
IdentityServer4 Documentation

发布 *1.0.0*

Brock Allen, Dominick Baier

2021 年 09 月 06 日

| | | |
|----------|--|-----------|
| 1 | 大图 | 3 |
| 1.1 | 验证 | 4 |
| 1.2 | API 访问 | 5 |
| 1.3 | OpenID Connect 和 OAuth 2.0 –更好地结合在一起 | 5 |
| 1.4 | IdentityServer4 如何提供帮助 | 5 |
| 2 | 术语 | 7 |
| 2.1 | IdentityServer | 8 |
| 2.2 | User | 8 |
| 2.3 | Client | 8 |
| 2.4 | Resources | 8 |
| 2.5 | Identity Token | 9 |
| 2.6 | Access Token | 9 |
| 3 | 支持的规范 | 11 |
| 3.1 | OpenID Connect | 11 |
| 3.2 | OAuth 2.0 | 11 |
| 4 | 打包和构建 | 13 |
| 4.1 | IdentityServer4 主仓库 | 13 |
| 4.2 | 快速入门用户界面 | 13 |
| 4.3 | 访问令牌验证处理程序 | 14 |
| 4.4 | 模板 | 14 |
| 4.5 | 开发版本 | 14 |
| 5 | 支持和咨询选项 | 15 |
| 5.1 | 免费支持 | 15 |
| 5.2 | 商业支持 | 16 |

| | |
|-------------------------------------|-----------|
| 6 演示服务器 | 17 |
| 7 贡献 | 19 |
| 7.1 如何贡献? | 19 |
| 7.2 一般反馈和讨论? | 20 |
| 7.3 Bugs 和功能请求? | 20 |
| 7.4 贡献代码和内容 | 20 |
| 7.5 贡献项目 | 20 |
| 8 概述 | 21 |
| 8.1 准备工作 | 21 |
| 9 使用客户端凭据保护 API | 23 |
| 9.1 源代码 | 23 |
| 9.2 准备工作 | 23 |
| 9.3 设置 ASP.NET Core 应用程序 | 24 |
| 9.4 定义 API Scope | 24 |
| 9.5 定义客户端 | 25 |
| 9.6 配置 IdentityServer | 26 |
| 9.7 添加 API | 28 |
| 9.7.1 控制器 | 28 |
| 9.7.2 添加 Nuget 依赖项 | 29 |
| 9.7.3 配置 | 29 |
| 9.8 创建客户端 | 30 |
| 9.9 调用 API | 32 |
| 9.10 API 授权 | 33 |
| 9.11 进一步的实验 | 34 |
| 10 使用 ASP.NET Core 的交互式应用程序 | 35 |
| 10.1 添加用户界面 | 35 |
| 10.2 创建 MVC 客户端 | 36 |
| 10.3 添加对 OpenID Connect 身份范围的支持 | 38 |
| 10.4 添加测试用户 | 39 |
| 10.5 将 MVC 客户端添加到 IdentityServer 配置 | 39 |
| 10.6 测试客户端 | 40 |
| 10.7 添加注销 | 43 |
| 10.8 从 UserInfo 端点获取声明 | 43 |
| 10.9 进一步的实验 | 44 |
| 10.10 添加对外部身份验证的支持 | 45 |
| 10.11 添加 Google 支持 | 45 |
| 10.12 进一步的实验 | 47 |
| 11 ASP.NET Core 和 API 访问 | 49 |

| | | |
|-----------|--|-----------|
| 11.1 | 修改客户端配置 | 49 |
| 11.2 | 修改 MVC 客户端 | 50 |
| 11.3 | 使用访问令牌 | 51 |
| 11.4 | 管理访问令牌 | 51 |
| 12 | 添加 JavaScript 客户端 | 53 |
| 12.1 | 新的 JavaScript 客户端项目 | 53 |
| 12.2 | 修改托管 | 54 |
| 12.3 | 添加静态文件中间件 | 54 |
| 12.4 | 引用 oidc-client | 54 |
| 12.5 | 添加您的 HTML 和 JavaScript 文件 | 55 |
| 12.6 | 为 JavaScript 客户端向 IdentityServer 添加客户端注册 | 58 |
| 12.7 | 使用 CORS 允许 Ajax 调用 Web API | 58 |
| 12.8 | 运行 JavaScript 应用程序 | 59 |
| 13 | 使用 EntityFramework Core 获取配置和操作数据 | 65 |
| 13.1 | IdentityServer4.EntityFramework | 65 |
| 13.2 | 使用 SqlServer | 66 |
| 13.3 | 数据库架构更改和使用 EF 迁移 | 66 |
| 13.4 | 配置存储 | 66 |
| 13.5 | 添加迁移 | 67 |
| 13.6 | 初始化数据库 | 68 |
| 13.7 | 运行客户端应用程序 | 70 |
| 14 | 使用 ASP.NET Core Identity | 71 |
| 14.1 | 新的 ASP.NET Core Identity 项目 | 71 |
| 14.2 | 检查新项目 | 72 |
| 14.2.1 | IdentityServerAspNetIdentity.csproj | 72 |
| 14.2.2 | Startup.cs | 72 |
| 14.2.3 | Config.cs | 72 |
| 14.2.4 | Program.cs 和 SeedData.cs | 74 |
| 14.2.5 | AccountController | 74 |
| 14.3 | 使用 MVC 客户端登录 | 74 |
| 14.4 | 缺少了什么? | 78 |
| 15 | Startup | 79 |
| 15.1 | 配置服务 | 79 |
| 15.2 | 密钥材料 | 79 |
| 15.3 | 内存配置存储 | 80 |
| 15.4 | 测试存储 | 80 |
| 15.5 | 额外服务 | 80 |
| 15.6 | 缓存 | 81 |
| 15.7 | 配置管道 | 82 |

| | |
|---|------------|
| 16 定义资源 | 83 |
| 16.1 身份资源 | 83 |
| 16.2 APIs | 85 |
| 16.2.1 Scopes | 85 |
| 16.2.2 基于 Scopes 的授权 | 86 |
| 16.2.3 参数化 Scopes | 87 |
| 16.2.4 API Resources | 88 |
| 16.2.5 迁移到 v4 的步骤 | 90 |
| 17 定义客户端 | 91 |
| 17.1 定义服务器到服务器通信的客户端 | 92 |
| 17.2 定义用于使用身份验证和委托 API 访问的交互式应用程序 | 92 |
| 17.3 在 appsettings.json 中定义客户端 | 93 |
| 18 登录 | 95 |
| 18.1 Cookie 认证 | 95 |
| 18.2 覆盖 cookie 处理程序配置 | 96 |
| 18.3 登录用户界面和身份管理系统 | 96 |
| 18.4 登录工作流程 | 96 |
| 18.5 登录上下文 | 97 |
| 18.6 颁发 cookie 和声明 | 97 |
| 19 使用外部身份提供商登录 | 99 |
| 19.1 为外部提供商添加身份验证处理程序 | 99 |
| 19.2 Cookie 的作用 | 99 |
| 19.3 触发身份验证处理程序 | 101 |
| 19.4 处理回调并登录用户 | 101 |
| 19.5 状态、URL 长度和 ISecureDataFormat | 103 |
| 20 Windows 身份验证 | 105 |
| 20.1 在使用 IIS 托管的 Windows 上 | 105 |
| 21 注销 | 109 |
| 21.1 删除身份验证 cookie | 109 |
| 21.2 通知客户端用户已注销 | 110 |
| 21.3 由客户端应用程序发起的注销 | 110 |
| 22 注销外部身份提供商 | 111 |
| 23 联合注销 | 113 |
| 24 联合网关 | 115 |
| 24.1 执行 | 116 |
| 25 同意 | 117 |

