https://mp.weixin.qq.com/s/7135y3MkUlPIp-flfwscig

<https://www.bilibili.com/video/av73194514?p=2>

本笔记包含的内容：验证、简单的授权（角色，策略）、动态的授权、授权\_策略详解：基于自定义的策略授权

**验证**

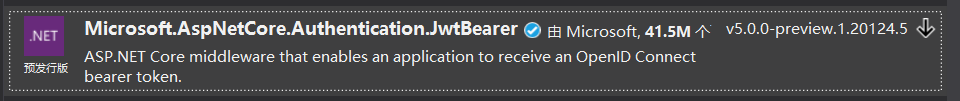
在要验证的方法前加上一个[Authorize]的Attribute，然后在ConfigureSeivices里AddAuthentication(); 这是认证方案，常用的是JWT，我们就需要配置一下JWT的认证方案。

Ids4和JWT的关系：ids4用到了jwt的一些内容。JWT是JSON Web Tokens，是Json web令牌。

[Authorize]是授权；

AddAuthentication();是认证；

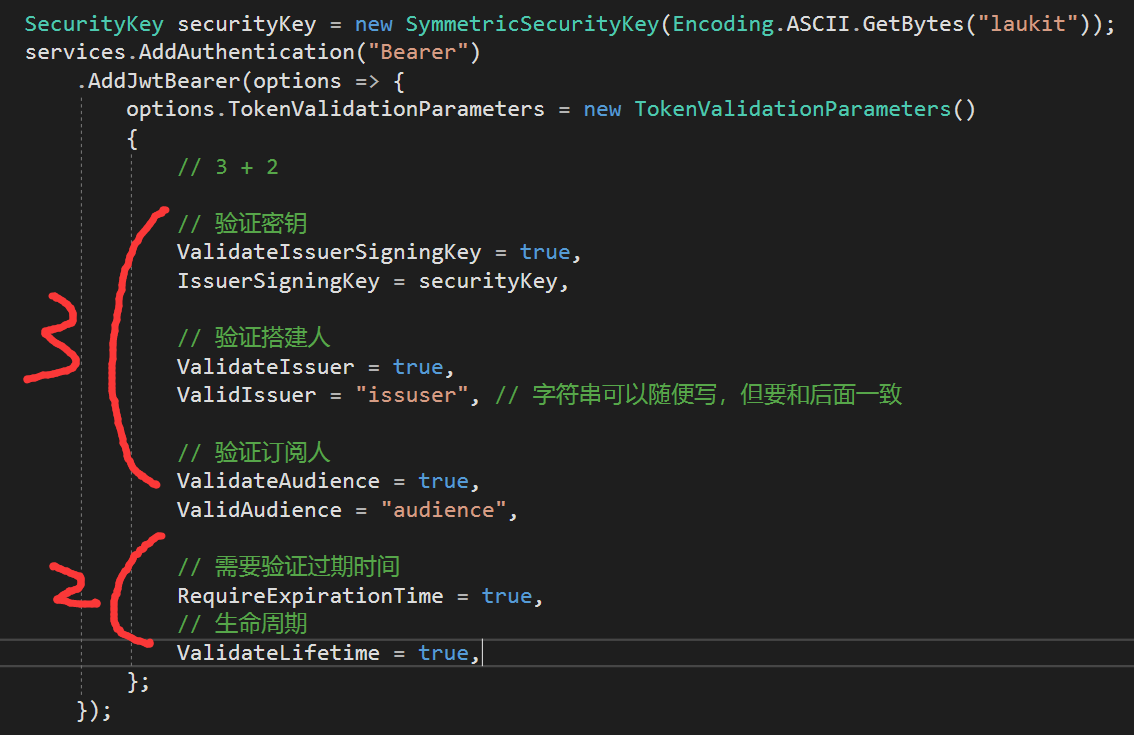
去Nuget搜JWT，安装下面这个包：



先new一个对称加密的密钥：其实一会才能用到，而且下面的密钥可能有bug。

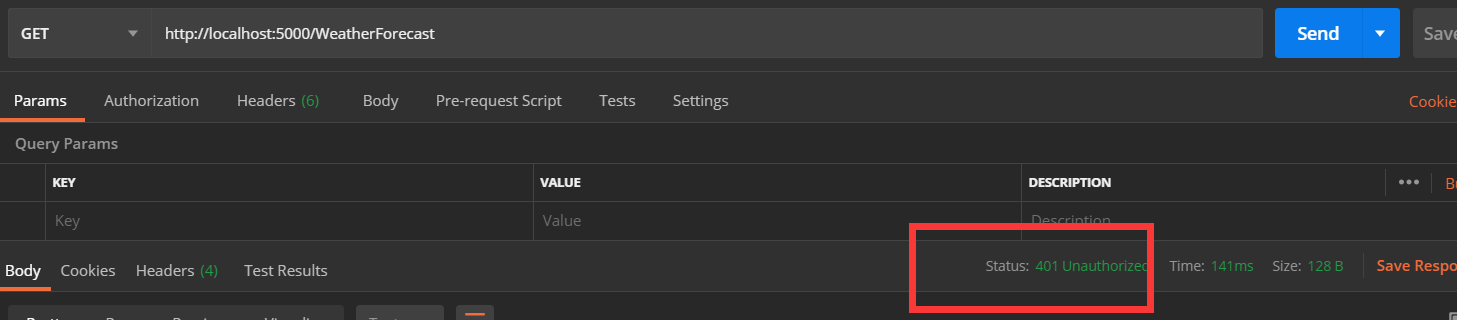


JWT封装需要的配置是3+2：



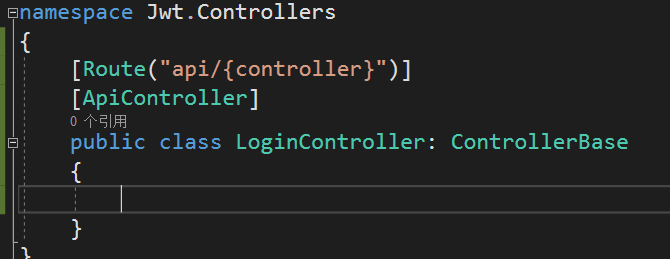
实际是Add一个Authentication服务，然后对JwtBearer进行配置。

测试一下，显示401，表示我们项目配置正确：

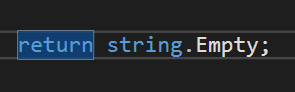


401无权限怎么解决呢，我们需要添加一个令牌去访问。先生成一个令牌：

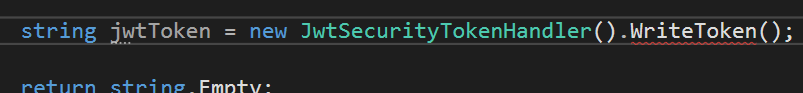
添加一个LoginController：



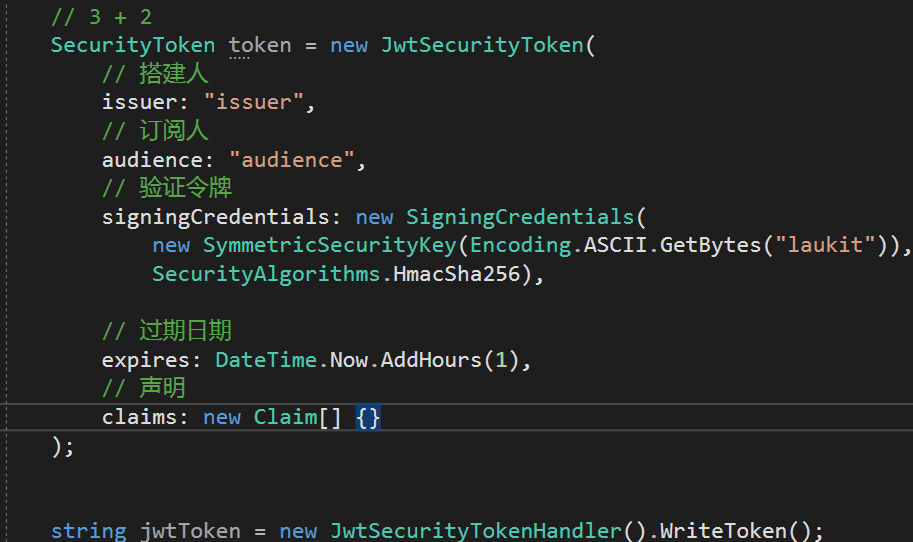
写一个GetToken方法，先倒着编程。因为方法需要返回一个Token，是string类型，我们先让它默认返回一个string.Empty：



然后定义一个string类型的Token：



查看WriteToken需要什么类型的参数，然后还是3+2：



这个3，无论在服务注册的时候还是生成令牌的时候，都是这3个，而且他们的内容要一模一样，不一样的话就验证不了。过期日期是令牌有效期，这里是1小时，声明是类似于身份证里的指纹之类的声明，可以在里面声明一些角色认证等。

