**Protocolo HTTP**

Ele é um marco da criação da WEB, que gira em torno da conversa do “**Client**” com o “**Server**”. Você faz um pedido (request) e ele te dá uma response(resposta).

**HTTP REQUEST**

**Verb** – São os verbos que utilizamos nas requisições (POST, GET).

**URI** – Endereço da requisição.

**Version** – Versão do protocolo.

**Request HEADER** – Cabeçalho da Requisição (versão do browser, versão do Windows etc.).

**Request Message** – É a requisição de fato.

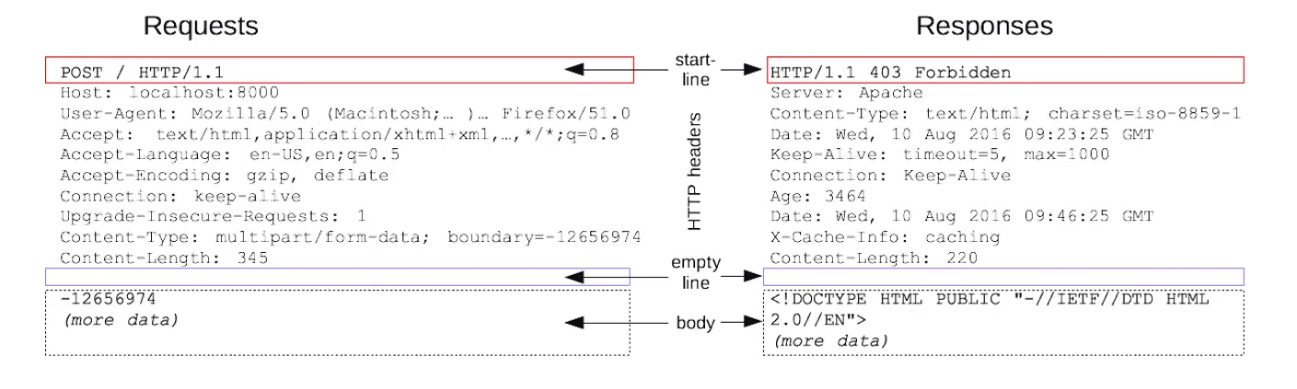
**HTTP RESPONSE**

**Response Code** - Código da resposta.

**HTTP Version** – Versão do Http.

**Response Header** – um código, um token etc.

**Response Message** – Requisição vinda do server.



**TCP – Transmission Control Protocol**

**TCP (Transmission Control Protocol)** is an important network protocol that lets two hosts connect and exchange data streams. TCP guarantees the delivery of data and packets in the same order as they were sent. Vint Cerf and Bob Kahn, who were DARPA scientists at the time, designed TCP in the 1970s.

**TLS** - **Transport Layer Security**

**Transport Layer Security (TLS)**, formerly known as **Secure Sockets Layer (SSL)**, is a protocol used by applications to communicate securely across a network, preventing tampering with and eavesdropping on email, web browsing, messaging, and other protocols. Both SSL and TLS are client / server protocols that ensure communication privacy by using cryptographic protocols to provide security over a network. When a server and client communicate using TLS, it ensures that no third party can eavesdrop or tamper with any message.

**REST vs SOAP**

**REST** – Envio de dados com base em REST (Representational State Transfer). Esse estado de representação é apenas texto (recebida e enviada). Ela é transmitida por meio do protocolo HTTP como se fossem um envelope (por possuir um head, body etc.).

**SOAP** – Simple Object Access Protocol. Além dos dados ele é um padrão baseado em XML sendo bastante verboso, tornando bem mais pesada.

**Arquitetura REST**

Uma arquitetura REST é criar um nível de abstração dentre as suas APIs. Quando falamos em arquitetura REST, não estamos dizendo como tudo funciona, mas sim que a lógica de que escrevemos está distribuída em APIs (serviços).

**Arquitetura REST em Microservices**

É uma arquitetura REST em que, cada API, possui um microservice. Um serviço que efetua apenas uma coisa. Quebrando um sistema monolito em pedaços.

**Arquitetura REST Monolítica**.

Um sistema único, não dividido, que roda em um único processo, uma aplicação de software em que diferentes componentes estão ligados a um único programa dentro de uma única plataforma.

**CRIANDO UMA API REST**

1º Criando através do dotnet CLI (command line)

1. Abro o command line or PowerShell na pasta desejada.
2. Digito o comando dotnet new -h (para obter ajuda dos demais comandos)
3. **Para criar uma WebAPI**, utilizo o comando **“dotnet new webapi”.**
4. Comando **“cd + nome da pasta”**: entra na pasta desejada.
5. Comando **“dir”** mostra todos os arquivos contidos naquele local.