| | | |
|-----------|------------------------|------------|
| 25.1 | 同意页面 | 117 |
| 25.2 | 授权上下文 | 117 |
| 25.3 | 将同意结果通知 IdentityServer | 118 |
| 25.4 | 将用户返回到授权端点 | 118 |
| 26 | 保护 API | 119 |
| 26.1 | 验证引用令牌 | 120 |
| 26.2 | 支持 JWT 和引用令牌 | 120 |
| 27 | 部署 | 123 |
| 27.1 | 典型架构 | 123 |
| 27.2 | 配置数据 | 124 |
| 27.3 | 密钥材料 | 125 |
| 27.4 | 操作数据 | 125 |
| 27.5 | ASP.NET Core 数据保护 | 125 |
| 27.6 | ASP.NET Core 分布式缓存 | 125 |
| 28 | 日志 | 127 |
| 28.1 | Serilog 的设置 | 127 |
| 29 | 事件 | 129 |
| 29.1 | 发出事件 | 129 |
| 29.2 | 自定义 sinks | 130 |
| 29.3 | 内置事件 | 131 |
| 29.4 | 自定义事件 | 132 |
| 30 | 密码学、密钥和 HTTPS | 133 |
| 30.1 | 令牌签名和验证 | 133 |
| 30.2 | 签名密钥滚动更新 | 134 |
| 30.3 | 数据保护 | 134 |
| 30.4 | HTTPS | 134 |
| 31 | 授权类型 | 135 |
| 31.1 | 机器到机器通信 | 136 |
| 31.2 | 交互式客户端 | 136 |
| 31.3 | 没有浏览器或输入设备受限的交互式客户端 | 137 |
| 31.4 | 自定义场景 | 137 |
| 32 | 客户端认证 | 139 |
| 32.1 | 创建共享密钥 | 139 |
| 32.2 | 使用共享密钥进行身份验证 | 140 |
| 32.3 | 使用非对称密钥进行身份验证 | 141 |
| 33 | 扩展授权 | 143 |

| | |
|--|------------|
| 33.1 示例：使用扩展授权的简单委托 | 144 |
| 34 资源所有者密码验证 | 149 |
| 35 刷新令牌 | 151 |
| 35.1 其他客户端设置 | 151 |
| 35.2 请求刷新令牌 | 152 |
| 35.3 使用刷新令牌请求访问令牌 | 152 |
| 35.4 自定义刷新令牌行为 | 152 |
| 36 引用令牌 | 155 |
| 37 持久授权 | 157 |
| 37.1 持久授权 | 157 |
| 37.2 授权消费 | 158 |
| 37.3 持久授权服务 | 158 |
| 38 访问令牌所有权证明 | 159 |
| 39 Mutual TLS | 161 |
| 39.1 服务器设置 | 161 |
| 39.2 ASP.NET Core 设置 | 162 |
| 39.3 IdentityServer setup | 163 |
| 39.4 客户端认证 | 164 |
| 39.4.1 使用客户端证书对 IdentityServer 进行身份验证 | 165 |
| 39.5 发送方受限访问令牌 | 166 |
| 39.5.1 确认声明 | 166 |
| 39.5.2 在 API 中验证和接受客户端证书 | 167 |
| 39.5.3 自省和确认声明 | 170 |
| 39.6 临时客户端证书 | 170 |
| 39.6.1 使用临时证书请求令牌 | 170 |
| 40 授权请求对象 | 173 |
| 40.1 通过引用传递请求 JWTs | 174 |
| 40.2 访问请求对象数据 | 175 |
| 41 自定义令牌请求验证和颁发 | 177 |
| 42 CORS | 179 |
| 42.1 基于客户端的 CORS 配置 | 179 |
| 42.2 自定义 Cors 策略服务 | 180 |
| 42.3 将 IdentityServer CORS 策略与 ASP.NET Cores CORS 策略混合使用 | 180 |
| 43 发现 | 181 |
| 43.1 扩展发现 | 181 |

| | |
|----------------------------------|------------|
| 44 添加更多 API 端点 | 183 |
| 44.1 发现 | 184 |
| 44.2 高级 | 184 |
| 44.3 声明转换 | 185 |
| 45 添加新协议 | 187 |
| 45.1 典型的身份验证工作流程 | 187 |
| 45.2 有用的 IdentityServer 服务 | 188 |
| 46 工具 | 191 |
| 47 发现端点 | 193 |
| 48 授权端点 | 195 |
| 49 令牌端点 | 197 |
| 49.1 例子 | 198 |
| 50 用户信息端点 | 199 |
| 50.1 例子 | 199 |
| 51 设备授权端点 | 201 |
| 51.1 例子 | 201 |
| 52 自省端点 | 203 |
| 52.1 例子 | 203 |
| 53 撤销端点 | 205 |
| 53.1 例子 | 205 |
| 54 结束会话端点 | 207 |
| 54.1 参数 | 207 |
| 54.2 例子 | 208 |
| 55 IdentityServer Options | 209 |
| 55.1 Endpoints | 209 |
| 55.2 Discovery | 210 |
| 55.3 Authentication | 210 |
| 55.4 Events | 210 |
| 55.5 InputLengthRestrictions | 210 |
| 55.6 UserInteraction | 211 |
| 55.7 Caching | 211 |
| 55.8 CORS | 211 |
| 55.9 CSP (内容安全政策) | 212 |
| 55.10 Device Flow | 212 |
| 55.11 Mutual TLS | 212 |

| | |
|---|------------|
| 56 Identity Resource | 213 |
| 57 API Scope | 215 |
| 57.1 Defining API scope in appsettings.json | 215 |
| 58 API Resource | 217 |
| 58.1 Defining API resources in appsettings.json | 218 |
| 59 Client | 219 |
| 59.1 Basics | 219 |
| 59.2 Authentication/Logout | 220 |
| 59.3 Token | 220 |
| 59.4 Consent Screen | 221 |
| 59.5 Device flow | 222 |
| 60 GrantValidationResult | 223 |
| 61 Profile Service | 225 |
| 61.1 IProfileService APIs | 225 |
| 61.2 ProfileDataRequestContext | 225 |
| 61.3 请求的范围和声明映射 | 226 |
| 61.4 IsActiveContext | 226 |
| 62 IdentityServer Interaction Service | 227 |
| 62.1 IIdentityServerInteractionService APIs | 227 |
| 62.2 AuthorizationRequest | 228 |
| 62.3 ResourceValidationResult | 228 |
| 62.4 ErrorMessage | 229 |
| 62.5 LogoutRequest | 229 |
| 62.6 ConsentResponse | 229 |
| 62.7 Grant | 229 |
| 63 Device Flow Interaction Service | 231 |
| 63.1 IDeviceFlowInteractionService APIs | 231 |
| 63.2 DeviceFlowAuthorizationRequest | 231 |
| 63.3 DeviceFlowInteractionResult | 232 |
| 64 Entity Framework Support | 233 |
| 64.1 Configuration Store support for Clients, Resources, and CORS settings | 233 |
| 64.2 ConfigurationStoreOptions | 234 |
| 64.3 Operational Store support for persisted grants | 235 |
| 64.4 OperationalStoreOptions | 235 |
| 64.5 Database creation and schema changes across different versions of IdentityServer | 236 |
| 65 ASP.NET Identity Support | 237 |

| | |
|---|------------|
| 66 Training | 239 |
| 66.1 Identity & Access Control for modern Applications (using ASP.NET Core 2 and IdentityServer4) . . . | 239 |
| 66.2 PluralSight courses | 239 |
| 67 Blog posts | 241 |
| 67.1 Team posts | 241 |
| 67.1.1 2020 | 241 |
| 67.1.2 2019 | 242 |
| 67.1.3 2018 | 242 |
| 67.1.4 2017 | 242 |
| 67.2 Community posts | 242 |
| 68 Videos | 245 |
| 68.1 2020 | 245 |
| 68.2 2019 | 245 |
| 68.3 2018 | 245 |
| 68.4 2017 | 246 |
| 68.5 2016 | 246 |
| 68.6 2015 | 246 |
| 68.7 2014 | 246 |



