Boas práticas em construção de API

11 de fevereiro de 2020 rbento 4

Fala Galera,

Nesse post venho trazendo algumas dicas importante de como podemos desenvolver nossas API. São algumas boas práticas que farão suas APIs serem mais simples e muito mais intuitivas para quem deseja utilizá-las.

Antes de falarmos das API precisamos falar sobre REST. E o que é esse tal de REST.

Introdução ao REST

O padrão arquitetural **REST** foi proposto por Roy Fielding, em torno de 2000, como uma abordagem para se criar Web Services baseado em Hipermídia (HTTP). Com isso podemos implementar um Web Rest em qualquer linguagem e os clientes que irão consumir os serviços podem ser feitos em qualquer linguagem de programação também.

Isso se deve ao fato que as respostas e as requisições são feitos em um protocolo conhecido entre os dois (API Rest e Cliente) geralmente sendo as requisições/resposta em **JSON (Javascript Object Notation)** ou em outras tecnologias como **XML (Extensible Markup Language)**.

Alguns princípios de design de APIs mais importante:

- API REST são projetadas para *recursos*, que tratam de qualquer tipo de objeto, dados ou serviço que possa ser acessado pelo cliente;
- Um recurso tem um *identificador*, o qual se trata de um URI que identifica exclusivamente esse recurso;
- Os clientes interagem com um serviço por meio da troca de *representações* de recursos (JSON ou XML)

Cuidado com a Semântica de seus serviços

Devemos tomar cuidados na organização da semântica do nossos serviços. Uma boa semântica dos serviços farão eles serem de fácil leitura e compreensão.

Veja algumas dicas para a construção de uma boa semântica:

- Organize sua API em torno de recurso;
- Concentre-se nas entidades comerciais que sua API expõe;
- O caminho de sua API ou URI (UNIFORM RESOUCE IDENTIFIER) deve ser baseado em substantivos e não em verbos

Veja aqui alguns exemplo:

Faça	Evite
GET /posts	GET /getAllPosts
GET /posts/123	GET /getPostById
POST /posts	POST /createPost

Utilize corretamente os Verbos HTTP

Os verbos HTTP definem as operações que nossos serviços fazem. Com isso o uso errado dos verbos HTTP podem ocasionar um mau entendimento de seus serviços.

O verbos HTTP mais comuns usados são:

- **GET:** RECUPERA UM RECURSO
- **POST:** CRIA UM NOVO RECURSO
- **PUT:** SUBSTITUI UM NOVO RECURSO
- **DELETE:** REMOVE UM RECURSO

Aplique corretamente o HTTP Status Code

O HTTP Status Code é a forma que seus serviços irão retornar as respostas para seus clientes. É importante aplicar corretamente o HTTP Status Code para cada operação, veja abaixo como podemos retornar corretamente o HTTP Status Code para cada Verbo HTTP.

- **GET:**
  - Retorne 200(OK) para caso de sucesso
  - Retorne 404 (NOT FOUND) se a entidade não for encontrada
- **POST:**
  - Retorne 201 (CREATED) para caso um novo recurso seja criado com sucesso
  - Retorne 400 (BAD REQUEST) caso a solicitação contenha dados inválidos
  - Retorne 422 (Unprocessable Entity) caso a solicitação caia em alguma regra de negócio
- **PUT:**
  - Retorne 200 (OK) se for atualizar um recurso existente
  - Retorne 400 (BAD REQUEST) caso a solicitação contenha dados inválidos
  - Considere utilizar 409 (CONFLICT) caso não consiga atualizar um recurso existente
- **DELETE:**
  - Retorne 204 (No Content) para sucesso
  - Retorne 404 (NOT FOUND) se a entidade não for encontrada

Respostas de erros padrões

Devemos criar padrões de respostas dos nossos serviços em caso de erros. Considere sempre utilizar um objeto no qual abrange o erro, o código do erro, mensagem do erro e uma lista de detalhes para caso de múltiplos erros.