最后改一下名字，填一下前面空出来的空：



也可以直接return上面的等号后的语句。

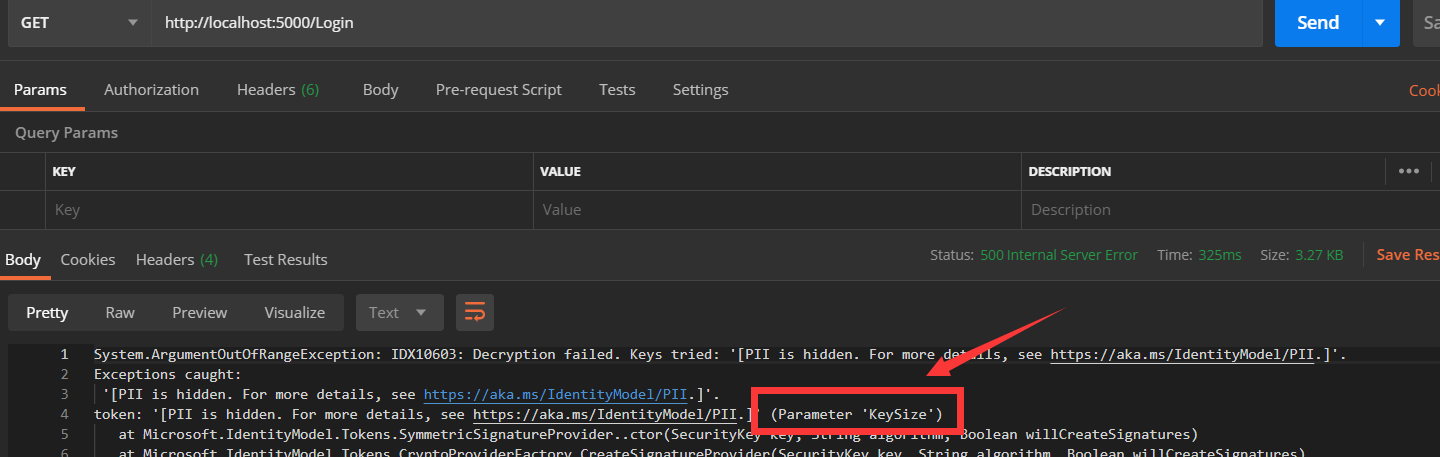
到现在为止，JWT已经完成了。它就有三块：授权（Attribute，[Authorize]）、认证方案（JwtBearer，把这个服务注入进去）、Token生成（LoginController）。

但是上面留了两个错误：

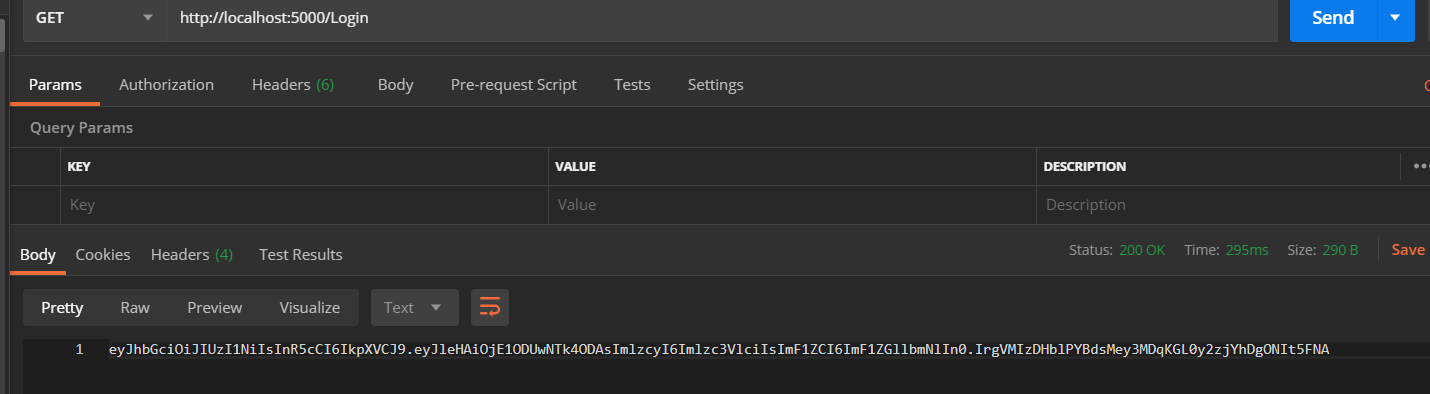
1. Kitlau作为密钥，它的长度不够，需要大于等于16位
2. 没有在中间件管道配置，验证没有开启，app.UseAuthentication ();加在UseRouting();后，app.UseAuthorization();前。