IdentityServer4 是用于 ASP.NET Core 的 OpenID Connect 和 OAuth 2.0 框架。

警告： 从 2020 年 10 月 1 日起，我们成立了一家新的 公司 。所有新的主要功能工作都将在我们新的 组织 中进行。新的 Duende IdentityServer 可在 FOSS (RPL) 和商业许可下使用。开发和测试始终免费。联系 了解更多信息。

IdentityServer4 将在 2022 年 11 月之前进行错误修复和安全更新。

注解： 本文档涵盖了主分支上的最新版本。这可能尚未发布。使用左下角的版本选择器选择特定版本的文档。

它在您的应用程序中启用以下功能：

身份验证即服务

所有应用程序（Web、原生、移动、服务）的集中登录逻辑和工作流。

IdentityServer 是 OpenID Connect 的官方 认证 实现。

单点登录/注销

通过多种应用程序类型进行单点登录（和注销）。

API 访问控制

为各种类型的客户端发布 API 的访问令牌，例如服务器到服务器、Web 应用程序、SPA 和原生/移动应用程序。

联合网关

支持外部身份提供商，如 Azure Active Directory、Google、Facebook 等。

这使您的应用程序免受如何连接到这些外部提供商的详细信息的影响。

专注于定制

最重要的部分 - IdentityServer 的许多方面都可以定制以满足 **您** 的需求。由于 IdentityServer 是一个框架，而不是盒装产品或 SaaS，您可以编写代码来调整系统，使其适合您的场景。

成熟的开源

IdentityServer 使用许可的 [Apache 2](#) 许可证，允许在其上构建商业产品。

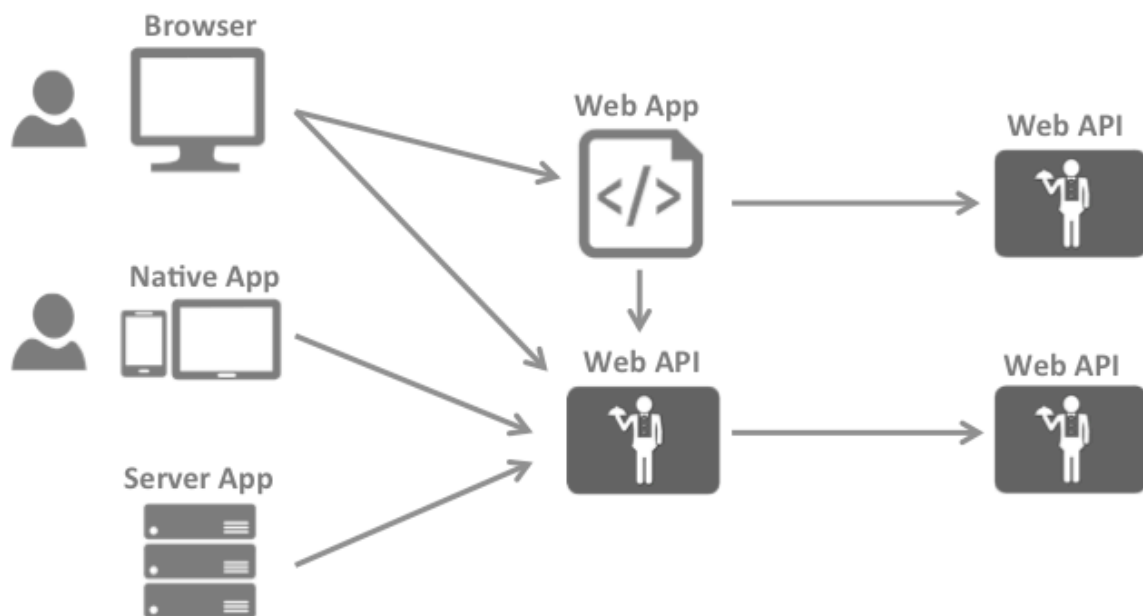
它也是 [.NET 基金会](#) 的一部分，提供治理和法律支持。

免费和商业支持

如果您需要帮助构建或运行您的身份平台，请告诉我们。我们可以通过多种方式帮助您。

大图

大多数现代应用程序看起来或多或少是这样的：



最常见的交互是：

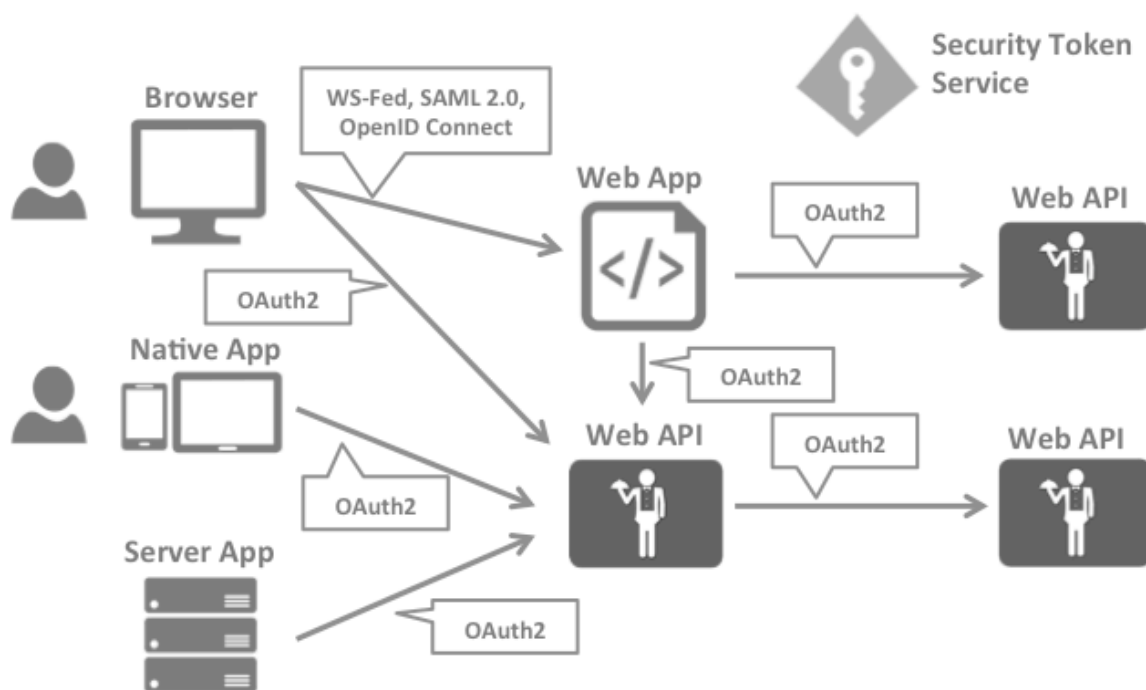
- 浏览器与 Web 应用程序通信
- Web 应用程序与 Web API 通信（有时是自己通信，有时是代表用户通信）

- 基于浏览器的应用程序与 Web API 通信
- 原生应用程序与 Web API 通信
- 基于服务器的应用程序与 Web API 通信
- Web API 与 Web API 通信（有时是自己通信，有时是代表用户通信）

