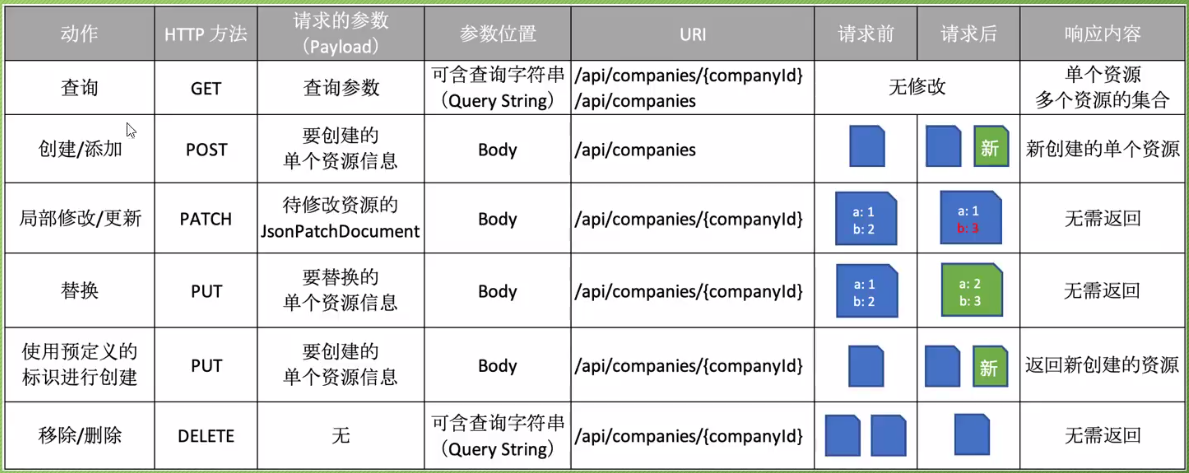
# Webapi

Webapi:使用http协议并通过网络调用的API (大多数API都不是RESTful API)

# RESTful API

示例：



# http状态码

## 1xx：属于信息性的状态码，web api并不使用1xx的状态码

## 2xx：请求执行成功

200-OK,表示请求成功(return Ok();)

201-Created，请求成功并创建了资源

204-No Content，请求成功，但不应该返回任何东西，例如删除操作

## 3xx:用于跳转，例如告诉搜索引擎某个页面的网址已经永久的改变。绝大多数API不需要使用这类状态码

## 4xx：客户端错误

400-Bad Request:表示API消费者发送到服务器的请求是有错误的

401-Unauthorized:表示没有提供授权信息或者提供授权信息不正确

403-Forbidden:表示身份认证已经成功，但是已认证的用户无法访问请求的资源

404-Not Found:表示请求的资源不存在（return NotFound();）

405-Method Not Allowed:当尝试发送请求资源的时候，使用了不被支持的http方法

406-Not Accepted:表示API消费者请求的表述格式（json.xml之类）并不被API所支持，并且API不会提供默认的表述格式

409-Conflict:表示请求与服务器当前状态冲突。如要创建的资源已存在或者要更新的资源在服务器也进行更新

415-Unsupported Media Type:有些请求必须带着数据发往服务器，这些数据都是属于特定的媒体类型，如果API不支持该媒体类型，就会返回415

422-Unprocessable entity:是Http扩展协议的一部分，说明服务器已经懂得了实体的Content-Type,实体的语法也没有问题，但是服务器仍然无法处理这个实体数据，通常表示语意上有错误或表示实体验证的错误

