens/jwks, prefira soluções simples (ex: armazenar chave em arquivo ou variável, não em banco).

Em resumo, o Gateway do tipo YARP ou NGINX+Lua atende aos requisitos: encaminha chamadas dinamicamente aos microserviços, autentica via JWT da Auth.API, recusa acessos indevidos e expõe apenas os endpoints do Gateway ao front-end.

Fontes: Documentação oficial do YARP sobre roteamento e políticas [11] [2]; tutoriais sobre uso de JWT no YARP [3] [4]; e exemplos de NGINX com validação de JWT em Lua [8] [9].

---

[1] [5] Implementing API Gateway Authentication With YARP
https://www.milanjovanovic.tech/blog/implementing-api-gateway-authentication-with-yarp

[2] [7] YARP Configuration Files | Microsoft Learn
https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/yarp/config-files?view=aspnetcore-9.0

[3] [4] Securing APIs with YARP: Authentication and Authorization in .NET 8 Minimal APIs - DEV Community
https://dev.to/leandroveiga/securing-apis-with-yarp-authentication-and-authorization-in-net-8-minimal-apis-2960

[6] [11] YARP Authentication and Authorization | Microsoft Learn
https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/yarp/authn-authz?view=aspnetcore-9.0

[8] [9] Implementing JWT Authentication in NGINX Without NGINX Plus
https://soufianebouchaara.com/implementing-jwt-authentication-in-nginx-without-nginx-plus/

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