通常，每一层（前端、中间层和后端）都必须保护资源并实施身份验证和/或授权——通常针对同一个用户存储。

将这些基本的安全功能外包给安全令牌服务可以防止在这些应用程序和端点之间复制该功能。

重构应用程序以支持安全令牌服务会导致以下架构和协议：



这样的设计将安全问题分为两部分：

1.1 验证

当应用程序需要知道当前用户的身份时，就需要进行身份验证。通常，这些应用程序代表该用户管理数据，并且需要确保该用户只能访问他被允许访问的数据。最常见的例子是（典型的）Web 应用程序—但是原生和基于 JS 的应用程序也需要身份验证。

最常见的身份验证协议是 SAML2p、WS-Federation 和 OpenID Connect ——SAML2p 是最流行和最广泛部署的。

OpenID Connect 是三者中最新的，但被认为是未来，因为它在现代应用程序中最有潜力。它从一开始就是为移动应用场景而构建的，旨在对 API 友好。

1.2 API 访问

应用程序有两种与 API 通信的基本方式—使用应用程序标识，或委托用户的身份。有时需要将这两种方法结合起来。

OAuth2 是一种允许应用程序从安全令牌服务请求访问令牌并使用它们与 API 通信的协议。这种委托降低了客户端应用程序和 API 的复杂性，因为身份验证和授权可以集中。

1.3 OpenID Connect 和 OAuth 2.0 –更好地结合在一起

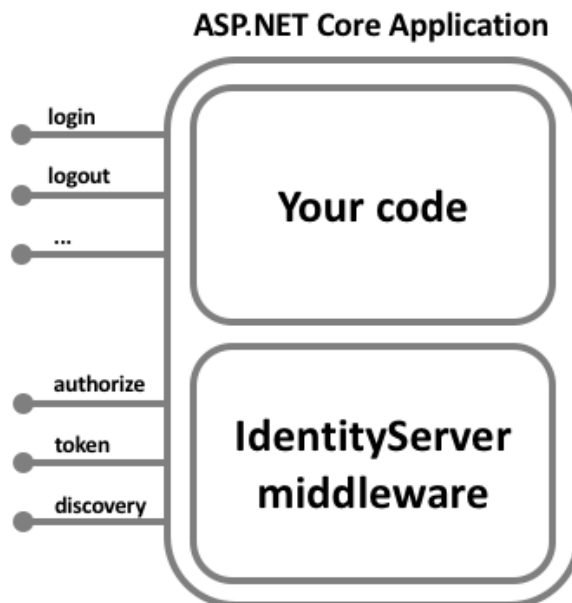
OpenID Connect 和 OAuth 2.0 非常相似—事实上，OpenID Connect 是 OAuth 2.0 之上的扩展。两个基本的安全问题，身份验证和 API 访问，被组合成一个单一的协议 - 通常与安全令牌服务的单次往返。

我们相信 OpenID Connect 和 OAuth 2.0 的结合是在可预见的未来保护现代应用程序的最佳方法。IdentityServer4 是这两种协议的一个实现，并且经过高度优化，可以解决当今移动、原生和 Web 应用程序的典型安全问题。

1.4 IdentityServer4 如何提供帮助

IdentityServer 是将符合规范的 OpenID Connect 和 OAuth 2.0 端点添加到任意 ASP.NET Core 应用程序的中间件。

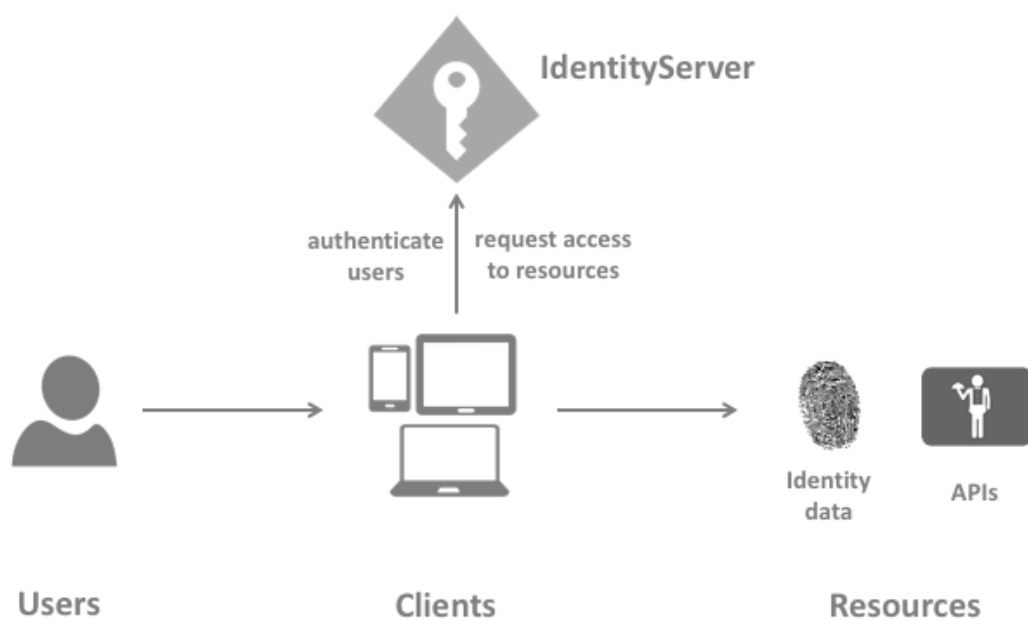
通常，您构建（或重用）一个包含登录和注销页面的应用程序（可能还包含同意页面 - 取决于您的需求），并且 IdentityServer 中间件向其中添加了必要的协议头，以便客户端应用程序可以使用那些标准协议与之对话。



托管应用程序可以任意复杂，但我们通常建议通过仅包含与身份验证相关的 UI 来尽可能地减少攻击面。

术语

规范、文档和对象模型使用了一些您应该了解的术语。



2.1 IdentityServer

IdentityServer 是一个 OpenID Connect 提供者——它实现了 OpenID Connect 和 OAuth 2.0 协议。

不同的文献对同一角色使用不同的术语——您可能还会发现安全令牌服务、身份提供者、授权服务器、IP-STS 等。

但简而言之，它们是一个向客户端发行安全令牌的软件。

IdentityServer 有许多作业和功能——包括：

- 保护您的资源
- 使用本地帐户存储或通过外部身份提供商对用户进行身份验证
- 提供会话管理和单点登录
- 管理和验证客户端
- 向客户端颁发身份和访问令牌
- 验证令牌

2.2 User

用户是使用已注册客户端访问资源的人。

2.3 Client

客户端是一个从 IdentityServer 请求令牌的软件——用于验证用户（请求身份令牌）或访问资源（请求访问令牌）。客户端必须先向 IdentityServer 注册，然后才能请求令牌。

客户端示例包括 Web 应用程序、原生移动或桌面应用程序、SPA、服务器进程等。

2.4 Resources

资源是您想要使用 IdentityServer 保护的东西——您的用户的身份数据或 API。

每个资源都有一个唯一的名称——客户端使用这个名称来指定他们想要访问哪些资源。

Identity data 关于用户的身份信息（又名 claim 声明），例如姓名或电子邮件地址。

APIs APIs 资源代表客户端想要调用的功能——通常建模为 Web API，但不一定。

2.5 Identity Token

身份令牌代表身份验证过程的结果。它至少包含用户的标识符（称为“sub”，又名 subject claim 主题声明）以及有关用户如何以及何时进行身份验证的信息。它可以包含额外的身份数据。

2.6 Access Token

访问令牌允许访问 API 资源。客户端请求访问令牌并将它们转发到 API。访问令牌包含有关客户端和用户（如果存在）的信息。API 使用该信息来授权对其数据的访问。