## 5xx:服务器端的错误

500-Internal server error:表示服务器出现错误

# 内容协商(请求和响应的内容格式)

根据API消费者设置的媒体类型，响应输出对应格式的结果

## Accept Header

Application/json（默认）

Application/xml

…

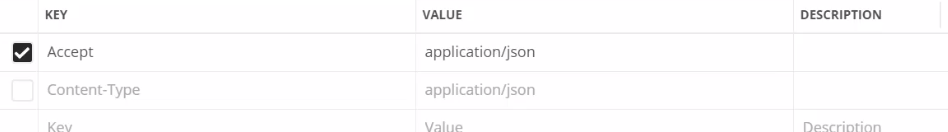
## Content-Type Header

Application/json（默认）

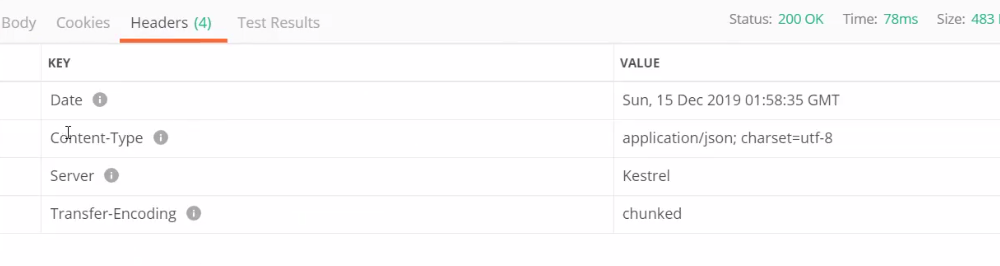
Application/xml

…

## 请求：



## 响应：



## 设置**请求和响应格式不一致时返回**406状态码：

**services.AddControllers(setup =>**

**{**

**setup.ReturnHttpNotAcceptable = true;//该值设置为true，请求和响应格式不一致时会返回406状态码**

**});**

## **设置支持输出xml格式**

**services.AddControllers(setup =>**

**{**

**//旧的方法**

**//setup.OutputFormatters.Add(new XmlDataContractSerializerOutputFormatter());//支持xml, OutputFormatters是个集合**

**//setup.OutputFormatters.Insert(0, new XmlDataContractSerializerOutputFormatter());//放在第一位则默认的就是xml格式**

**}).AddXmlDataContractSerializerFormatters();//Input和Output都添加xml格式**

# ActionResult<T>

# AutoMapper: 对象到对象的映射器

1. Nuget安装AutoMapper.Microsoft.Extensions.DependencyInjection
2. ConfigServices方法注册服务

**services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());**

1. **类继承Profile**

**示例：**

**public class EmployeeProfile: Profile**

**{**

**public EmployeeProfile()**

**{**

**CreateMap<Employee, EmployeeDto>()**

**.ForMember(dest => dest.Name,**

**opt => opt.MapFrom(src => $"{src.FirstName} {src.LastName}"))**

**.ForMember(dest => dest.GenderDisplay**

**, opt => opt.MapFrom(src => src.Gender.ToString()))**

**.ForMember(dest => dest.Age,**

**opt => opt.MapFrom(src => DateTime.Now.Year - src.DateOfBirth.Year));**

**CreateMap<EmployeeAddDto, Employee>();**

**CreateMap<EmployeeUpdateDto, Employee>();**

**CreateMap<Employee, EmployeeUpdateDto>();**

**}**

**}**

1. **代码中映射**

**\_mapper.Map<IEnumerable<CompanyDto>>(companies);**

# **异常处理（自定义异常信息）**

**if (env.IsDevelopment())**

**{**

**app.UseDeveloperExceptionPage();**

**}**

**else**

**{**

**//存在未处理异常**

**app.UseExceptionHandler(appBuilder =>**

**{**

**appBuilder.Run(async context =>**

**{**

**context.Response.StatusCode = 500;**

**await context.Response.WriteAsync("Unexpected Error!");**

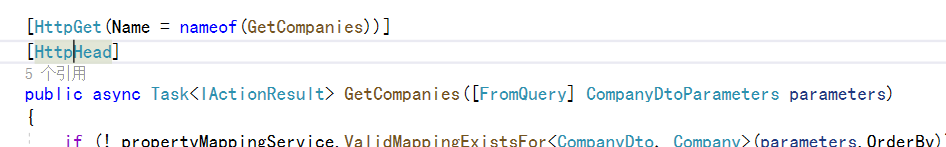
**});**

**});**

**}**

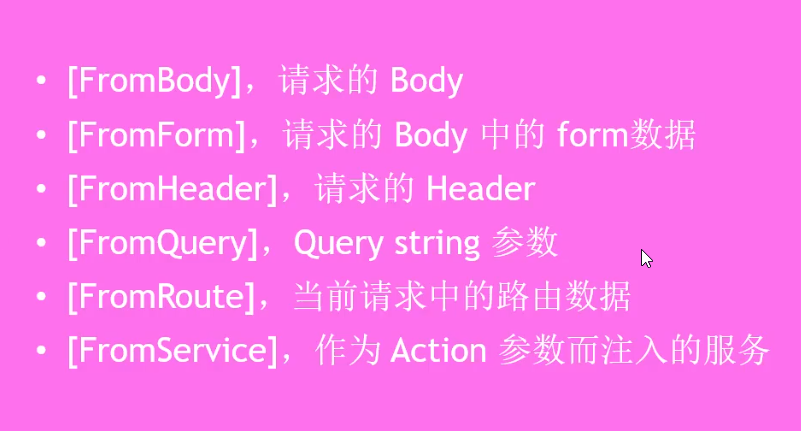
# HTTP HEAD

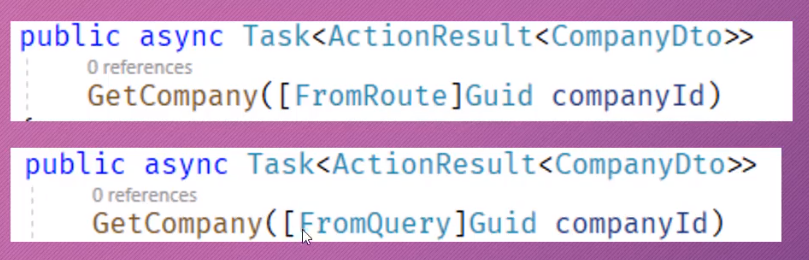
HEAD几乎和GET一样，只是HEAD的API不应该返回响应的body，HEAD可以用来在资源上获取一些信息



