07

(continuação) Documentação customizada

No vídeo anterior mostrei o começo da customização para a documentação da nossa API. Nesse texto vou complementá-lo com informações para você detalhá-la ainda mais. Vamos lá?

Instalando o pacote adicional para usar anotações

Primeiro precisaremos instalar um novo pacote que vai permitir que coloquemos anotações com informações sobre os controladores e actions da API. O nome do pacote é `Swashbuckle.Swagger.Annotations`. Instale-o via NuGet.

> **Documentação do Swashbuckle Annotations** Maiores detalhes e informações sobre o que você pode fazer com as anotações podem ser encontradas aqui (https://github.com/domaindrivendev/Swashbuckle.AspNetCore/blob/master/README.md#swashbuckleaspnetcoreannotations) Tudo o que fiz foi usando essa referência.

Próximo passo é habilitar as anotações na classe `Startup` com o método `EnableAnnotations()` nas opções de geração do swagger (método `AddSwaggerGen()`), assim:

```
services.AddSwaggerGen(options => {

    //outras opções omitidas...

    options.EnableAnnotations();
});
```

Documentando o mecanismo de autenticação da API

Um desenvolvedor que irá consumir sua API precisará saber como ele vai se autenticar. Para documentar a parte de segurança de sua API usaremos os métodos `AddSecurityDefinition` e `AddSecurityRequirement` também como opções de geração do swagger.

O primeiro método declara qual o esquema de autenticação de sua API. Em nosso caso estamos usando tokens, então nosso código para documentar isso será:

```
services.AddSwaggerGen(options => {

    //outras opções omitidas...

    options.AddSecurityDefinition("Bearer", new ApiKeyScheme {
        Name = "Authorization",
        In = "header",
        Type = "apiKey",
        Description = "Autenticação Bearer via JWT"
    });
});
```

Repare que configuramos a definição de segurança através de uma classe `ApiKeyScheme`. Informamos onde o token será enviado (`In = "header"`) e o tipo de mecanismo `apiKey`. Essas opções são usadas para permitir que o Swagger-UI se autentique na API. Já o nome e descrição são usados na documentação em si.

Em seguida é preciso indicar que operações usam o esquema definido acima. Isso é feito com o método `AddSecurityRequirement`. Como nossa API está com autenticação em todas as suas operações, vamos aplicá-lo de maneira global com o código abaixo:

```
services.AddSwaggerGen(options => {

    //outras opções omitidas...

    //definição do esquema de segurança utilizado
    options.AddSecurityDefinition("Bearer", new ApiKeyScheme {
        Name = "Authorization",
        In = "header",
        Type = "apiKey",
        Description = "Autenticação Bearer via JWT"
    });

    //que operações usam o esquema acima - todas
    options.AddSecurityRequirement(
        new Dictionary<string, IEnumerable<string>> {
            { "Bearer", new string[] { } }
    });
});
```

Essa configuração irá habilitar um botão chamado `Authorize` em seu Swagger-UI. Através dele você (e quem consultar sua documentação!) poderá autenticar-se na API para realizar testes nas operações documentadas em seguida.

Documentando as operações

Para incluir um resumo do que a operação realiza usaremos a anotação `SwaggerOperation` com o argumento `Summary`. Veja abaixo o código para documentar a operação de inclusão de um livro.

```
[SwaggerOperation(Summary = "Registra novo livro na base.")]
public IActionResult Incluir([FromForm] LivroUpload model)
```

Você também pode documentar quais os media-types que operação irá produzir. Essa info é bem importante para o dev que está consumindo sua api preparar adequadamente seu código. Para isso usamos outro argumento na anotação, o `Produces`, que espera um array de strings.

Veja um exemplo na operação de recuperação de um livro:

```
[SwaggerOperation(
    Summary = "Recupera o livro identificado por seu {id}.",
    Produces = new[] {"application/json", "application/xml"}
)]
[ProducesResponseType(statusCode: 200, Type = typeof(LivroApi))]
[ProducesResponseType(statusCode: 500, Type = typeof(ErroResponse))]
[ProducesResponseType(404)]
public IActionResult Recuperar(int id)
```

```
public IActionResult Recuperar(int id)
```

Essa documentação deve ser feita para cada operação. Portanto se sua API é grande, sugiro que comece logo no momento de criação de cada action/operação.

Depois que terminar esse passo, o Swagger-UI já começará a mostrar suas operações com as informações que digitou no resumo, além de uma lista de media-types que serão aceitos (lembra do assunto content negotiation?).

Agrupando as operações usando Tags

Nossa próxima tarefa é agrupar as operações relacionadas a livros e as relacionadas a listas de leitura. Isso já é feito no código quando colocamos as actions dentro de cada controlador específico. Mas você pode desejar agrupar operações que estão distribuídas em vários controladores.

Esse agrupamento é feito através de **tags**. Sua declaração fica em cada operação usando mais um argumento na anotação `SwaggerOperation` chamado `Tags` que também espera um array de strings. Exemplo:

```
[SwaggerOperation(
    Summary = "Recupera o livro identificado por seu {id}.",
    Tags = new[] { "Livros" },
    Produces = new[] {"application/json", "application/xml"}
)]
[ProducesResponseType(statusCode: 200, Type = typeof(LivroApi))]
[ProducesResponseType(statusCode: 500, Type = typeof(ErroResponse))]
[ProducesResponseType(404)]
public IActionResult Recuperar(int id)
```

Com essa declaração de tags sua documentação agrupará as operações embaixo de cada tag. Eu defini duas tags, uma chamada `Livros` e outra chamada `Listas`.