IdentityServer 实现了以下规范：

3.1 OpenID Connect

- OpenID Connect Core 1.0 ([spec](#))
- OpenID Connect Discovery 1.0 ([spec](#))
- OpenID Connect RP-Initiated Logout 1.0 - draft 01 ([spec](#))
- OpenID Connect Session Management 1.0 - draft 30 ([spec](#))
- OpenID Connect Front-Channel Logout 1.0 - draft 04 ([spec](#))
- OpenID Connect Back-Channel Logout 1.0 - draft 06 ([spec](#))

3.2 OAuth 2.0

- OAuth 2.0 ([RFC 6749](#))
- OAuth 2.0 Bearer Token Usage ([RFC 6750](#))
- OAuth 2.0 Multiple Response Types ([spec](#))
- OAuth 2.0 Form Post Response Mode ([spec](#))
- OAuth 2.0 Token Revocation ([RFC 7009](#))

- OAuth 2.0 Token Introspection ([RFC 7662](#))
- Proof Key for Code Exchange ([RFC 7636](#))
- JSON Web Tokens for Client Authentication ([RFC 7523](#))
- OAuth 2.0 Device Authorization Grant ([RFC 8628](#))
- OAuth 2.0 Mutual TLS Client Authentication and Certificate-Bound Access Tokens ([RFC 8705](#))
- JWT Secured Authorization Request ([draft](#))

IdentityServer 由许多 nuget 包组成。

4.1 IdentityServer4 主仓库

[github](#)

包含核心 IdentityServer 对象模型、服务和中间件以及 EntityFramework 和 ASP.NET Identity 集成。

nugets:

- [IdentityServer4](#)
- [IdentityServer4.EntityFramework](#)
- [IdentityServer4.AspNetIdentity](#)

4.2 快速入门用户界面

[github](#)

包含一个简单的入门 UI，包括登录、注销和同意页面。

4.3 访问令牌验证处理程序

[nuget](#) | [github](#)

用于验证 API 中的令牌的 ASP.NET Core 身份验证处理程序。处理程序允许在同一 API 中同时支持 JWT 和引用令牌。

4.4 模板

[nuget](#) | [github](#)

包含 dotnet CLI 的模板。

4.5 开发版本

此外，我们将 CI 构建发布到我们的包存储库。将以下 `nuget.config` 添加到您的项目中：

```
<?xml version="1.0" encoding="utf-8"?>
  <configuration>
    <packageSources>
      <clear />
      <add key="IdentityServer CI" value="https://www.myget.org/F/identity/api/
↪v3/index.json" />
    </packageSources>
  </configuration>
```

支持和咨询选项

我们为 IdentityServer 提供了多种免费和商业支持和咨询选项。

5.1 免费支持

免费支持基于社区并使用公共论坛。

StackOverflow

越来越多的人使用 IdentityServer 来监控 StackOverflow 上的问题。如果时间允许，我们也会尽量回答问题。

您可以使用此订阅源订阅所有与 IdentityServer4 相关的问题：

<https://stackoverflow.com/questions/tagged/?tagnames=identityserver4&sort=newest>

提出新问题时请使用 “IdentityServer4” 标签。

Gitter

您可以在我们的 Gitter 聊天室中与其他 IdentityServer4 用户聊天：

<https://gitter.im/IdentityServer/IdentityServer4>

报告 Bug

如果您认为您发现了 bug 或意外行为，请在 Github 问题跟踪器 上打开一个问题 (issue)。我们会尽快回复您。请理解我们也有日常工作，可能太忙而无法立即回复。

在发布之前，还需检查 贡献 指南。

5.2 商业支持

我们正在围绕身份和访问控制架构进行咨询、指导和定制软件开发，特别是 IdentityServer。请 [与我们联系](#)，讨论可能的选择。

训练

我们定期举办有关现代应用程序的身份和访问控制的研讨会。在 [这里](#) 查看议程和即将到来的公开日期。我们也可以在贵公司私下进行培训。联系[我们](#) 请求现场培训。

AdminUI、WS-Federation、SAML2p 和 FIDO2 支持

我们的合作伙伴 Rock Solid Knowledge 在 identityserver.com 上提供了商业附加产品。

CHAPTER 6

演示服务器

您可以使用自己喜欢的客户端库尝试 IdentityServer4。我们在 demo.identityserver.io 有一个测试实例。在主页上，您可以找到有关如何配置客户端以及如何调用 API 的说明。

我们对社区贡献非常开放，但是您应该遵循一些指导方针，这样我们就可以毫不费力地处理这个问题。

7.1 如何贡献？

最简单的贡献方式是打开一个问题 (issue) 并开始讨论。然后我们可以决定是否以及如何实现功能或更改。如果您应该提交带有代码更改的拉取请求 (pull request)，请从描述开始，只进行最小的更改并提供涵盖这些更改的测试。

另请先阅读：[成为一个优秀的开源公民](#)

7.2 一般反馈和讨论？

请在 [核心仓库问题跟踪器](#) 上开始讨论。

7.3 Bugs 和功能请求？

请在相应的 [GitHub](#) 存储库中记录新问题：

- [Core](#)
- [AccessTokenValidation](#)

7.4 贡献代码和内容

您需要先签署贡献者许可协议，然后才能贡献任何代码或内容。这是一个自动化过程，将在您打开拉取请求后开始。

7.5 贡献项目

如果您开始贡献项目（例如支持数据库 X 或配置存储 Y），我们将非常感谢。将它告诉我们，以便我们可以在我们的文档中发布和链接它。

我们通常不想拥有这些贡献库的所有权，我们已经非常忙于支持核心项目。

命名约定

截至 2017 年 10 月，IdentityServer4.* nuget 命名空间已为我们的包保留。请使用以下命名约定：

`YourProjectName.IdentityServer4`

或者

`IdentityServer4.Contrib>YourProjectName`

快速入门为各种常见的 IdentityServer 场景提供了分步说明。他们从绝对的基础开始，然后变得更加复杂——建议您按顺序进行。

- 将 IdentityServer 添加到 ASP.NET Core 应用程序
- 配置 IdentityServer
- 为各种客户端颁发令牌
- 保护 Web 应用程序和 API
- 添加对基于 EntityFramework 的配置的支持
- 添加对 ASP.NET Identity 的支持

每个快速入门都有一个参考解决方案——您可以在 [samples](#) 文件夹中找到代码。

8.1 准备工作

您应该做的第一件事是安装我们的模板：

```
dotnet new -i IdentityServer4.Templates
```

它们将用作各种教程的起点。

注解: 如果您使用私有 NuGet 源, 请不要忘记添加 `-nuget-source` 参数: `-nuget-source https://api.nuget.org/v3/index.json`

OK ——让我们开始吧!

注解: 快速入门针对 IdentityServer 4.x and ASP.NET Core 3.1.x ——还有 [ASP.NET Core 2](#) 和 [ASP.NET Core 1](#) 的快速入门.