**CURIOSIDADES:**

* Ao trabalhar com as APIs, elas também têm suas convenções, assim como o MVC. Com as APIs, a **Controller** recebe uma annotation **[ApiController**].
* Ela herda de ControllerBase, uma Controller mais simplificada utilizada para o uso de API’s.
* Os métodos recebem os nomes dos verbos, já indicando a sua finalidade. Exemplo (public void Post, Get,Put, Delete).

**CONTROLLER**

A controller no desenvolvimento de APIs elas são de extrema importância, pois são com elas que temos uma base de nossa API.

A nossa **controller** herda de **ControllerBase**, parecida com uma controller padrão, porém, um pouco mais simples. Com isso, ela é complementada com a annotation **[ApiController].**

**ROTAS**

É comum em nossas annotation verbs, elas passarem uma rota, isso é muito comum em projetos. Exemplo: [HttpGet({id:int})]. É recomendado já especificarmos o tipo de dado que será aceito pela URL.

**ActionsResult**

* Quando estamos trabalhando com nossos retornos de nossas views, nós podemos utilizar o ActionResult tipado (ActionResult<IEnumarable<>) ou não (ActionResult). Porém, há uma diferença que pode influenciar, a ActionResult permite retornar alguns resultados como **BadRequest, NotFound, Ok** todos eles necessitam de um Result, ao contrário de um método IEnumerable.
* Outro motivo é que quando fazemos uma **ActionResult** e não **especificamos o tipo de retorno** dele, ele apenas receberá um **resultado**, não nos possibilitando em umas ocasiões por exemplo, de retornamos uma coleção.

**Formatadores de respostas**

* **[FromBody]** nos diz que o que estamos recebendo está vindo no corpo do nosso request.
* **[FromRoute]** podemos deixar explicito que o Id por exemplo, está vindo da rota.
* **[FromForm]** está vindo de um formulário por exemplo.
* Temos um formatador de resposta que utilizamos como annotation é o **[ProducesResponseType(typeof(nomeDaEntidade), StatusCodes.Status201Created)]**. A diferença para 201 para o 200, é que o 201 ele retorna um Ok que foi criado, já o 200 apenas uma confirmação para qualquer coisa.
* **[ProducesResponsesType(StatusCodes.Status400BadRequest)]** para qualquer erro obtido.

**Formatadores de dados de resposta Personalizados**

* Podemos criar uma classe “mãe” que herda de nossa ControllerBase, tornando assim a nossa principal.

**Public abstract class MainController : ControllerBase**

**{**

**Protected ActionResul CustomResponse**

**}**

* Assim, podemos criar nossa ActionResult personalizada.

**[CONFERIR VIDEO FORMATADOR DE RESPONSE PERSONALIZADO PARA ENTENDER MAIS A FUNDO]**

**Analisadores e Convenções**

Primeiro passo, é instalar os pacotes **Analyzers**. Como é um pacote a parte, temos que adicioná-lo pela **Nugget Package manager**.

Eles servem para te lembrar das coisas que você implementou e esqueceu de anotar no método. Elas serviram mais para frente na implementação da documentação.

* Install-Package Microsoft.AspNetCore.Mvc.Api.Analyzers

Já as convenções, elas servem **para nos mostrar os retornos comuns** que aquela **Action** pode ter.

Por exemplo, em nosso Post, nos sabemos que ele pode retornar 2 tipos de resultados, tanto um **201 (OK)** quanto um **400(Bad Request).**

Então, utilizando as convenções ficariam assim:

**[ApiConventionMethod(typeof(DefaultApiConventions), nameof(DefaultApiConventions.(TIPO DE REQUEST(VERB)].**

Para que eu não precise colocar a todo momento essa minha convenção. Acima de minha classe, do Controller, coloco minha Convention.

[**ApiConventionType(typeof(DefaultApiConventions)].**

Caso ainda não queira colocar em minha Controller, posso colocar em Startup. Em cima do namespace, emito um assembly.

**[assembly: ApiConventionType(typeof(DefaultApiConventions))]**

**Minha Primeira API**

* Inicializando criando minha API na pasta;
* Configurando Minhas Models;
* Configurando DbContext, com o construtor e DbSets;
* Adicionando na minha statup/services meu serviço.DbContext(options => options.MySQL(Configuration.GetConnetionString()).
* Adiciono em meu appsettings meu connection com o banco de dados.
* Faço minha migration.

**CRIANDO API DO ZERO**

* Visual Studio -> Blank Solution.
* Na minha pasta raiz, crio uma **pasta chamada src**.
* Adiciono meus projetos, se eles já existirem.
* Adiciono as minhas dependências a esses projetos.