Documentando os parâmetros

Outra coisa importante é documentar os parâmetros esperados pelas operações, seus tipos e obrigatoriedade. Isso é feito com a anotação `SwaggerParameter`. Veja o exemplo para a recuperação de uma lista de leitura:

```
[HttpGet("{tipo}")]
[SwaggerOperation(
    Summary = "Recupera a lista de leitura identificada por seu {tipo}.",
    Tags = new[] { "Listas" },
    Produces = new[] { "application/json", "application/xml" }
)]
[ProducesResponseType(200, Type = typeof(Lista))]
[ProducesResponseType(500, Type = typeof(ErroResponse))]
public IActionResult Recuperar(
    [FromRoute] [SwaggerParameter("Tipo da lista a ser obtida.")] TipoListaLeitura tipo)
{
    var lista = CriaLista(tipo);
    return Ok(lista);
}
```

A geração do Swagger irá colocar essa descrição ao lado do parâmetro. Além disso também ira indicar se ele é obrigatório ou não, o que é inferido pelos metadados da rota definida para a operação.

Outra coisa é que como o parâmetro tipo é um enumerado, o Swagger irá indicar quais são seus valores válidos. Contudo, observe que esses valores são 0, 1 e 2. Não seria legal se mostrássemos os valores descritivos do enumerado (paraLer, lendo e lidos)? Isso é possível com os métodos `DescribeAllEnumsAsStrings()` e `DescribeStringEnumsInCamelCase()` assim:

```
services.AddSwaggerGen(options => {

    //outras opções omitidas...

    //descrevendo enumerados como strings
    options.DescribeAllEnumsAsStrings();
    options.DescribeStringEnumsInCamelCase();
});
```

Documentando operações de forma global

Até agora documentamos uma operação de forma individual usando a anotação `SwaggerOperation`. Mas e quando houver situações onde a mesma documentação se aplica a todas as operações? Em nossa API temos um caso assim: toda operação que não tiver um token válido irá produzir como resposta o código 401. Imagina se tivéssemos que anotar todas as operações com `[ProducesResponseType(401)]`? E olha que nossa API é pequena hein...

Afinal, como aplicar uma documentação de forma global? Para isso existe a interface `IOperationFilter` que permite aplicar um código a cada operação sendo documentada. Em nosso caso iremos adicionar mais um tipo de resposta (401).

Vamos implementar a interface com a classe `AuthResponsesOperationFilter` ...

```
public class AuthResponsesOperationFilter : IOperationFilter
{
}
```

...que precisa de um método `Apply` cujos argumentos são a operação sendo documentada e o contexto:

```
public class AuthResponsesOperationFilter : IOperationFilter
{
    public void Apply(Operation operation, OperationFilterContext context)
    {
    }
}
```

O que precisamos fazer é adicionar uma nova resposta na operação:

```
public class AuthResponsesOperationFilter : IOperationFilter
{
    public void Apply(Operation operation, OperationFilterContext context)
    {
        operation.Responses.Add(
            "401",
            new Response { Description = "Unauthorized" });
```

```
    }
}
```

Por último é necessário ligar esse filtro na configuração da API. Usaremos novamente as opções do método `AddSwaggerGen()`:

```
services.AddSwaggerGen(options => {

    //outras opções omitidas...

    //adicionando o filtro para incluir respostas 401 nas operações
    options.OperationFilter<AuthResponsesOperationFilter>();
});
```

Isso fará com que a documentação de cada operação tenha mais uma resposta 401 com a descrição "Unauthorized".

Adicionando info após a geração da documentação

E quando precisamos customizar a documentação gerada pelo SwaggerGen para adicionar ou modificar alguma informação? Para isso existe a interface `IDocumentFilter`. Vamos colocar uma descrição genérica em cada tag usando esse filtro.

Vamos implementar a interface com a classe `TagDescriptionsDocumentFilter`...

```
public class TagDescriptionsDocumentFilter : IDocumentFilter
{
}
```

...que precisa de um método `Apply` cujos argumentos são a documentação gerada e o contexto:

```
public class TagDescriptionsDocumentFilter : IDocumentFilter
{
    public void Apply(SwaggerDocument swaggerDoc, DocumentFilterContext context)
    {
    }
}
```

Vamos colocar nesse método a descrição das duas tags criadas anteriormente:

```
public class TagDescriptionsDocumentFilter : IDocumentFilter
{
    public void Apply(SwaggerDocument swaggerDoc, DocumentFilterContext context)
    {
        swaggerDoc.Tags = new[] {
            new Tag { Name = "Livros", Description = "Consulta e mantém os livros." },
            new Tag { Name = "Listas", Description = "Consulta as listas de leitura." }
        };
    }
}
```

Por fim devemos ligar esse filtro na configuração da API com o método `AddSwaggerGen()`:

```
services.AddSwaggerGen(options => {

    //outras opções omitidas...

    //adicionando o filtro para incluir descrições nas tags
    options.DocumentFilter<TagDescriptionsDocumentFilter>();
});
```

Código do projeto com a documentação final

Nessa atividade (https://cursos.alura.com.br/course/api-rest-net-core-2-padronizacao/task/49083) eu disponibilizo o código-fonte do projeto com todas as definições de documentação que expliquei aqui.