使用客户端凭据保护 API

以下 Identity Server 4 快速入门提供了各种常见 IdentityServer 方案的分步说明。这些从绝对的基础开始，随着它们的进展变得更加复杂。我们建议您按顺序进行操作。

要查看完整列表，请转到 [IdentityServer4 快速入门概述](#)。

第一个快速入门是使用 IdentityServer 保护 API 的最基本场景。在本快速入门中，您将定义一个 API 和一个用于访问它的客户端。客户端将使用其客户端 ID 和密钥从身份服务器请求访问令牌，然后使用该令牌获得对 API 的访问权限。

9.1 源代码

与所有这些快速入门一样，您可以在 [IdentityServer4](#) 存储库中找到它的源代码。本快速入门的项目是 [快速入门 #1: 使用客户端凭据保护 API](#)。

9.2 准备工作

dotnet CLI 的 IdentityServer 模板是快速入门的良好起点。要安装模板，请打开控制台窗口并键入以下命令：

```
dotnet new -i IdentityServer4.Templates
```

它们将用作各种教程的起点。

9.3 设置 ASP.NET Core 应用程序

首先为应用程序创建一个目录——然后使用我们的模板创建一个包含基本 IdentityServer 设置的 ASP.NET Core 应用程序，例如：

```
md quickstart
cd quickstart

md src
cd src

dotnet new is4empty -n IdentityServer
```

这将创建以下文件：

- IdentityServer.csproj ——项目文件和 Properties\launchSettings.json 文件
- Program.cs 和 Startup.cs ——主应用程序入口点
- Config.cs ——IdentityServer 资源和客户端配置文件

您现在可以使用您喜欢的文本编辑器来编辑或查看文件。如果你想有 Visual Studio 支持，你可以添加这样的解决方案文件：

```
cd ..
dotnet new sln -n Quickstart
```

并让它添加您的 IdentityServer 项目（记住这个命令，因为我们将在下面创建其他项目）：

```
dotnet sln add .\src\IdentityServer\IdentityServer.csproj
```

注解：此模板中使用的协议是 https，在 Kestrel 上运行时端口设置为 5001 或在 IISExpress 上运行时设置为随机端口。您可以在 Properties\launchSettings.json 文件中更改它。对于生产场景，您应该始终使用 https。

9.4 定义 API Scope

API 是系统中要保护的资源。资源定义可以通过多种方式加载，您用于创建上述项目的模板显示了如何使用“代码即配置”方法。

已经为您创建了 Config.cs。打开它，更新代码看起来像这样：

```
public static class Config
{
    public static IEnumerable<ApiScope> ApiScopes =>
        new List<ApiScope>
        {
            new ApiScope("api1", "My API")
        };
}
```

(在 [这里](#) 查看完整的文件).

注解: 如果您将在生产中使用它, 那么为您的 API 提供一个逻辑名称很重要。开发人员将使用它通过您的身份服务器连接到您的 api。它应该用简单的术语向开发人员和用户描述您的 api。

9.5 定义客户端

下一步是定义一个客户端应用程序, 我们将使用它来访问我们的新 API。

对于这种情况, 客户端将没有交互式用户, 而是使用 IdentityServer 的所谓的客户端密钥进行身份验证。

为此, 添加客户端定义:

```
public static IEnumerable<Client> Clients =>
    new List<Client>
    {
        new Client
        {
            ClientId = "client",

            // 没有交互式用户, 使用 clientid/secret 进行身份验证
            AllowedGrantTypes = GrantTypes.ClientCredentials,

            // 用于身份验证的密钥
            ClientSecrets =
            {
                new Secret("secret".Sha256())
            },

            // 客户端有权访问的范围
            AllowedScopes = { "api1" }
        }
    };
};
```

您可以将 `ClientId` 和 `ClientSecret` 视为应用程序本身的登录名和密码。向身份服务器标识您的应用程序，以便它知道哪个应用程序正在尝试连接到它。

9.6 配置 IdentityServer

加载资源和客户端定义发生在 `Startup.cs` 中——将代码更新为如下所示：

```
public void ConfigureServices(IServiceCollection services)
{
    var builder = services.AddIdentityServer()
        .AddDeveloperSigningCredential() //
        ➔ 这仅适用于没有证书可以使用的开发场景。
        .AddInMemoryApiScopes(Config.ApiScopes)
        .AddInMemoryClients(Config.Clients);

    // 为简洁起见省略
}
```

就是这样——现在应该配置您的身份服务器。如果您运行服务器并将浏览器导航到 `https://localhost:5001/.well-known/openid-configuration`，您应该会看到所谓的发现文档。发现文档是身份服务器中的标准端点。您的客户端和 API 将使用发现文档来下载必要的配置数据。



```

{
  "issuer": "https://localhost:5001/",
  "jwks_uri": "https://localhost:5001/.well-known/openid-configuration/jwks",
  "authorization_endpoint": "https://localhost:5001/connect/authorize",
  "token_endpoint": "https://localhost:5001/connect/token",
  "userinfo_endpoint": "https://localhost:5001/connect/userinfo",
  "end_session_endpoint": "https://localhost:5001/connect/endsession",
  "check_session_iframe": "https://localhost:5001/connect/checksession",
  "revocation_endpoint": "https://localhost:5001/connect/revocation",
  "introspection_endpoint": "https://localhost:5001/connect/introspect",
  "device_authorization_endpoint": "https://localhost:5001/connect/deviceauthorization",
  "frontchannel_logout_supported": true,
  "frontchannel_logout_session_supported": true,
  "backchannel_logout_supported": true,
  "backchannel_logout_session_supported": true,
  "scopes_supported": [
    "openid",
    "offline_access"
  ],
  "claims_supported": [],
  "grant_types_supported": [
    "authorization_code",
    "client_credentials",
    "refresh_token",
    "implicit",
    "urn:ietf:params:oauth:grant-type:device_code"
  ],
  "response_types_supported": [
    "code",
    "token",
    "id_token",
    "id_token token",
    "code id_token",
    "code token",
    "code id_token token"
  ],
  "response_modes_supported": [
    "form_post",
    "query",
    "fragment"
  ],
  "token_endpoint_auth_methods_supported": [
    "client_secret_basic",
    "client_secret_post"
  ],
  "id_token_signing_alg_values_supported": [
    "RS256"
  ],
  "subject_types_supported": [
    "public"
  ],
  "code_challenge_methods_supported": [
    "plain",
    "S256"
  ],
  "request_parameter_supported": true
}

```

在第一次启动时，IdentityServer 会为你创建一个开发者签名密钥，它是一个名为 `tempkey.jwk` 的文件。您不必将该文件签入您的源代码管理，如果它不存在，它将被重新创建。

9.7 添加 API

接下来，向您的解决方案添加 API。

您可以使用 Visual Studio 中的 ASP.NET Core Web API 模板，也可以使用 .NET CLI 来创建 API 项目，就像我们在此处所做的那样。从 src 文件夹中运行以下命令：

```
dotnet new webapi -n Api
```

然后通过运行以下命令将其添加到解决方案中：

```
cd ..  
dotnet sln add .\src\Api\Api.csproj
```

将 API 应用程序配置为仅在 `https://localhost:6001` 上运行。您可以通过编辑 Properties 文件夹中的 `launchSettings.json` 文件来完成此操作。将应用程序 URL 设置更改为：

```
"applicationUrl": "https://localhost:6001"
```

9.7.1 控制器

添加一个名为 `IdentityController` 的新类：

```
[Route("identity")]  
[Authorize]  
public class IdentityController : ControllerBase  
{  
    [HttpGet]  
    public IActionResult Get()  
    {  
        return new JsonResult(from c in User.Claims select new { c.Type, c.Value });  
    }  
}
```

此控制器稍后将用于测试授权需求，以及通过 API 的眼睛可视化声明身份。

9.7.2 添加 Nuget 依赖项

为了使配置步骤工作，必须添加 **nuget** 包依赖项，在根目录中运行此命令：

```
dotnet add .\src\api\Api.csproj package Microsoft.AspNetCore.Authentication.  
    ↪ JwtBearer
```