仅仅在ConfigureServices注入服务，不在中间件管道开启，服务注入和不注入没有任何区别。

先改一下Login的路由，把api删掉，我们不想把路径中带上api。当然保留也可以。

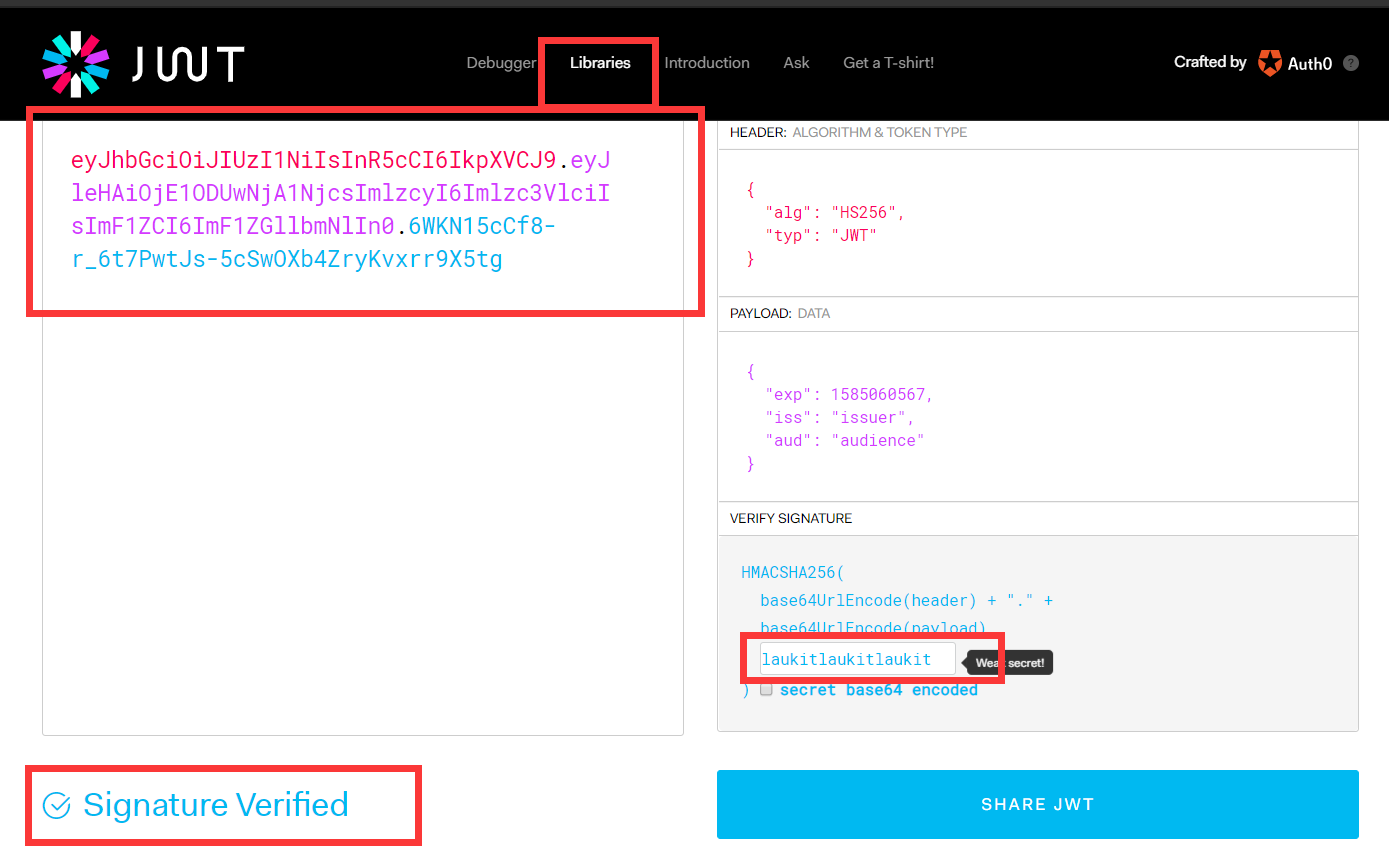


报错了，密钥长度不够。增加长度，注意保证注册服务和生成Token的密钥一致。



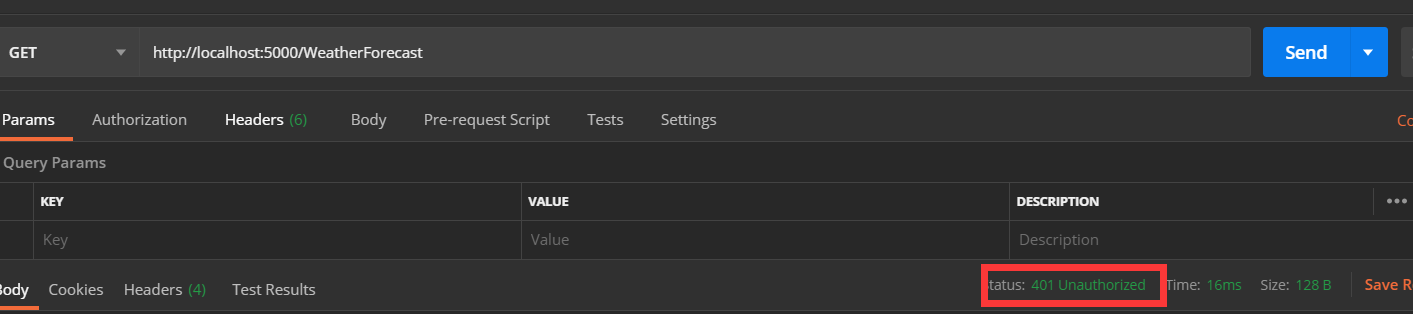
生成好了jwt这个Token。发现它很短，因为我们没有声明主体。

先去jwt的官网验证一下这个token：<https://jwt.io/> 看看里面都有什么信息。

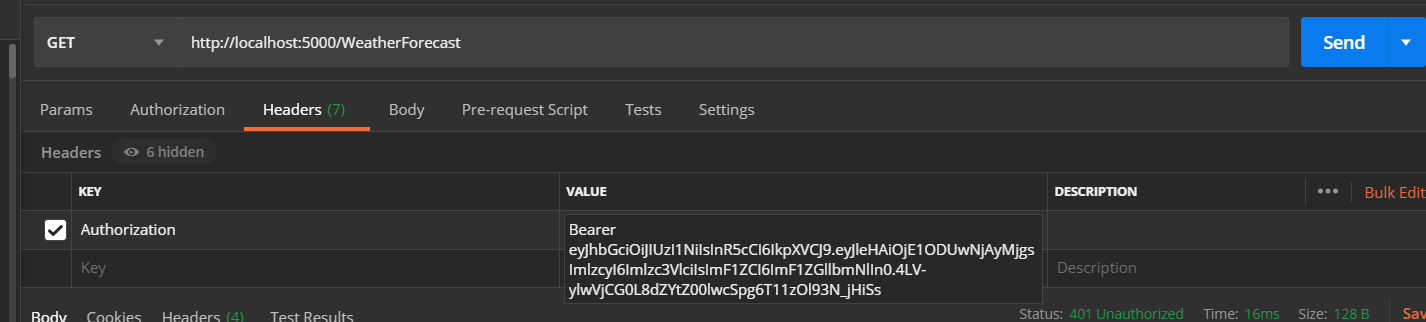


把token复制进去，签名复制到右下那里，发现左下角显示签名有效。

再访问WeatherForecast发现401：

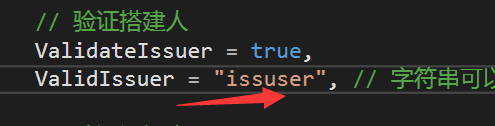


复制刚才拿到的token，再在request的header里添加：其中value是Bearer+空格+Token

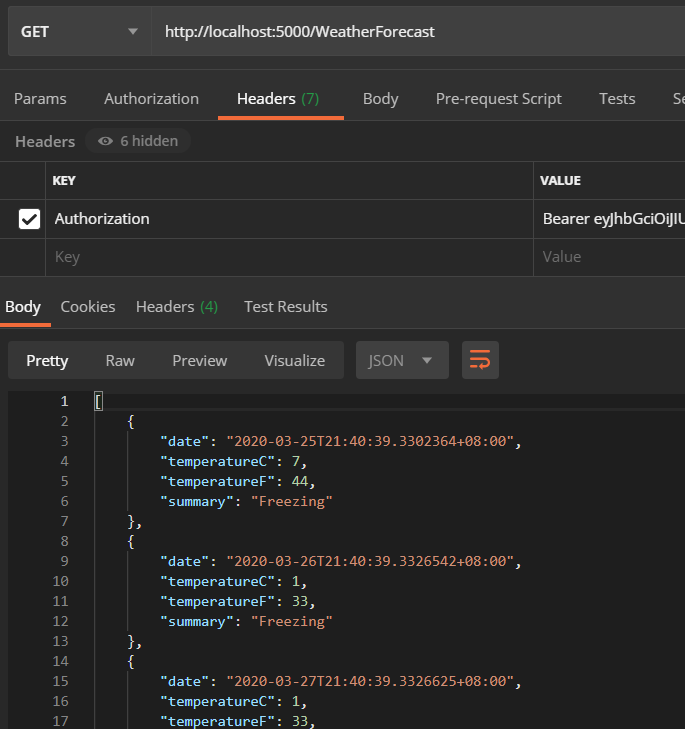


发送之后依然401，然后我们去Startup的中间件管道中设置中间件，开启认证

发现还是不行，回头找bug：



这里应该是issuer，打错了。如果是大项目，这要找到天涯海角。所以以后这种东西一定要复制：



JwtBearer认证体系要求我们的request header里的Authorization这个Key的Value必须以Bearer空格开头。

https://www.bilibili.com/video/BV1sE411y7ot?p=3

**简单的授权**

上一章全是认证，只有[Authorize]是授权，这次再来看授权。认证就是例如你进一个建筑、会馆等，入场的时候需要出示的证件就是认证。授权是：你坐到哪个位子上，你只能在哪个区域活动，你只能得到什么东西。

拿宣讲会来说，VIP用户可以参加晚宴，普通用户不行，但是普通用户便宜。

所有的授权都来自于令牌的认证。

认证就是你拿用户名和密码登录成功后，返回给你一个token，就是生成令牌的过程。

授权就是你拿着这个有效的令牌去请求的过程就是授权。

[Authorize]是最简单的授权，是无状态的。只要你的令牌是真的就行。例如地铁卡可以在任意一站进出，就是无状态的，但身份证就是有状态的。

无状态的授权如果想有状态的话，需要在声明(Claim)里赋予相应的信息。



这种就是角色是admin的才能使用的方法或接口。

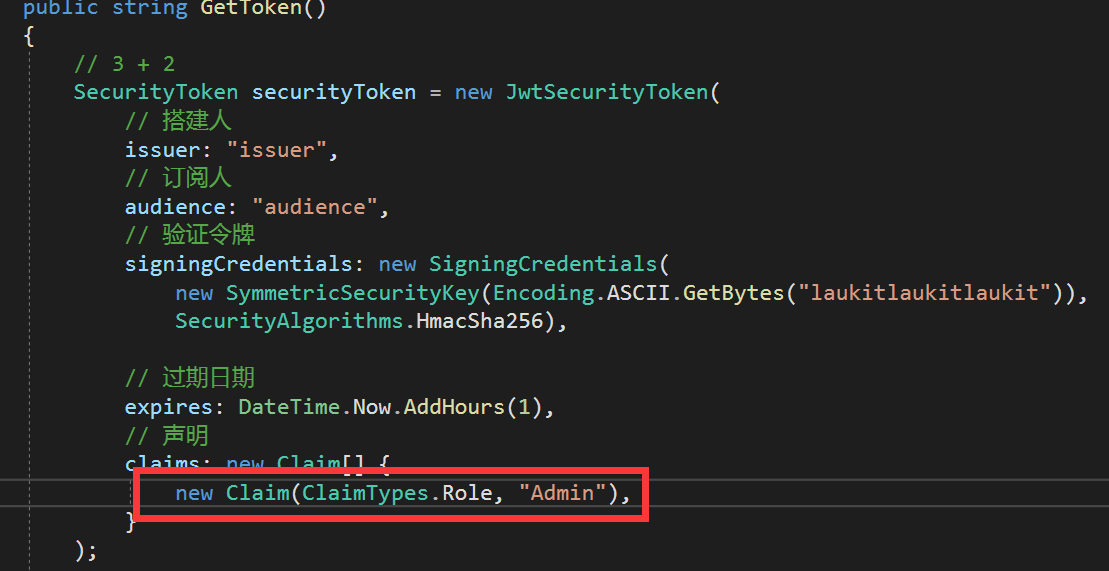


这种是策略是Admin，如果用策略，可能包含很多复杂的逻辑，比如男性，角色是xxx，身高185以上才可以访问…这种。这个时候的Admin不是单纯的一个概念了，而是它的一个规则。