**COMO FUNCIONA O FLUXO DE ARQUITETURA – LEITURA**

Diagrama

Descrição gerada automaticamente

* Temos um banco de dados, um cliente mobile ou browser, que está consumindo a nossa aplicação.
* O cliente solicita através de um request um pedido, este mesmo que será recebida pela API e automaticamente vai acessar com o nosso repositório (base de dados).
* Após essa leitura, ela irá retornar uma entitidade desse negócio e precisaremos devolver essa entity para o cliente. Porém, não é uma boa prática devolvermos essa entity diretamente, mas sim retornando-a por meio de uma DTO/ViewModel. Com isso, utilizamos o AutoMapper, que transforma essas entidades em viewModels.
* Por fim, a resposta para o cliente é um ok – 200, e a message de fato.

**COMO FUNCIONA O FLUXO DE ARQUITETURA – GRAVAÇÃO**

* Novamente tenho meu cliente, seja por meio de um banco de dados, um mobile ou browser ele faz uma requisição.
* Porém, essa requisição ela é do verbo POST, PUT e DELETE, daí que surge alguns possíveis problemas. Por quê? Pois são mudanças que fazemos no estado da entidade (editando, colocando ou deletando).
* A minha API vai receber esse input, vai receber essa minha entidade (ex: Produto) vai transformá-la em ViewModel e teremos que ter o trabalho de converter isso em uma entidade.
* Nesse processo, vamos invocar processos assíncronos da camada de negócios, que fará uma serie de validações para testar se a entidade está consistente, preenchido corretamente.
* Se ele encontra problemas, ele não lançará uma exceptions, mas sim EVENTOS. Esses eventos são os erros encontrados.
* API com sua camada de inteligência, vai até esses eventos e irá procurar eventos de erros. Esses que ela retornará para o cliente um json com o Status Code = 400 (No caso de erros).
* Em caso de sucesso no banco de dados, será emitido um OK. Ela(API) vai pegar essa entidade persistida na base, vai serializado e será enviada para o cliente por meio do StatusCode = 200 OK.

**IMPLEMENTANDO VIEWS MODELS**

* Em nossa API, não podemos expor as nossas entidades. A APi deve ter os seus próprios exchanges, entrada e saída dos dados, preservando assim a consistência da sua entidade.
* Dentro do nosso Project, criamos uma pasta ViewModels.(Poderiam se chamar DTO’s).
* Cuidado ao relacionar as entidades por meio das **ViewModels** e criar pequenos “nós”. Isso daria um problema na hora da formação do json.

**IMPLEMENTANDO AS CONTROLLERS**

* Quando utilizamos o Scaffold para construirmos a nossa API, ela já vem com uma Controller exemplo. Nos iniciaremos com ela, fazendo algumas modificações:
  + O conteúdo dentro dela apagaremos.
  + O nome da classe é **MainController**, por possuir algumas ações principais como (validações de notificações de erro, validação de modelstate, validação da operação de negócios).
  + Tiro a minha rota padrão acima de nossa [ApiController]
  + Transformo a classe em public abstract. Podendo apenas ser herdada.
* As nossas próximas classes herdaremos de nossa MainController, passando a sua rota também. Ex:
  + [Route("api/[fornecedores]")]

public class FornecedoresController : MainController

* Criando o primeiro método, criamos obtendo uma lista dessa minha entidade. Como não é um método que exija valor de retorno, eu posso escolher a forma de IActionResult. Além disso, todos devem ser assíncronos.
* Ao realizar esse método, ele retorna uma entidade, essa que na hora de converter para FornecedorViewModel, daria um erro. Para corrigir isso, devemos utilizar o AutoMapper.
  + Para isso, no manager Package Console e faço:
  + **Install-package AutoMapper.Extensions.Microsoft.DependencyInjection**
  + Para o AutoMapper funcionar, em Startup -> ConfigureServices -> services.AddAutoMapper(typeof(Startup));
  + Para configurá-lo, vou adicioná-lo dentro de uma pasta chamada Configuration.
  + Nessa pasta, crio um arquivo chamado **AutoMapperConfig** : **Profile** (Profile é um tipo da AutoMapper que resolve todos os maps que estiverem dentro dele, ou seja, a transformação da Entity para View Model).
  + E crio meus maps (**CreateMap<Entity, EntityViewModel>()** caso aconteça também o mapeamento ao contrário, adiciono meu ReverseMap).
* De volta a minha classe Controller, eu vou adicionar uma dependency injection IMapper \_mapper.
* Por fim, em nossa query, adiciono o \_mapper.Map passando como argumento o meu tipo de map e a fonte do dado.
  + var fornecedor = \_mapper.Map<IEnumerable<FornecedorViewModel>>(await \_fornecedorRepository.ObterTodos());
* Quando receber uma ViewModel de um Post, em meu \_mapper passo as seguintes instruções:
  + \_mapper.Map<TipoEntidade>(TipoViewModel);