# 过滤和搜索

## Binding Source 特性



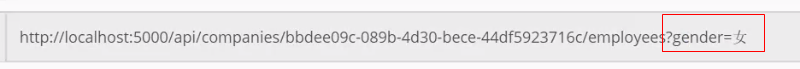


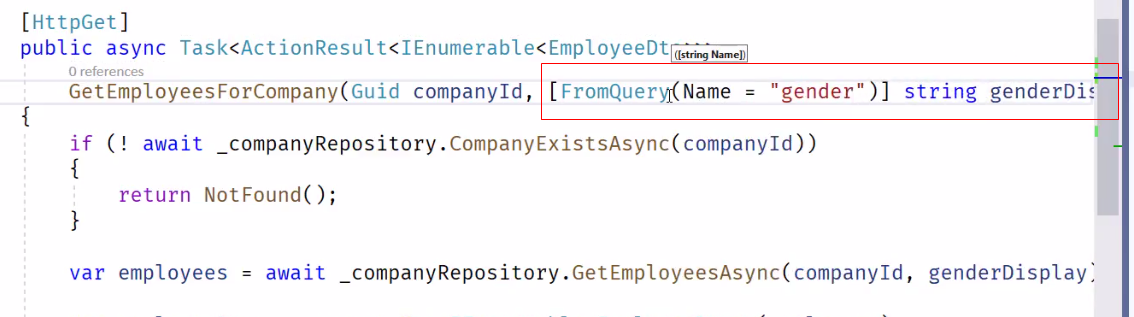
## ApiController更改后的规则



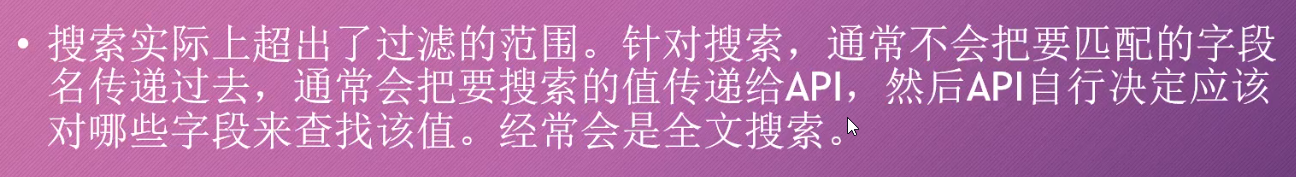
## 过滤：根据条件限定返回的集合

过滤的意思就是把某个字段的名字以及想要让该字段匹配的值一起传递给API，并将这些作为返回集合的一部分





## 搜索：根据预定义的规则，把符合条件的数据添加到集合



# Http Options 和 XM支持

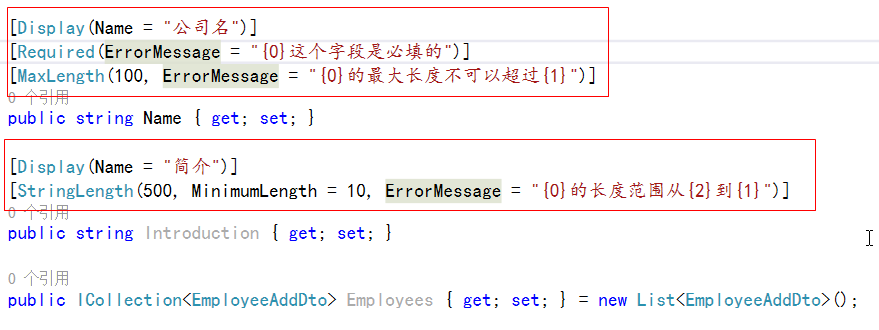
Option请求可以获得针对某个api的通信选项信息

# 输入验证

## 定义验证规则



## Attribute示例



## 自定义Attribute示例

**public class EmployeeNoMustDifferentFromFirstNameAttribute : ValidationAttribute**

**{**

**protected override ValidationResult IsValid(object value, ValidationContext validationContext)**

**{**

**var addDto = (EmployeeAddOrUpdateDto)validationContext.ObjectInstance;**

**if (addDto.EmployeeNo == addDto.FirstName)**

**{**

**return new ValidationResult(ErrorMessage, new[] { nameof(EmployeeAddOrUpdateDto) });**

**}**

**return ValidationResult.Success;**

**}**

**}**



## IValidatableObject接口

**public abstract class EmployeeAddOrUpdateDto : IValidatableObject**

**{**

**[Display(Name = "员工号")]**

**[Required(ErrorMessage = "{0}是必填项")]**

**[StringLength(10, MinimumLength = 10, ErrorMessage = "{0}的长度是{1}")]**

**public string EmployeeNo { get; set; }**

**public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)**

**{**

**if (FirstName == LastName)**

**{**

**yield return new ValidationResult("姓和名不能一样",**

**new[] { nameof(FirstName), nameof(LastName) });**

**}**

**}**

## **其他验证方式**

### FluentValidation

1. 容易创建复杂的验证规则
2. 验证规则与Model分开
3. 容易进行单元测试

## **如果验证错误**

### 默认的



### **自定义**

**//处理输入验证错误**

**Services.ConfigureApiBehaviorOptions(setup =>**

**{**

**setup.InvalidModelStateResponseFactory = context =>**

**{**

**var problemDetails = new ValidationProblemDetails(context.ModelState)**

**{**

**Type = "http://www.baidu.com",**

**Title = "有错误！！！",**

**Status = StatusCodes.Status422UnprocessableEntity,**

**Detail = "请看详细信息",**

**Instance = context.HttpContext.Request.Path**

**};**

**problemDetails.Extensions.Add("traceId", context.HttpContext.TraceIdentifier);**

**return new UnprocessableEntityObjectResult(problemDetails)**

**{**

**ContentTypes = { "application/problem+json" }**

**};**

**};**

**});**

# **缓存**

# 并发控制

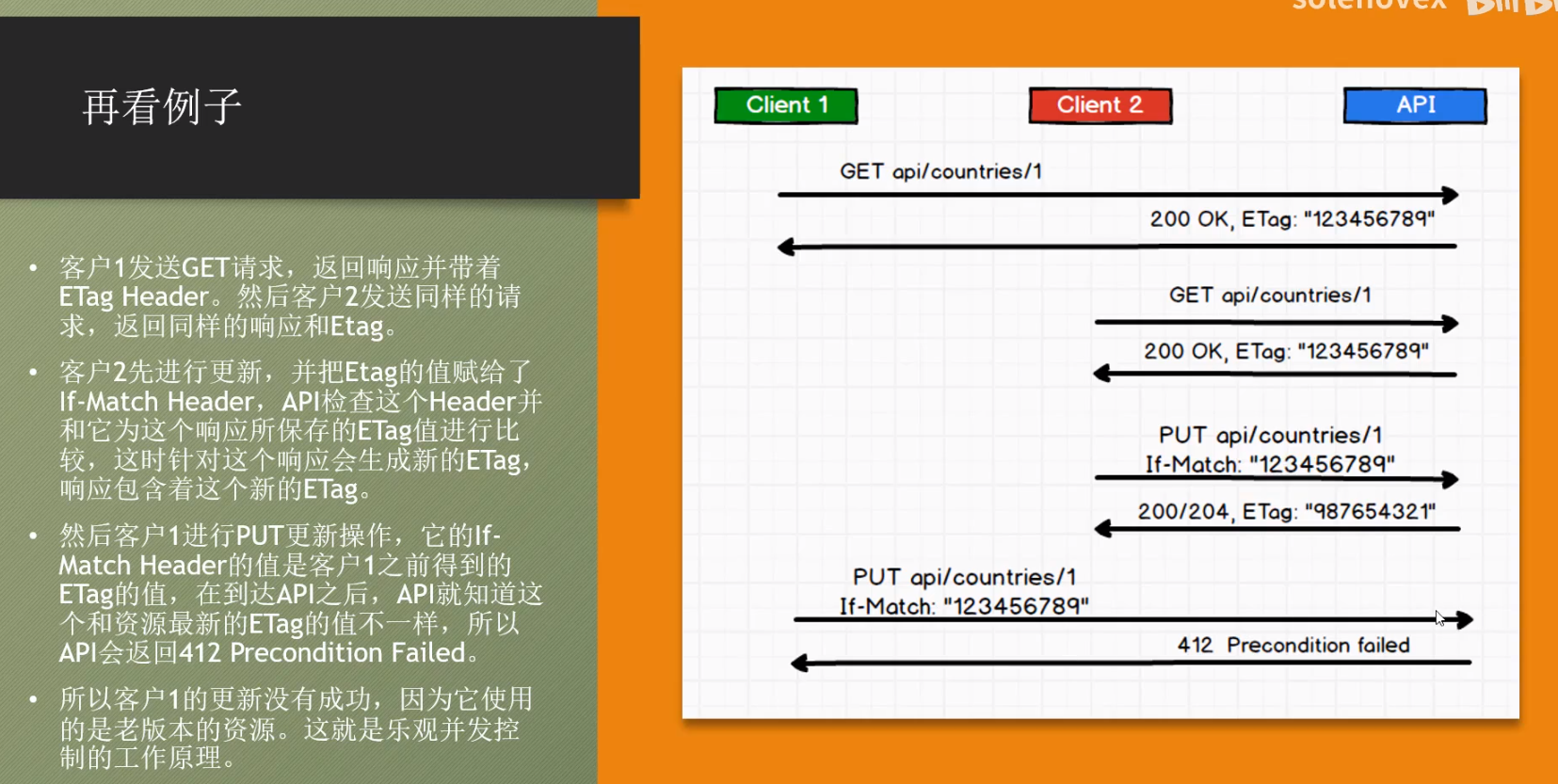