无状态的情况下，Login一次，获取到token，就能用它加到header里获取到数据。它不仅是有效的，也是有权限的。我们拿着这个令牌，不但能进会场，还能到指定的位置去坐下。如果添加了角色，再拿到token发送请求就会403Forbidden，说明令牌是有效的，但没有权限。无效的话是401 Unauthorized我们拿着正确的令牌却不能访问某些有状态的接口。

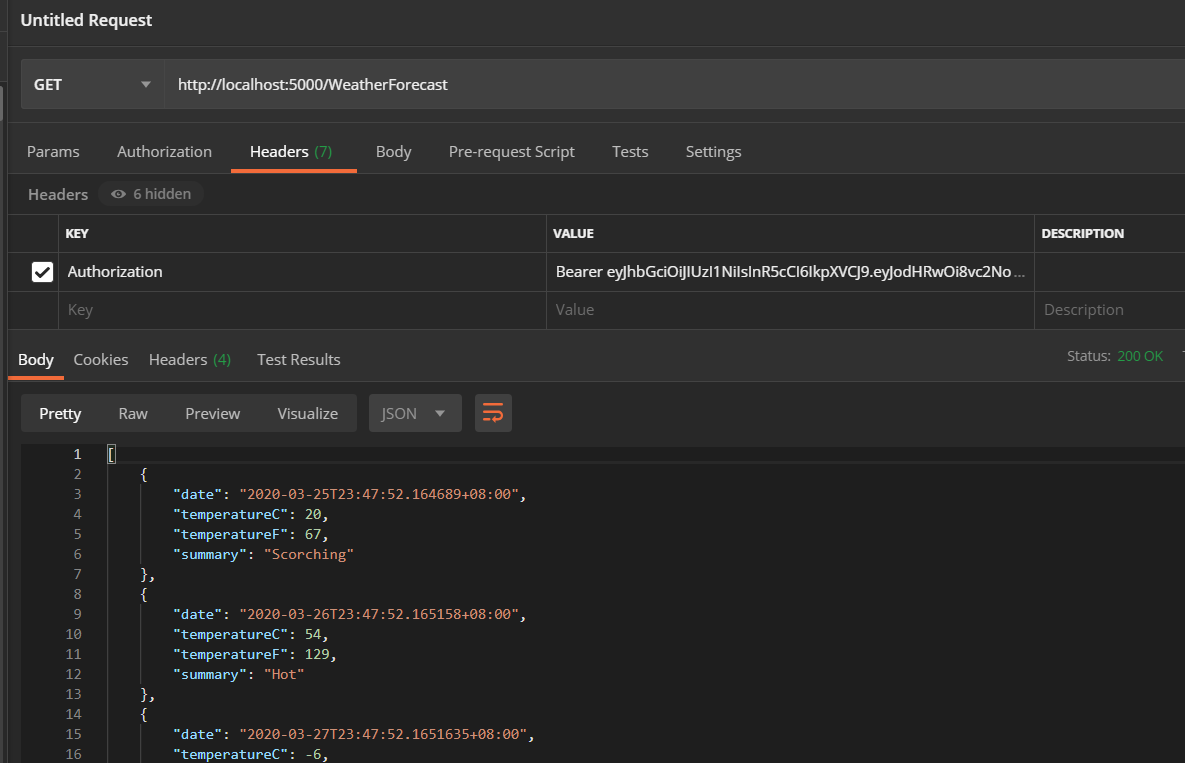


因为要求角色是Admin，我们就在生成token的LoginController的GetToken方法中的securityToken对象中的claims里添加。

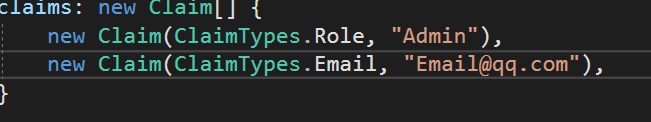


这段代码的意思就是我们在这里生成了一个有效的令牌，令牌中有一个声明，声明其一是说这个角色是Admin。

获取新令牌，发现变长了很多。然后用新令牌来请求：

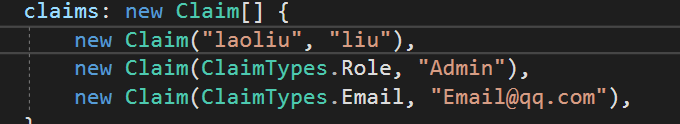


在Claim里再加一条声明：



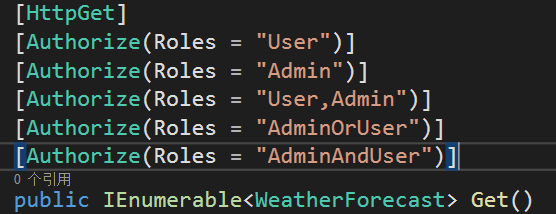
去获取令牌，发现令牌又长了，当然用新令牌也能获取到数据。

甚至还能自定义键值对，如第一行，这种形式就像session和viewbag等。：



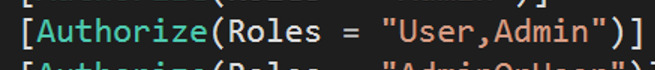
前后端分离之前，我们把一些有用的东西如session键值对都放到了内存里，而前后端分离（和分布式等）以后，这些session都不是唯一的了，凌乱了，这个时候我们就把东西全都扔到token里面让客户端去维护，客户端去提交，我们只负责验证就行了。

下一个：



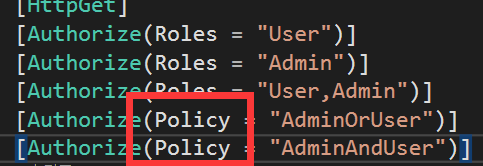
这五个规则都不一样。

第一个必须是User才能访问，第二个必须是Admin才能访问。

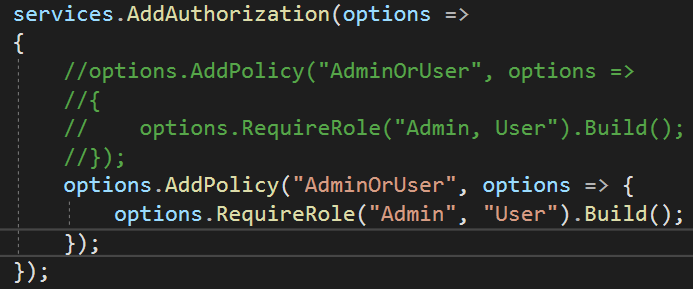


上面这条既不是或也不是且，它就单纯是一个字符串”User,Admin”。

由于上面第四条AdminOrUser等可能只是单纯的角色名字字符串，会妨碍我们理解他们之间的关系，所以我们把他们改成策略。



然后回ConfigureServices里注册策略：举一个例子

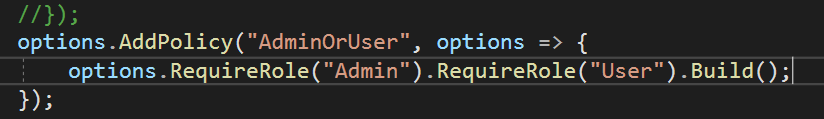


此时Login的Controller里的Claim里发放的还是Admin角色。

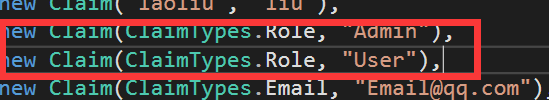
注释掉下面的策略测试上面，发现给的Token被403 Forbidden了，claim的角色改成User也一样。

但如果像上图一样注释掉上面测试下面，发现无论是Admin还是User都可以获取到数据。

把上面两个都注释掉，加上下面这个：



发现无论是Claim里是Admin还是User，都被403了。但如果两个都有：



再测试发现这次给的token能够拿到数据。所以这个策略应该改名位AdminAndUser比较规范。

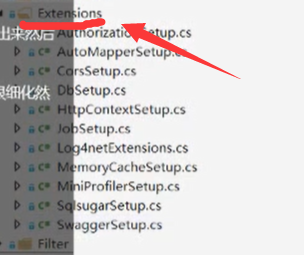
总结一下，在ConfigureServices里注册的时候，定义或的话就是RequireRole(“Admin”,”User”)，定义且的话就是RequireRole(“Admin”).RequireRole(”User”)，别忘了最后还要Build()一下，Build的作用是建立一个新的Policy。

而在发放Token的LoginController里，发放两个Role就要跟上面那张图一样写两次。

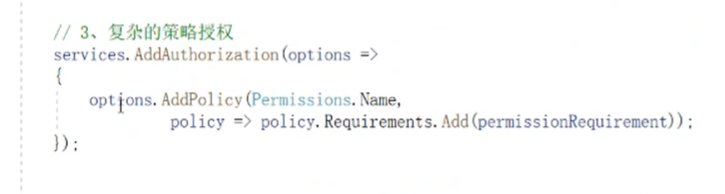
现在已经从简单的角色授权升级到定义了一个小小的策略，但如何实现特别复杂的授权？如何去动态的进行授权？