Recentemente escrevi um post no qual podemos criar esses padrões no ASP.NET Core, fique a vontade para dá uma conferida e depois me conte o que achou dele =]

O artigo é este aqui: Criando padrões de resposta em suas APIs com ASP.NET Core

Veja como o nosso objeto de resposta de erro deve parecer:

```
//Erros genéricos
{
 “code”: “0001",
 “message”: “Acesso negado!”
}


//Erros de campos específicos
{
 “code”: “0002”,
 “errors”: [
 {“nome”: [“Nome em branco”]},
 {“idade”: [“Idade em branco”, “Você é menor de idade”]}
 ]
}
```

Utilize compressão sempre que possível

Utilize compressão sempre que possível e diminua o tráfego na rede. Com técnicas de compressão nossas respostas podem ser comprimidas, fazendo nossas APIs responder mais rápido e consequentemente poupando nossa infraestrutura de rede. Utilizando o compressor GZIP podemos comprimir nossos objetos e diminuir o seu tamanho em torno de 30% a 50% do tamanho original.

Veja como configurar no ASP.NET Core

```
1.  public class Startup
2.     {
3.         public Startup(IConfiguration configuration)
4.         {
5.             Configuration = configuration;
6.         }
7.
8.         public IConfiguration Configuration { get; }
9.
10.        // This method gets called by the runtime. Use this method to add services
    to the container.
11.        public void ConfigureServices(IServiceCollection services)
12.        {
13.            //Configurando a compactação de resposta
14.            services.AddResponseCompression(options =>
15.            {
16.                options.EnableForHttps = true;
17.                options.Providers.Add<GzipCompressionProvider>();
18.                options.MimeTypes =
    ResponseCompressionDefaults.MimeTypes.Concat(new[] { "application/json" });
19.            });
20.
21.            services.AddControllers();
22.
23.        }
24.
25.        // This method gets called by the runtime. Use this method to configure the
    HTTP request pipeline.
26.        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
27.        {
28.            if (env.IsDevelopment())
29.            {
30.                app.UseDeveloperExceptionPage();
31.            }
32.
33.            app.UseHttpsRedirection();
34.
35.            app.UseRouting();
36.
37.            app.UseAuthorization();
38.
39.            //Ative a compactação de resposta
40.            app.UseResponseCompression();
41.
42.            app.UseEndpoints(endpoints =>
43.            {
44.                endpoints.MapControllers();
```

```
45.             });
46.         }
47.     }
```

Versionamento de API

Sempre que possível também, versione sua API. O versionamento dos seus serviços irá permitir que você controle quais clientes podem usar determinadas versões de APIs e permitirá direcionar o seus clientes para utilizar versões novas ou mesmo em preview.

Abaixo um exemplo de como podemos versionar nossas APIs:

- **POR URL**: api/com/v1
- **POR SUBDOMÍNIO:** v1.api.com
- **POR HEADER:** “Accept”=“application/vnd.api.v1.json”
- **POR QUERYSTRING:** api.com/endpoint?version=2.0

Algumas dicas de segurança na construção de APIs

- Implemente OAuth para os seus clientes consumirem suas API REST;
- Habilite o CORS (CROSS-ORIGIN RESOURCE SHARING) para controlar o acesso a sua API REST
- Evite o uso de Token estático para acesso a sua API
- Sempre use HTTPS para comunicação de sua API
- Não trafegue dados sensíveis na sua API

Conclusão

Chegamos ao fim deste post que são dicas e boas práticas que aprendi durante minhas experiências. Claro, você pode discordar de alguns temas que citei acima. Mas como eu falei anteriormente, essas dicas são baseadas nas minhas experiências ao longo da minha carreira.

Espero que vocês utilizem essas dicas e que elas realmente sejam úteis para vocês.

Comenta aí quais dicas vocês gostaram mais =]

Abs e até a próxima.