## 悲观并发控制



## 乐观并发控制



### 例子



# NLog保存日志到数据库

## 配置文件

第一步：首先添加包NLog.Config （可通过微软添加包命令Install-Package 包名进行添加，也可以通过管理NuGet程序包进行添加），添加成功后会生成NLog.config配置文件。并对该配置文件进行配置。详细配置可参考Git上 NLog说明。

<?xml version="1.0" encoding="utf-8" ?>

<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.nlog-project.org/schemas/NLog.xsd NLog.xsd"

autoReload="true"

throwExceptions="false"

internalLogLevel="Warn"

internalLogFile="Logs/nlog-internal.log">

<!--internalLogLevel="Off"-->

<!-- optional, add some variables

https://github.com/nlog/NLog/wiki/Configuration-file#variables

-->

<variable name="myvar" value="myvalue"/>

<!--

See https://github.com/nlog/nlog/wiki/Configuration-file

for information on customizing logging rules and outputs.

-->

<targets>

<!--

add your targets here

See https://github.com/nlog/NLog/wiki/Targets for possible targets.

See https://github.com/nlog/NLog/wiki/Layout-Renderers for the possible layout renderers.

-->

<!--

Write events to a file with the date in the filename.

<target xsi:type="File" name="f" fileName="${basedir}/logs/${shortdate}.log"

layout="${longdate} ${uppercase:${level}} ${message}" />

-->

<!-- write logs to file -->

<target xsi:type="File" name="allfile" fileName="Logs/${date:format=yyyyMM}/nlog-all-${shortdate}.log"

layout="${longdate}|${event-properties:item=EventId.Id}|${uppercase:${level}}|${logger} ${newline}${message} ${exception} ${newline}" />

<target xsi:type="File" name="ownFile-web" fileName="Logs/${date:format=yyyyMM}/nlog-own-${shortdate}.log"

layout="${longdate}|${event-properties:item=EventId.Id}|${uppercase:${level}}|${logger} ${newline}${message} ${exception} ${newline} --- |url: ${aspnet-request-url}|action: ${aspnet-mvc-action}" />

<target xsi:type="Null" name="blackhole" />