**动态的进行授权**

比较高端的操作是把ConfigureServices里的方法全部提出去，放到一个Extensions文件夹里：

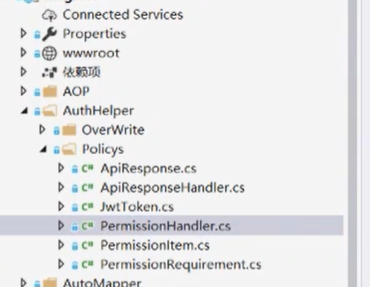


这样更好看也更方便修改。像授权这一块就可以放到AuthorizationSetup.cs里



定义了一个PermissionRequirement类的permissionRequirement对象，这个类也继承了一个接口IAuthorizationRequirement：

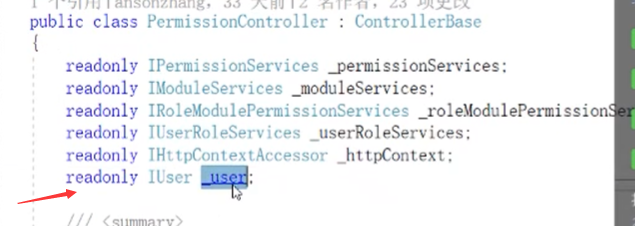


放在了这个目录下，这里还有一个处理器：PermissionHandler，可以在里面写一些业务逻辑。

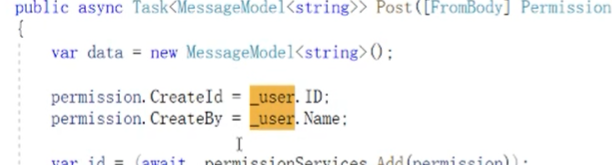
这个处理器里可以获取到HttpContext上下文，可以从上下文里获取到当前访问的URL，也可以去得到它的角色名称，拿这个角色名称去匹配URL就行了。

如何去取HttpContext上下文的内容？

如何去获取用户的信息：



封装了一个用户的接口IUser，注在PermissionController里直接去调用它的接口。像下面这样



如果访问的时候得不到用户的信息，有几个问题：

1. 中间件的顺序不对。
2. 当前这个Controller一定要带上[Authorize]这个特性：

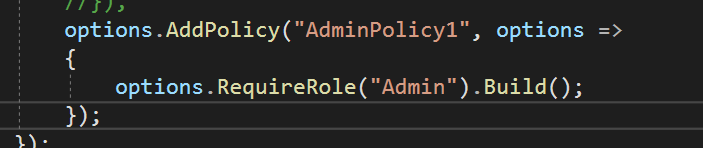


页面没加Authorize，不应该用它获取用户的信息。如果你的网站没有登录，那为什么要获取用户信息呢？而且只有加了权限才会去验证token。

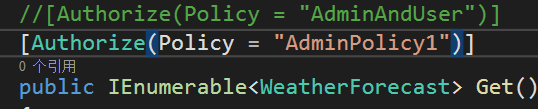
https://www.bilibili.com/video/BV1sE411y7ot?p=4

**授权\_策略详解：****基于自定义的策略授权**

先搞一个叫AdminPolicy1的策略：



然后先去我们用到的服务的Controller里添加[Authorization(Policy = “AdminPolicy1”)]这个特性：

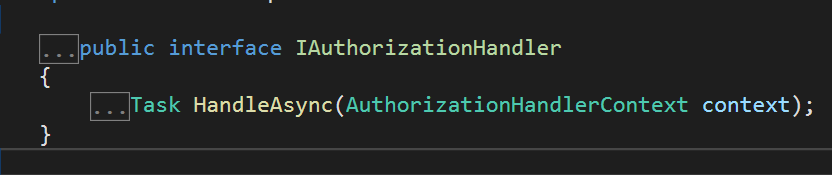


新建一个文件夹：PolicyRequirement，在里面创建一个类：MustRoleAdminHandler，意思是Role必须为Admin 的处理器。

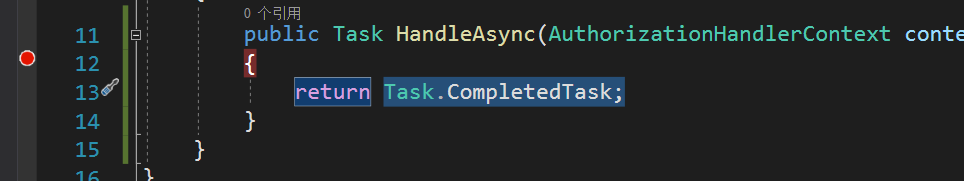
处理器是干什么的？如果简单的采用角色授权，我们无法看到它内部的原理。我们不知道它怎么判断角色是Admin，也不知道是在哪里判断的。所以我们来自己去封装一个处理器，来看它内部的原理。

新建的类需要继承一个接口：IAuthorizationHandler

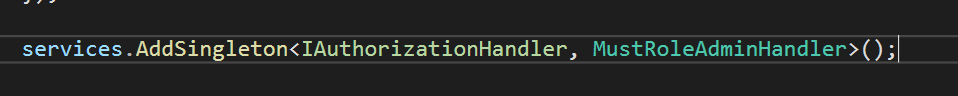
F12可以看到这个接口里有一个方法：



我们把鼠标移动到自己新建的类继承接口的程序语句中，下划红色波浪线的地方，选择修补程序中的实现接口，就会自动生成这个方法。让它先return一个Task.CompletedTask;然后在第一个大括号打上断点：



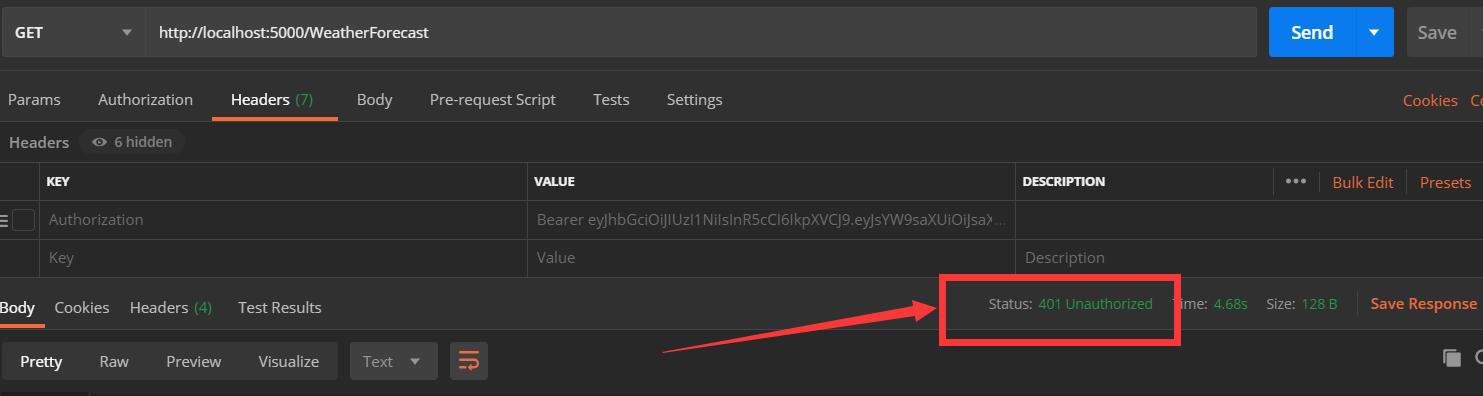
然后去ConfigureServices里把我们定义的处理器注入到Container里，使用单例的形式：



到现在为止，我们已经写完了最简单的策略授权。

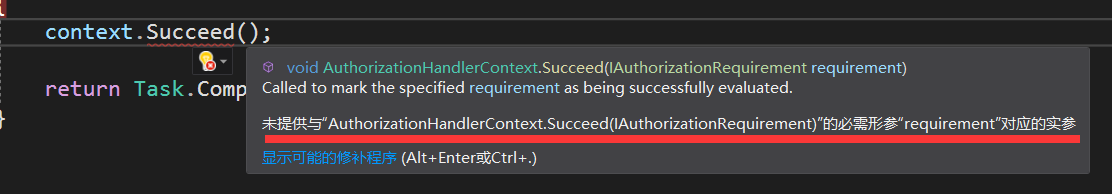
因为打了断点，我们需要调试，不能再用dotnet watch run这种省事的方法，所以我们调试启动，发现成功进入断点。