9.7.3 配置

最后一步是将身份验证服务添加到 **DI**（依赖注入）并将身份验证中间件添加到管道中。这些将：

- 验证传入的令牌以确保它来自受信任的发行者
- 验证令牌是否有效与此 API 一起使用（又名 **audience** 受众）

将 *Startup* 更新为如下所示：

```
public class Startup  
{  
    public void ConfigureServices(IServiceCollection services)  
    {  
        services.AddControllers();  
  
        services.AddAuthentication("Bearer")  
            .AddJwtBearer("Bearer", options =>  
            {  
                options.Authority = "https://localhost:5001";  
  
                options.TokenValidationParameters = new TokenValidationParameters  
                {  
                    ValidateAudience = false  
                };  
            });  
    }  
  
    public void Configure(IApplicationBuilder app)  
    {  
        app.UseRouting();  
  
        app.UseAuthentication();  
        app.UseAuthorization();  
  
        app.UseEndpoints(endpoints =>  
        {  
            endpoints.MapControllers();  
        });  
    }  
}
```

(下页继续)

(续上页)

```
    });  
}  
}
```

- `AddAuthentication` 将身份验证服务添加到 DI 并将 `Bearer` 配置为默认方案。
- `UseAuthentication` 将身份验证中间件添加到管道中, 因此每次调用主机时都会自动执行身份验证。
- `UseAuthorization` 添加授权中间件以确保匿名客户端无法访问我们的 API 端点。

在浏览器上导航到控制器 `https://localhost:6001/identity` 应该返回 401 状态代码。这意味着您的 API 需要凭证并且现在受 IdentityServer 保护。

注解: 如果您想知道为什么上面的代码禁用了受众验证, 请查看[这里](#) 以获得更深入的讨论。

9.8 创建客户端

最后一步是编写一个请求访问令牌的客户端, 然后使用此令牌访问 API。为此, 在您的解决方案中添加一个控制台项目, 记住在 `src` 中创建它:

```
dotnet new console -n Client
```

然后和以前一样, 使用:

```
cd ..  
dotnet sln add .\src\Client\Client.csproj
```

IdentityServer 上的令牌端点实现了 OAuth 2.0 协议, 您可以使用原始 HTTP 来访问它。但是, 我们有一个名为 `IdentityModel` 的客户端库, 它将协议交互封装在一个易于使用的 API 中。

将 `IdentityModel` NuGet 包添加到您的客户端。这可以通过 Visual Studio 的 Nuget 包管理器或 `dotnet CLI` 完成:

```
cd src  
cd client  
dotnet add package IdentityModel
```

`IdentityModel` 包括一个与发现端点一起使用的客户端库。这样你只需要知道 IdentityServer 的基地址 - 可以从元数据中读取实际的端点地址:

```
// 从元数据中发现端点  
var client = new HttpClient();
```

(下页继续)

(续上页)

```
var disco = await client.GetDiscoveryDocumentAsync("https://localhost:5001");
if (disco.IsError)
{
    Console.WriteLine(disco.Error);
    return;
}
```

注解：如果您在连接时遇到错误，则可能是您正在运行 *https* 并且 *localhost* 的开发证书不受信任。您可以运行 `dotnet dev-certs https --trust` 以信任开发证书。这只需要做一次。

接下来，您可以使用发现文档中的信息向 **IdentityServer** 请求令牌以访问 *api1*：

```
// 请求令牌
var tokenResponse = await client.RequestClientCredentialsTokenAsync(new
    ClientCredentialsTokenRequest
{
    Address = disco.TokenEndpoint,

    ClientId = "client",
    ClientSecret = "secret",
    Scope = "api1"
});

if (tokenResponse.IsError)
{
    Console.WriteLine(tokenResponse.Error);
    return;
}

Console.WriteLine(tokenResponse.Json);
```

(完整文件可以在 [这里](#) 找到)

注解：将访问令牌从控制台复制并粘贴到 [jwt.ms](#) 以检查原始令牌。

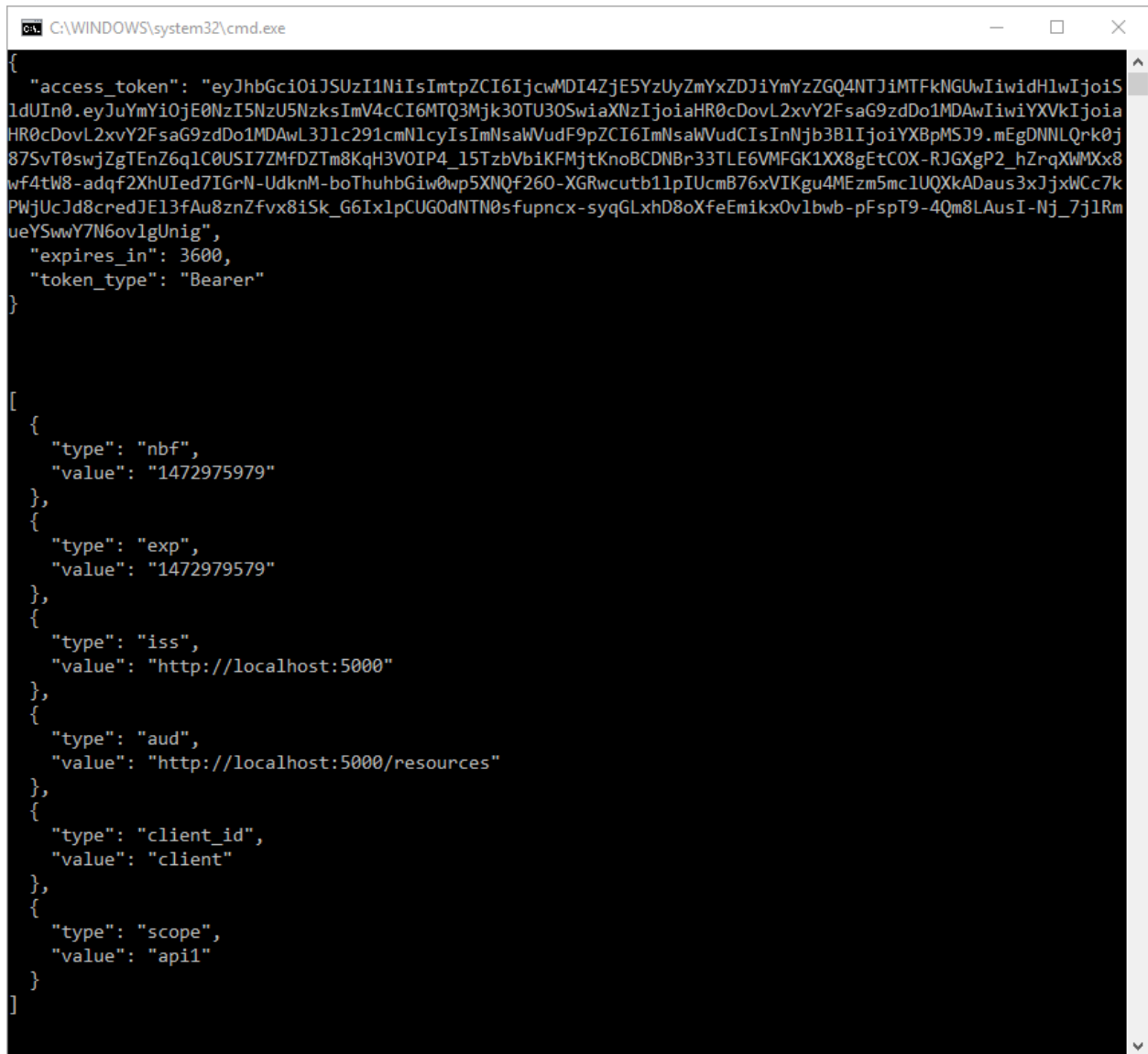
9.9 调用 API

要将访问令牌发送到 API，您通常使用 HTTP 授权标头。这是使用 `SetBearerToken` 扩展方法完成的：

```
// 调用 api
var apiClient = new HttpClient();
apiClient.SetBearerToken(tokenResponse.AccessToken);

var response = await apiClient.GetAsync("https://localhost:6001/identity");
if (!response.IsSuccessStatusCode)
{
    Console.WriteLine(response.StatusCode);
}
else
{
    var content = await response.Content.ReadAsStringAsync();
    Console.WriteLine(JArray.Parse(content));
}
```