<target xsi:type="Database" name="database">

<connectionString>${var:connectionString}</connectionString>

<commandText>

insert into syslogs (Application,Levels,Operatingtime,Operatingaddress,Userid,Logger,Callsite,Requesturl,Referrerurl,Action,Message,Exception)

values (@application,@levels,@operatingtime,@operatingaddress,@userid,@logger,@callSite,@requesturl,@referrerurl,@action,@message,@exception);

</commandText>

<parameter name="@application" layout="WebApi" />

<parameter name="@levels" layout="${level}" />

<parameter name="@operatingTime" layout="${date}" />

<parameter name="@operatingaddress" layout="${aspnet-Request-IP}" />

<parameter name="@userid" layout="1" />

<parameter name="@logger" layout="${logger}" />

<parameter name="@callSite" layout="${callsite}" />

<parameter name="@requesturl" layout="${aspnet-request-url}" />

<parameter name="@referrerurl" layout="${aspnet-request}" />

<parameter name="@action" layout="${aspnet-mvc-action}" />

<parameter name="@message" layout="${message}" />

<parameter name="@exception" layout="${exception:tostring}" />

</target>

</targets>

<rules>

<!-- add your logging rules here -->

<!--

Write all events with minimal level of Debug (So Debug, Info, Warn, Error and Fatal, but not Trace) to "f"

<logger name="\*" minlevel="Debug" writeTo="f" />

-->

<!--All logs, including from Microsoft-->

<!--minlevel 改为Trace 跟踪全部 Error 只捕获异常-->

<logger name="\*" minlevel="Error" writeTo="allfile" />

<!--Skip Microsoft logs and so log only own logs-->

<logger name="Microsoft.\*" minlevel="Trace" writeTo="blackhole" final="true" />

<logger name="\*" minlevel="Trace" writeTo="ownFile-web" />

<logger name="\*" minlevel="Trace" writeTo="database" /> 1

</rules>

</nlog>

## 在数据库中添加该记录日志表

## 修改Program.cs文件

添加包NLog.Web.AspNetCore，在Program.cs中的WebHost加入".UseNLog()"（该属于程序集NLog.Web，需要添加引用using NLog.Web;）,即为添加nlog.

namespace WebApi

{

public class Program

{

public static void Main(string[] args)

{

CreateWebHostBuilder(args).Build().Run();