用postman试一下，也是命中断点，然后继续运行：

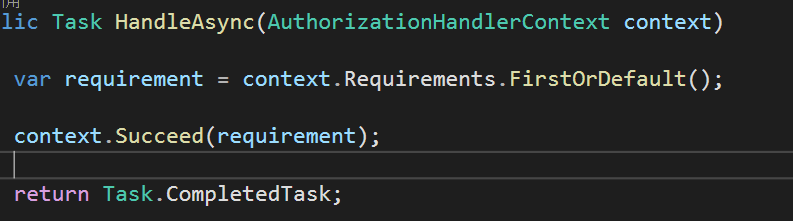


果然401 Unauthorized了。

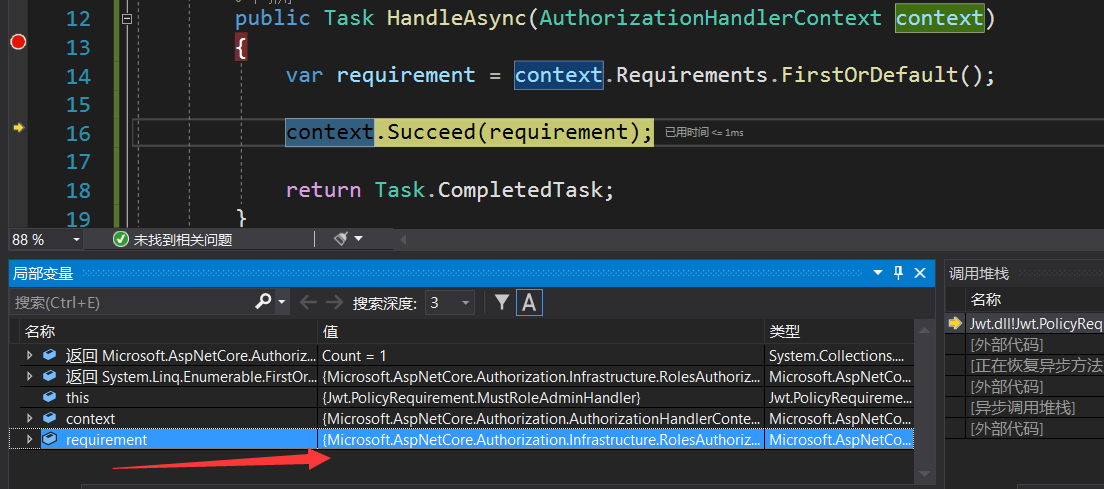
如果我们在Handler的HandleAsync中返回一个context.succeed，就能200了，先尝试写一下代码：



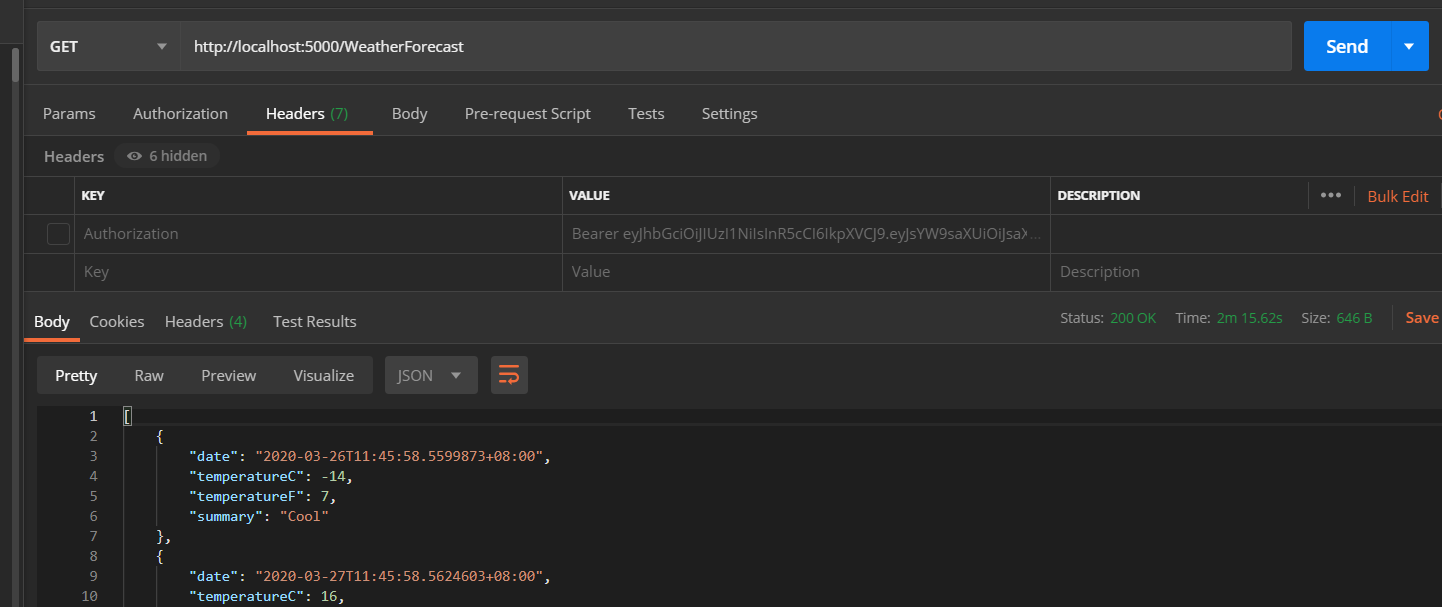
发现它需要一个requirement形参，如何拿到这个参数？我们定义一个：



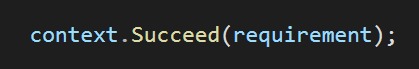
我们从context.Requirement里获取这个实例，并把requirement传给Succeed。



一步一步运行，发现requirement里面已经有一个值，然后继续运行，可以看到Postman里返回了结果。

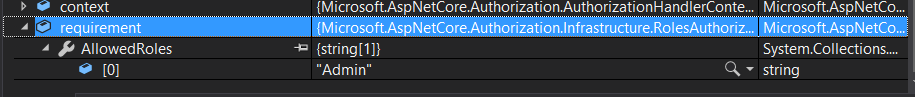


注意，此时我们关掉了Headers里的键值对，没有进行jwtBearer认证，没有Login获取token也没有给它token。目前我都能获取到它的值。原因是我们直接执行了：

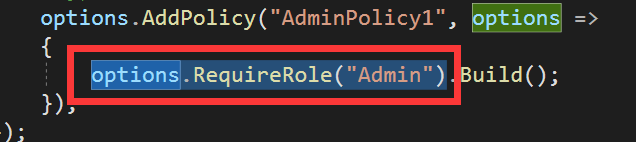


直接把请求当作成功，把结果返回出去了。这个就是策略授权。不想返回就context.Fail();这样返回的状态码是401 UnAuthorized。

而requirement的值：



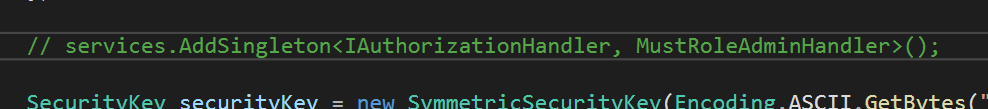
其实这个requirement就是：



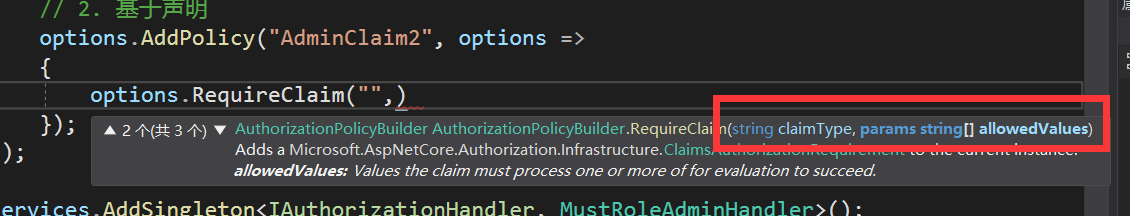
它就是当前策略名中对应的策略实体。

策略授权有三大块，1是基于角色授权，就是上面这张截图的代码。

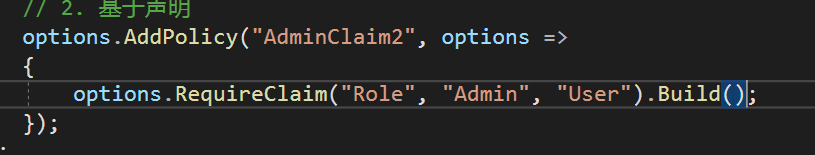
2是基于声明：注意测试前两种时需要把services里我们注入的自己创建的Handler注释掉，不然会短路掉角色授权和声明：