**COMO SERIALIZAR UM DTO PARA JSON**

* Primeiro passo, utilizar o site **csharpToJson.IO** para serializar os objetos.
* Colar a sua classe em C# (**RETIRAR TODAS ANNOTATIONS**)
* Copiar o código JSON gerado.
* Abre o **Postman =>** Seleciona o método **Post =>** Um pouco mais abaixo seleciono o **Body =>** E o tipo desse body será **JSON.**

**PADRONIZANDO ERROS DE VALIDAÇÃO E DE NEGÓCIOS**

* **Primeiro passo:** é acessar a classe Startup.
  + Dentro de ConfigureServices adicionar um serviço: service.Configure<ApiBehaviorOptions>(options => { options.SupressModelStateInvalidFilter = true }).
  + Esse método acima ele suprime que nós queremos invalidar o padrão para erros.
* **Segundo Passo**: Criar a exceptions custom na MainController.
* Criar um método protected ActionResult CustomResponse(ModelStateDictionary modelState) e um protected void NotificarErroModelInvalida(ModelStateDictionary modelState)
* MINHA REPOSITORY TEM OS METODOS GENERICOS. MEU SERVICE TEM AS AÇÕES QUE MEU “NEGOCIO” FARA.

**Padronizando erros de validação e de negócios**

Para configurar da maneira personalizada os erros, é preciso na Startup desabilitar o behavior da API após o MVC().

Upload de Imagem WEB API

Para o upload de imagens, é preciso que no client a imagem será serializada em base 64, para depois ser retornada novamente ao usuário.

**SEGURANÇA DA API**

* Primeiro passo, é adicionar o atributo [Authorize]. Ele exige que, toda pessoa que for acessar aquele modulo ou aquela parte, esteja logado.
* Caso você queira um método que possa ser acessado de maneira livre, você pode adicionar o [AllowAnonymous].

**CONFIGURAÇÃO DO IDENTITY**

* Primeiro, vamos criar uma classe static de **configuração** (IdentityConfig) fazendo a extensão do IServicesCollection (this IServicesColletion services, IConfiguration configuration).
* Nela, podemos adicionar a nossa ConnectionStrings.
* Em Startup, colocamos o método que fará referência a essa classe.

1. Vamos criar uma pasta **Data** que ficara a nossa classe que herdará de **IdentityDbContext** passando o nosso options com DbContextOptions.
   * A ideia é que esse **IdentityDbContext** não crie um banco dentro do banco de Dados, mas sim adicione as tabelas do Identity (tanto que utiliza a mesma connectionString do Context da aplicação.
2. Após isso, podemos fazer a nossa migration e update-db;
3. Para de fato começarmos a usar o Identity, na nossa IdentityConfig, precisamos adicionar o services.AddDefaultIdentity<IdentityUser() e suas extensões.

services.AddDefaultIdentity<IdentityUser>()

.AddRoles<IdentityRole>()

.AddEntityFrameworkStores<ApplicationDbContext>()

1. .AddDefaultTokenProviders();
2. Por último, adicionar app.UseAuthentication() em Configure***, ANTES DO*** app.UseMvcConfiguration();

CONTROLLER DE AUTENTICAÇÃO

1. Primeiro, é preciso criar a nossa **Controller** (AuthController) e designar a sua rota.
2. Segundo passo, vou criar uma **UserViewModel** que será utilizada para passar as confirmações de Email, senha e confirmação de senha para novos usuários logados e um apenas para login.
3. De volta em nossa classe AuthController, vamos criar um método assíncrono de **Registar** (recebendo um RegisterUserViewModel).
4. Vamos fazer a injeção de duas dependências. Uma para registro de contas/user e outra para fazer login.

private readonly SignInManager<IdentityUser> \_signInManager;

private readonly UserManager<IdentityUser> \_userManager;

1. Dentro da minha classe de Registrar, vou testar se minha Model está válida, caso não, retorno a minha CustomResponse.
   * Caso passe, vamos criar um usuário utilizando o IdentityUser.

var user = new IdentityUser

{

UserName = registerUser.Email,

Email = registerUser.Email,

EmailConfirmed = true

};

* + Vamos criar um var Result que salvará esse usuário.