}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>

WebHost.CreateDefaultBuilder(args)

.UseStartup<Startup>()

.UseNLog(); //加入nlog日志

}

}

## 修改Startup.cs的Configure方法

在Startup.cs中的Configure方法中添加记日志代码，即需要加载的配置文件和配置日志写入数据库连接字符串代码。注意：为避免中文乱码问题需要添加System.Text.Encoding.CodePages包。

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

#region Nlog记日志

//将日志记录到数据库 config/NLog.config NLog.LogManager.LoadConfiguration("nlog.config").GetCurrentClassLogger(); NLog.LogManager.Configuration.Variables["connectionString"] = Configuration.GetConnectionString("DefaultConnection"); Encoding.RegisterProvider(CodePagesEncodingProvider.Instance); //避免日志中的中文输出乱码

#endregion

if (env.IsDevelopment())

app.UseDeveloperExceptionPage();

else

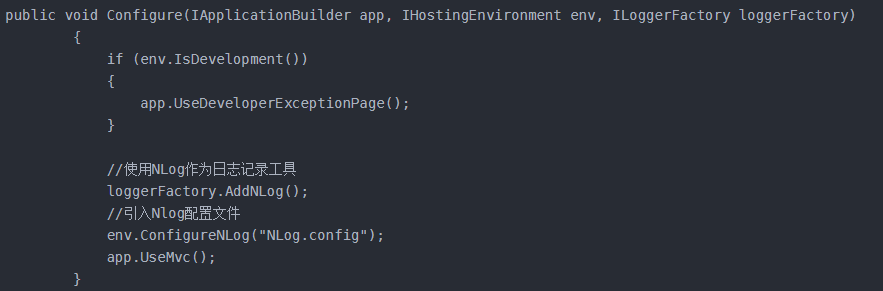
app.UseHsts();

app.UseHttpsRedirection();

app.UseMvc();

}

或者



## 使用

public class UsersController : Controller

{

private readonly ILogger logger;

public UsersController(ILoggerFactory loggerFactory)

{

this.logger = loggerFactory.CreateLogger<UsersController>();

#region 测试日志

logger.LogTrace("开发阶段调试，可能包含敏感程序数据", 1);

logger.LogDebug("开发阶段短期内比较有用，对调试有益。");

logger.LogInformation("你访问了首页。跟踪程序的一般流程。");

logger.LogWarning("警告信息！因程序出现故障或其他不会导致程序停止的流程异常或意外事件。");

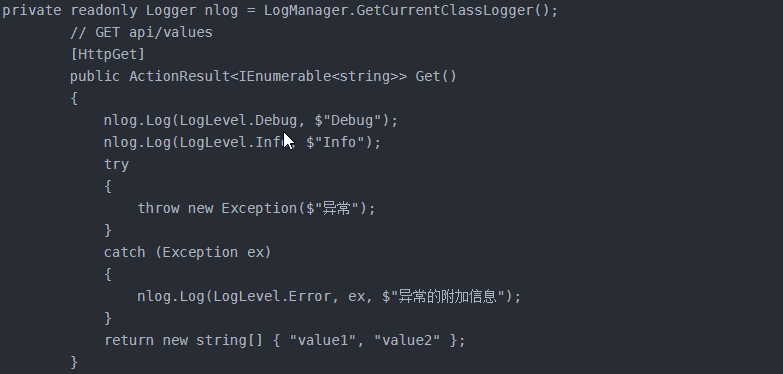
logger.LogError("错误信息。因某些故障停止工作");

logger.LogCritical("程序或系统崩溃、遇到灾难性故障！！！");

#endregion

}

或者



# Swagger

## 安装Swashbuckle.AspNetCore

Nuget安装

## 添加并配置 Swagger 中间件

将 Swagger 生成器添加到 Startup.ConfigureServices 方法中的服务集合中：

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<TodoContext>(opt =>

opt.UseInMemoryDatabase("TodoList"));

services.AddMvc()

.SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

// Register the Swagger generator, defining 1 or more Swagger documents

services.AddSwaggerGen();

}

在 Startup.Configure 方法中，启用中间件为生成的 JSON 文档和 Swagger UI 提供服务：public void Configure(IApplicationBuilder app)

{

// Enable middleware to serve generated Swagger as a JSON endpoint.

app.UseSwagger();

// Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),

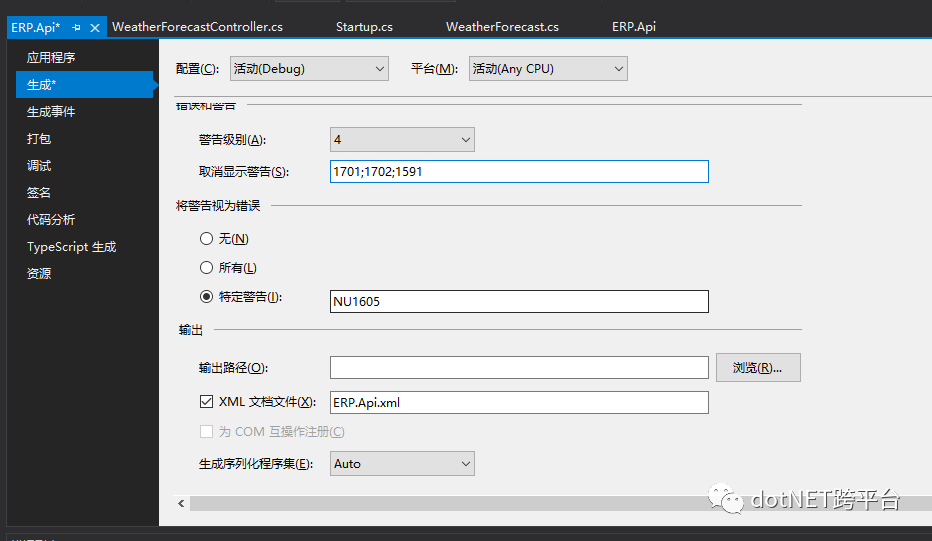
// specifying the Swagger JSON endpoint.

app.UseSwaggerUI(c => { c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1"); });

}

## 生成方法说明xml和取消警告

Xml文件输出路径为默认（startup.cs同一级）即可



## API 信息和说明

传递给 AddSwaggerGen 方法的配置操作会添加诸如作者、许可证和说明的信息：

using Microsoft.OpenApi.Models;

使用 OpenApiInfo 类修改 UI 中显示的信息：

// Register the Swagger generator, defining 1 or more Swagger documents

services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo

{

Version = "v1",

Title = "ToDo API",

Description = "A simple example ASP.NET Core Web API",

TermsOfService = new Uri("https://example.com/terms"),

Contact = new OpenApiContact

{

Name = "Shayne Boyer",

Email = string.Empty,

Url = new Uri("https://twitter.com/spboyer"),

},

License = new OpenApiLicense

{

Name = "Use under LICX",

Url = new Uri("https://example.com/license"),

}

});

// 为 Swagger JSON and UI设置xml文档注释路径

var basePath = Path.GetDirectoryName(AppContext.BaseDirectory);//获取应用程序所在目录(绝对，不受工作目录影响，建议采用此方法获取路径)

var xmlPath = Path.Combine(basePath, "ERP.Api.xml");

c.IncludeXmlComments(xmlPath);

});

## 修改默认打开页面

修改launchSetting.json文件

"launchUrl": "swagger/index.html",

# JWT(JSON WEB TOKEN)

## 什么是JWT

Json web token (JWT), 是为了在网络应用环境间传递声明而执行的一种基于JSON的开放标准（[(RFC 7519](https://link.jianshu.com?t=https:/tools.ietf.org/html/rfc7519)).该token被设计为紧凑且安全的，特别适用于分布式站点的单点登录（SSO）场景。JWT的声明一般被用来在身份提供者和服务提供者间传递被认证的用户身份信息，以便于从资源服务器获取资源，也可以增加一些额外的其它业务逻辑所必须的声明信息，该token也可直接被用于认证，也可被加密。

## 基于token的认证和传统的session认证的区别

### 传统的session认证

我们知道，http协议本身是一种无状态的协议，而这就意味着如果用户向我们的应用提供了用户名和密码来进行用户认证，那么下一次请求时，用户还要再一次进行用户认证才行，因为根据http协议，我们并不能知道是哪个用户发出的请求，所以为了让我们的应用能识别是哪个用户发出的请求，我们只能在服务器存储一份用户登录的信息，这份登录信息会在响应时传递给浏览器，告诉其保存为cookie,以便下次请求时发送给我们的应用，这样我们的应用就能识别请求来自哪个用户了,这就是传统的基于session认证。

但是这种基于session的认证使应用本身很难得到扩展，随着不同客户端用户的增加，独立的服务器已无法承载更多的用户，而这时候基于session认证应用的问题就会暴露出来.

### 基于session认证所显露的问题

**Session**: 每个用户经过我们的应用认证之后，我们的应用都要在服务端做一次记录，以方便用户下次请求的鉴别，通常而言session都是保存在内存中，而随着认证用户的增多，服务端的开销会明显增大。

**扩展性**: 用户认证之后，服务端做认证记录，如果认证的记录被保存在内存中的话，这意味着用户下次请求还必须要请求在这台服务器上,这样才能拿到授权的资源，这样在分布式的应用上，相应的限制了负载均衡器的能力。这也意味着限制了应用的扩展能力。

**CSRF**: 因为是基于cookie来进行用户识别的, cookie如果被截获，用户就会很容易受到跨站请求伪造的攻击。

## 基于token的鉴权机制

基于token的鉴权机制类似于http协议也是无状态的，它不需要在服务端去保留用户的认证信息或者会话信息。这就意味着基于token认证机制的应用不需要去考虑用户在哪一台服务器登录了，这就为应用的扩展提供了便利。