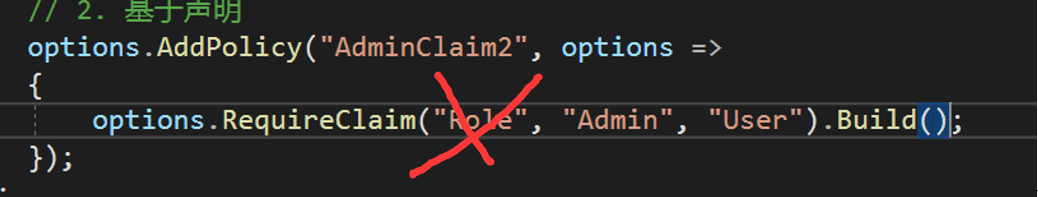


然后继续：



可以看到RequireClaim的参数全是字符串，第一个是声明的类型，后面有数目可变的参数字符串，自然是允许的声明的值。

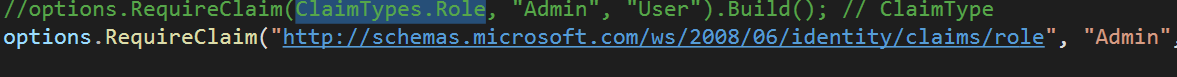




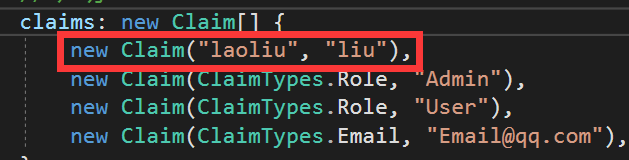
注意：上面的代码老师视频里写错了，这第一个string应该是ClaimTypes.Role，而不是“Role“，因为ClaimTypes.Role想用字符串表示的话应该是下面这一段http开头的字符串，所以我们用ClaimTypes.Role比较方便。Email等属性也是一样，但自定义属性不一样。

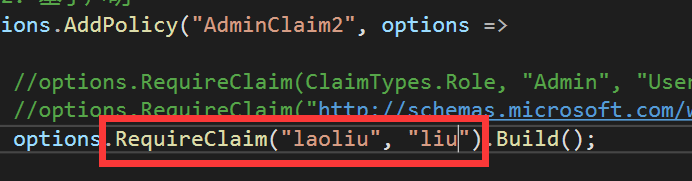


如果要强行用字符串，就是下面这样写。



然后我们尝试自定义属性，就用Claim里本来写的laoliu：





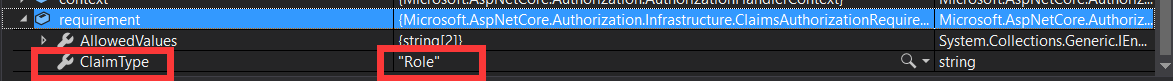
拿到token后去jwt.io分析一下token，里面有“laoliu”和“liu”这个键值对：



可以分析出自定义的在最上面，ClaimTypes里有的在中间，claim以外的属性在下面的4中。自定义的claim没有前面http等一大长串前缀。

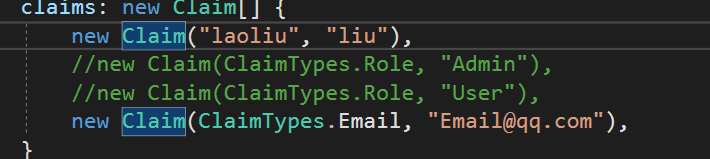
回到出错之前的图（当然是纠错之后）我们写成这样，其实名字不应该再叫AdminClaim2，因为我们允许了User的Claim，这样写不规范。这样的话只要我们的token里的Role声明是这两个（其中的一个就行），就可以访问。

测试的时候发现Claim里什么都没有竟然都可以？（问题已解决，就在上面）

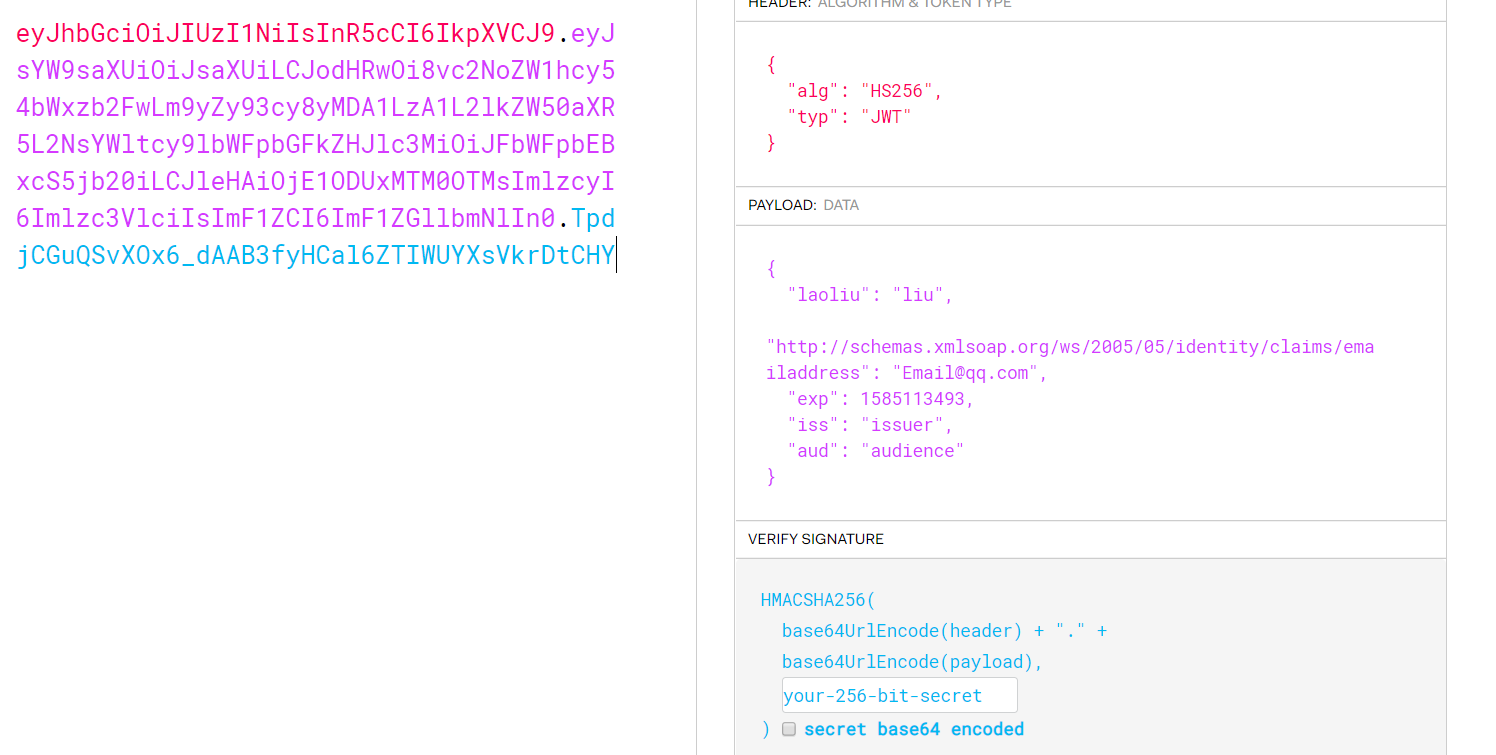


Requirement里是一个ClaimType，值是Role。上面的AllowedValues是Admin和User。

然而此时我已经把Claim给的role都注释掉了：



拿到一个token测一下：



发现没有role，只有laoliu。确确实实是没有role的Admin和User的Claim。为什么呢？

下方（问题已解决）

这个问题待解决，老师也没有讲，感觉应该查文档。

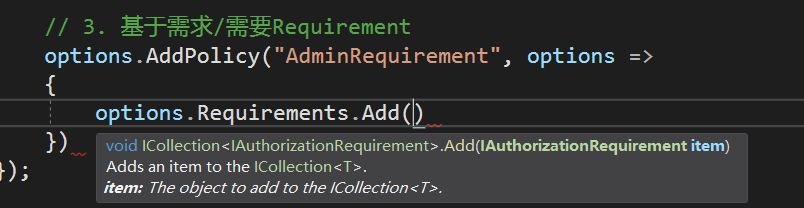
<https://docs.microsoft.com/zh-cn/dotnet/architecture/microservices/secure-net-microservices-web-applications/authorization-net-microservices-web-applications#implement-policy-based-authorization>

<https://github.com/blowdart/AspNetAuthorizationWorkshop#step-4-simple-policies>

首先，想测试官方定义的ClaimType和普通的Role，需要先注释掉下面我们自己创建的Handler的Service，测试完之后再取消注释。

经过我反复测试，发现问题是前面老师视频里的代码写错了，已经在上面改正，可以上去查阅。

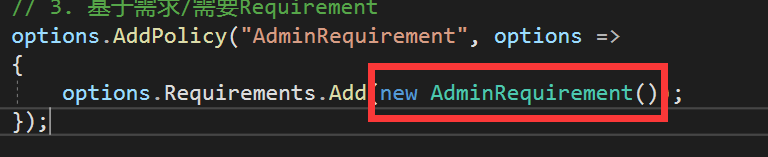
3是基于需求/需要Requirement，



基于角色授权是基于Role，基于Claim是基于ClaimType，而我们实际开发的时候可能不仅仅只用这两个，我们可能自己需要一些参数，这个时候我们就需要去自己定义一个实体，想从里面获取内容。从上图中可以看出它的参数是一个继承了IAuthorizationRequirement的实例。