var result = await \_userManager.CreateAsync(user, registerUser.Password);

* + Por fim, verificar se foi registrado com sucesso.

if (result.Succeeded)

{

await \_signInManager.SignInAsync(user, isPersistent: false);

}

foreach(var erro in result.Errors)

{

NotificarErro(erro.Description);

}

1. Para validar login já criado e logar, é bem parecido, porém recebe o parâmetro **LoginUserModel** e testa caso a senha esteja errada fechando o acesso por determinado tempo.

public async Task<ActionResult> Login(Logi nUserViewModel loginUser)

{

if (!ModelState.IsValid) return CustomResponse(ModelState);

var result = await \_signInManager.PasswordSignInAsync(loginUser.Email, loginUser.Password, false, true);

if (result.Succeeded) return CustomResponse(loginUser);

if (result.IsLockedOut)

{

NotificarErro("Usuário temporariamente bloqueado por tentativas inválidas");

return CustomResponse(loginUser);

}

NotificarErro("Usuário ou Senha incorretos");

return CustomResponse(loginUser);

}

1. }

**CUSTOMIZANDO ERROS DO IDENTITY**

1. Podemos criar uma classe de extensão que vai estender e herdar de **IdentityErrorDescriber**. Ela é uma classe do Identity que vem com alguns padrões de erros que poderão ser gerados.

**JWT – JSON WEB TOKEN**

Interface gráfica do usuário

Descrição gerada automaticamente

[**https://jwt.io/**](https://jwt.io/)

1. Criar uma classe **AppSettings** que terá três propriedades, Secret (que guardará nossa secret criptografada, ExpiracaoHoras (duração), Emissor (quem emite, no caso o próprio sistema), ValidoEm (Tempo valido).
2. Dentro de appsettings.json, adiciono mais um nó, com as informações sobre essa autenticação.
3. Na nossa classe de configuração do Identity, vamos adicionar o campo que identifica esse nosso nó, reconhecendo a extensão e injeção de dependência.
4. Criaremos uma chave para codificar o nosso secret.

**AUTORIZAÇÃO VIA CLAIM NO JWT**

O Identity fornece uma opção para o uso de Claims no ASP.NET. Porém, uma forma muito bacana, é criar uma extensão do Identity para fazer essa validação.

namespace Dev.Assuncao.API.Extensions

{

public class CustomAuthorization

{

public static bool ValidarClaimsUsuario(HttpContext context, string claimName, string claimValue)

{

return context.User.Identity.IsAuthenticated &&

context.User.Claims.Any(c => c.Type == claimName && c.Value.Contains(claimValue));

}

}

public class ClaimsAuthorizeAttribute : TypeFilterAttribute

{

public ClaimsAuthorizeAttribute(string claimName, string claimValue) : base(typeof(RequisitoClaimFilter))

{

Arguments = new object[] { new Claim(claimName, claimValue) };

}

}

public class RequisitoClaimFilter : IAuthorizationFilter

{

private readonly Claim \_claim;

public RequisitoClaimFilter(Claim claim)

{

\_claim = claim;

}

public void OnAuthorization(AuthorizationFilterContext context)

{

if (!context.HttpContext.User.Identity.IsAuthenticated)

{

context.Result = new StatusCodeResult(401);

return;

}

if (!CustomAuthorization.ValidarClaimsUsuario(context.HttpContext, \_claim.Type, \_claim.Value))

{

context.Result = new StatusCodeResult(403);

}

}

}

}

Após criar essa classe que substitui nosso filtro padrão, é preciso ir na Controller e adicionar a annotation [ClaimsAuthorize(“Type”, “Value”).

Porém, com essa configuração apenas, não será possível utilizá-la vi API, pois se no meu token, não tiver as informações de minhas claims, não saberá qual função esta liberada para aquele user

Então, onde geramos nosso JWT, devemos passar nossas claims e roles de nosso usuário.

var user = await \_userManager.FindByEmailAsync(email);

var claims = await \_userManager.GetClaimsAsync(user);

var userRoles = await \_userManager.GetRolesAsync(user);

claims.Add(new Claim(JwtRegisteredClaimNames.Sub, user.Id));

claims.Add(new Claim(JwtRegisteredClaimNames.Email, user.Email));

claims.Add(new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()));

claims.Add(new Claim(JwtRegisteredClaimNames.Nbf, ToUnixEpochDate(DateTime.UtcNow).ToString()));

claims.Add(new Claim(JwtRegisteredClaimNames.Iat, ToUnixEpochDate(DateTime.UtcNow).ToString(), ClaimValueTypes.Integer64));