### 流程

1. 用户使用用户名密码来请求服务器
2. 服务器进行验证用户的信息
3. 服务器通过验证发送给用户一个token
4. 客户端存储token，并在每次请求时附送上这个token值
5. 服务端验证token值，并返回数据

这个token必须要在每次请求时传递给服务端，它应该保存在请求头里， 另外，服务端要支持CORS(跨来源资源共享)策略，一般我们在服务端这么做就可以了Access-Control-Allow-Origin: \*

## JWT的构成

第一部分我们称它为头部（header),第二部分我们称其为载荷（payload, 类似于飞机上承载的物品)，第三部分是签证（signature)。

### Header

jwt的头部承载两部分信息：

* 声明类型，这里是jwt
* 声明加密的算法 通常直接使用 HMAC SHA256

完整的头部就像下面这样的JSON：



然后将头部进行base64编码（该编码是可以对称解密的),构成了第一部分.



### Playload

载荷就是存放有效信息的地方。这个名字像是特指飞机上承载的货品，这些有效信息包含三个部分：

* 标准中注册的声明
* 公共的声明
* 私有的声明

#### **标准中注册的声明** (建议但不强制使用) ：

* **iss**: jwt签发者
* **sub**: jwt所面向的用户
* **aud**: 接收jwt的一方
* **exp**: jwt的过期时间，这个过期时间必须要大于签发时间
* **nbf**: 定义在什么时间之前，该jwt都是不可用的.
* **iat**: jwt的签发时间
* **jti**: jwt的唯一身份标识，主要用来作为一次性token,从而回避重放攻击。

#### **公共的声明**

公共的声明可以添加任何的信息，一般添加用户的相关信息或其他业务需要的必要信息.但不建议添加敏感信息，因为该部分在客户端可解密.

#### **私有的声明**

私有声明是提供者和消费者所共同定义的声明，一般不建议存放敏感信息，因为base64是对称解密的，意味着该部分信息可以归类为明文信息。

定义一个payload:



然后将其进行base64加密，得到Jwt的第二部分。



### Signature

jwt的第三部分是一个签证信息，这个部分需要base64加密后的header和base64加密后的payload使用.连接组成的字符串，然后通过header中声明的加密方式进行加盐secret组合加密，然后就构成了jwt的第三部分。：

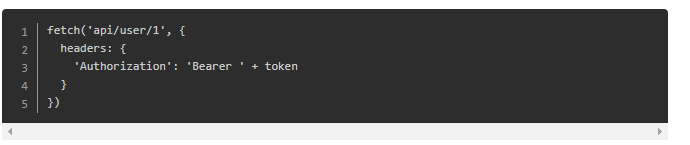
* header (base64后的)
* payload (base64后的)
* secret



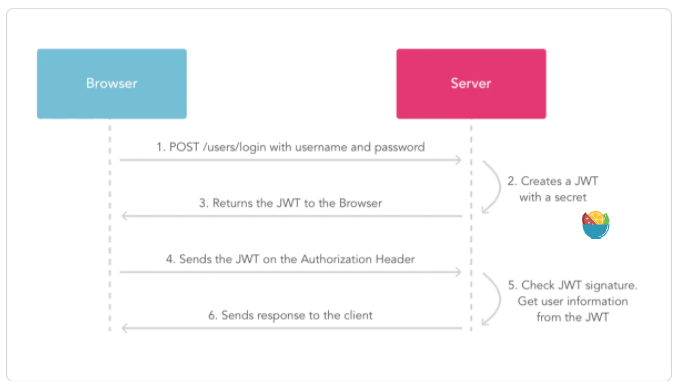
注意：secret是保存在服务器端的，jwt的签发生成也是在服务器端的，secret就是用来进行jwt的签发和jwt的验证，所以，它就是你服务端的私钥，在任何场景都不应该流露出去。一旦客户端得知这个secret, 那就意味着客户端是可以自我签发jwt了。

## 如何应用

一般是在请求头里加入Authorization，并加上Bearer标注：



服务端会验证token，如果验证通过就会返回相应的资源。整个流程就是这样的:



## 优点

* 因为json的通用性，所以JWT是可以进行跨语言支持的，像JAVA,JavaScript,NodeJS,PHP等很多语言都可以使用。
* 因为有了payload部分，所以JWT可以在自身存储一些其他业务逻辑所必要的非敏感信息。
* 便于传输，jwt的构成非常简单，字节占用很小，所以它是非常便于传输的。
* 它不需要在服务端保存会话信息, 所以它易于应用的扩展

## 安全相关

* 不应该在jwt的payload部分存放敏感信息，因为该部分是客户端可解密的部分。
* 保护好secret私钥，该私钥非常重要。
* 如果可以，请使用https协议