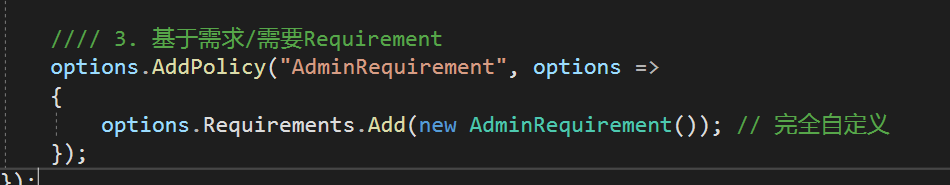
所以我们现在需要建一个类来做出这个实例。我们在PolicyRequirement文件夹下建一个叫AdminRequirement.cs的类，它需要实现IAuthorizationRequirement这个接口。

然后我们去services里给它的参数new一个这个实例：



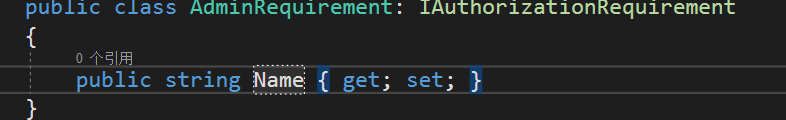
实际上基于角色授权和基于Claim这第1、2种，平常基本不用，会这两种只是为第三种做铺垫。

第三种：

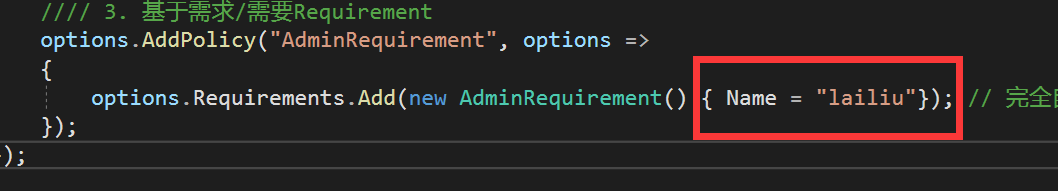


完全自定义。应该是在我们新建的AdminRequirement.cs中自定义。

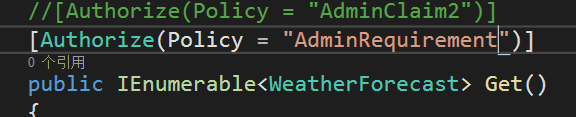
为了演示效果，在AdminRequirement.cs里新建一个Name字段：



然后在configureServices里给它赋值：



然后去服务的controller中把这个策略用上：

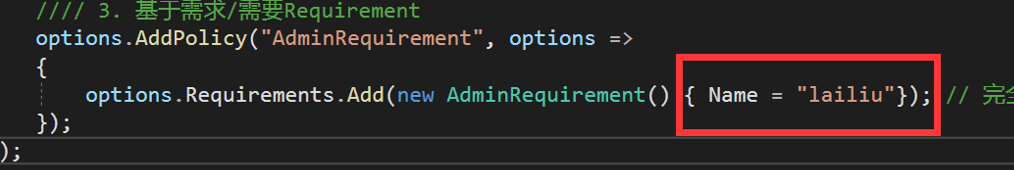


加断点运行调试一下：



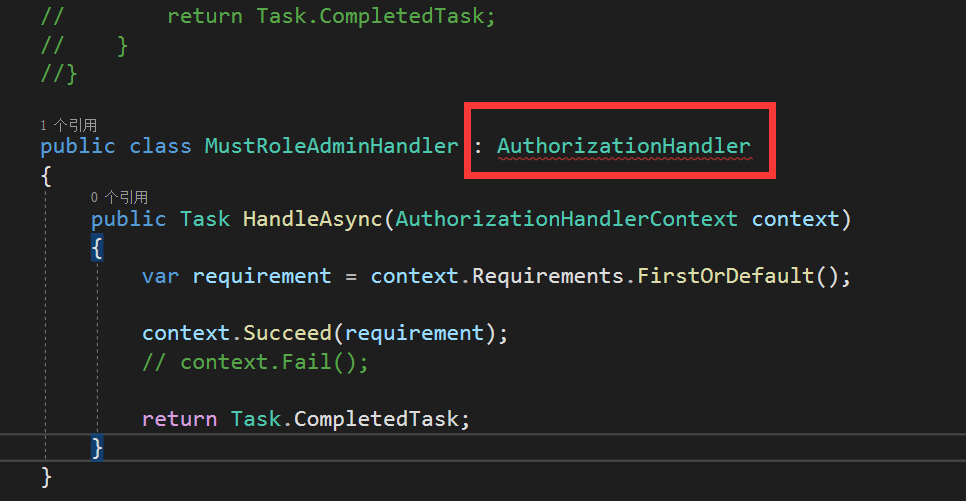
发现requirement是一个我们自定义的AdminRequirement类的实例，它的Name字段的值是laoliu。

总结一下：我们可以在下面这里：

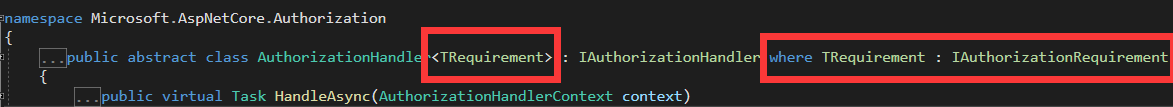


把好几个值传给Handler处理器，处理器里肯定需要用到这个requirement。

后来官方发现我们实现IAuthorizationHandler这样写起来很不优雅，就把它封装到了一个抽象类里，就是接口名去掉I就行了，我们试一下：

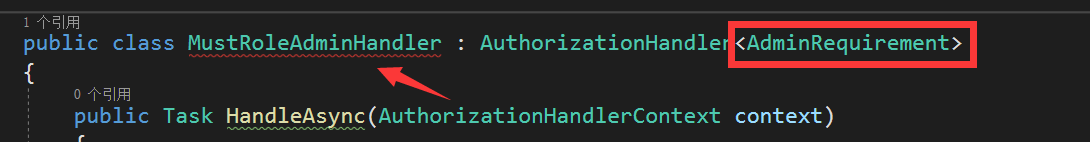


F12进去可以发现这个抽象类实现了前面的IAuthorizationHandler接口。封装这个抽象类的目的除了更好看更优雅之外，主要是提供了封装性。

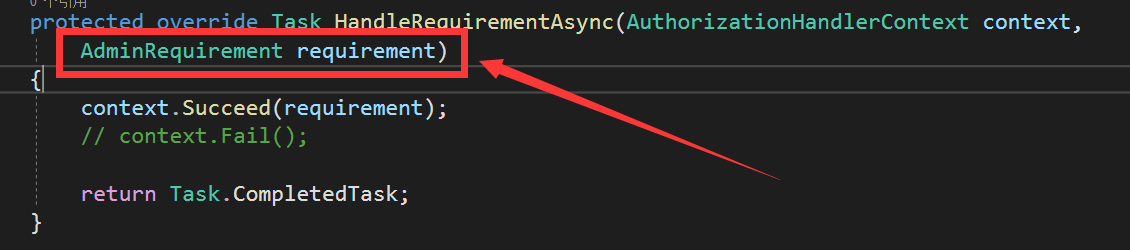


至于报错的原因是它需要加一个泛型，泛型定义中的 where 子句指定对用作泛型类型、方法、委托或本地函数中类型参数的参数类型的约束。 约束可指定接口、基类或要求泛型类型为引用、值或非托管类型。 它们声明类型参数必须具备的功能。

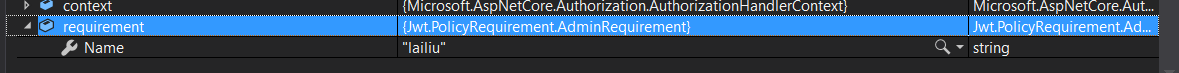
所以这个泛型TRequirement必须实现IAuthorizationRequirement接口。在这里也就是我们自定义的AdminRequirement：



依然报错，说明我们必须实现这个抽象类，即实现这个类定义的抽象方法，用快速修补程序自动生成，然后把上面注释掉的方法体内的代码复制过来：



因为参数中已经传入requirement，我们不需要再从上下文中获取，就删掉第一条获取用的代码。打上断点调试一下：



成功获取到了自定义的类的实例和赋给它的值。

三种授权可以一起生效，似乎是只要一种授权通过了，即使不满足另外两种授权，也能取到值，只要满足一个就能通过。