（如果您在 Visual Studio 中，您可以右键单击解决方案并选择“多个启动项目”，并确保 Api 和 IdentityServer 将启动；然后运行解决方案；之后，要逐步执行客户端代码，您可以右键单击“客户端”项目并选择调试…启动新实例）。输出应该是这样的：



```

{
  "access_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6IjcwMDI4ZjE5YzUyZmYxZDZiYmYzZGQ4NTJiMTFkNGUwIiwidHlwIjoiaW50LmV4YmY0IjE0NzI5NzU5NzksImV4cCI6MTQ3Mjk3OTU3OSwiaXNzIjoiaHR0cDovL2xvY2FsaG9zdDo1MDAwIiwiaXVkiIjoiaHR0cDovL2xvY2FsaG9zdDo1MDAwL3J1c291cmNlcyIsImNsYWVudF9pZCI6ImNsakVudCIsInNjb3B1IjoiaXBpMSJ9.mEgDNNLQrk0j87SvT0swjZgTEEnZ6q1C0USI7ZMfDZTm8KqH3V0IP4_15TzbVbiKFMjtKnoBCDNBr33TLE6VMFGK1XX8gEtCOX-RJGXgP2_hZrqXMMx8wf4tW8-adqf2XhUIed7IGrN-UdknM-boThuhbGiw0wp5XNQf260-XGRwcutb1lpIUcmB76xVIKgu4MEzm5mc1UQXkADaus3xJjxWc7kPwjUcJd8credJE13fAu8znZfvx8iSk_G6IxlpCUGOdNTN0sfupncx-syqGLxhD8oXfeEmikxOv1bwb-pFspT9-4Qm8LAusI-Nj_7j1RmueYSwwY7N6ovlgUnig",
  "expires_in": 3600,
  "token_type": "Bearer"
}

[
  {
    "type": "nbf",
    "value": "1472975979"
  },
  {
    "type": "exp",
    "value": "1472979579"
  },
  {
    "type": "iss",
    "value": "http://localhost:5000"
  },
  {
    "type": "aud",
    "value": "http://localhost:5000/resources"
  },
  {
    "type": "client_id",
    "value": "client"
  },
  {
    "type": "scope",
    "value": "api1"
  }
]

```

注解：默认情况下，访问令牌将包含有关范围、生命周期（nbf 和 exp）、客户端 ID (client_id) 和颁发者名称 (iss) 的声明。

9.10 API 授权

现在，API 接受您的身份服务器颁发的任何访问令牌。

在下文中，我们将添加允许检查范围是否存在于客户端请求（并被授权）的访问令牌中的代码。为此，我们将使用 ASP.NET Core 授权策略系统。将以下内容添加到 Startup 中的 ConfigureServices 方法中：

```
services.AddAuthorization(options =>
{
    options.AddPolicy("ApiScope", policy =>
    {
        policy.RequireAuthenticatedUser();
        policy.RequireClaim("scope", "api1");
    });
});
```

您现在可以在各个级别执行此策略，例如

- 全局
- 对于所有 API 端点
- 对特定的 controllers/actions

通常，您为路由系统中的所有 API 端点设置策略：

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers()
        .RequireAuthorization("ApiScope");
});
```

9.11 进一步的实验

本演练重点介绍了迄今为止的成功路径

- 客户端能够请求令牌
- 客户端可以使用令牌访问 API

您现在可以尝试引发错误以了解系统的行为方式，例如

- 尝试在 IdentityServer 未运行时连接到它（不可用）
- 尝试使用无效的客户端 ID 或机密来请求令牌
- 尝试在令牌请求期间请求无效范围
- 尝试在未运行时调用 API（不可用）
- 不要将令牌发送到 API
- 将 API 配置为需要与令牌中的范围不同的范围

CHAPTER 10

使用 ASP.NET Core 的交互式应用程序

注解： 对于任何先决条件（例如模板），请先查看 *overview*。

在本快速入门中，我们希望通过 OpenID Connect 协议向我们在上一章中构建的 IdentityServer 添加对交互式用户身份验证的支持。

一旦到位，我们将创建一个 MVC 应用程序，该应用程序将使用 IdentityServer 进行身份验证。

10.1 添加用户界面

OpenID Connect 所需的所有协议支持都已内置到 IdentityServer 中。您需要为登录、注销、同意和错误提供必要的 UI 部分。

虽然每个 IdentityServer 实现的外观和感觉以及确切的工作流程可能总是有所不同，但我们提供了一个基于 MVC 的示例 UI，您可以将其用作起点。

这个 UI 可以在 [Quickstart UI 仓库](#) 中找到。您可以克隆或下载此存储库并将控制器、视图、模型和 CSS 放入您的 IdentityServer Web 应用程序中。

或者，您可以使用 .NET CLI（从 `src/IdentityServer` 文件夹中运行）：

```
dotnet new is4ui
```

添加 MVC UI 后，您还需要在 DI 系统和管道中启用 MVC。当您查看 `Startup.cs` 时，您会发现 `ConfigureServices` 和 `Configure` 方法中的注释告诉您如何启用 MVC。

注解: 还有一个名为 `is4inmem` 的模板，它结合了包括标准 UI 在内的基本 IdentityServer。

运行 IdentityServer 应用程序，您现在应该会看到一个主页。

花一些时间检查控制器和模型——特别是 `AccountController`，它是主要的 UI 入口点。您对它们理解得越好，将来进行修改就越容易。大多数代码使用“功能文件夹”样式位于“Quickstart”文件夹中。如果这种风格不适合您，请随意以您想要的任何方式组织代码。

10.2 创建 MVC 客户端

接下来，您将创建一个 MVC 应用程序。为此使用 ASP.NET Core “Web 应用程序”（即 MVC）模板。

从 `src` 文件夹运行：

```
dotnet new mvc -n MvcClient
cd ..
dotnet sln add .\src\MvcClient\MvcClient.csproj
```

注解: 我们建议通过 IIS Express 使用自托管选项。其余文档假设您在端口 5002 上使用自托管。

要向 MVC 应用程序添加对 OpenID Connect 身份验证的支持，您首先需要将包含 OpenID Connect 处理程序的 `nuget` 包添加到您的项目中，例如：

```
dotnet add package Microsoft.AspNetCore.Authentication.OpenIdConnect
```

.. 然后将以下内容添加到 `Startup` 中的 `ConfigureServices`

```
using System.IdentityModel.Tokens.Jwt;

// ...

JwtSecurityTokenHandler.DefaultMapInboundClaims = false;

services.AddAuthentication(options =>
{
    options.DefaultScheme = "Cookies";
    options.DefaultChallengeScheme = "oidc";
})
.AddCookie("Cookies")
.AddOpenIdConnect("oidc", options =>
{

```

(下页继续)

(续上页)

```
options.Authority = "https://localhost:5001";

options.ClientId = "mvc";
options.ClientSecret = "secret";
options.ResponseType = "code";

options.SaveTokens = true;
});
```

AddAuthentication 将身份验证服务添加到 DI。

我们使用 cookie 在本地登录用户（通过 "Cookies" 作为 DefaultScheme），我们将 DefaultChallengeScheme 设置为 oidc，因为当我们需要用户登录，我们将使用 OpenID Connect 协议。

然后我们使用 AddCookie 来添加可以处理 cookie 的处理程序。

最后，AddOpenIdConnect 用于配置执行 OpenID Connect 协议的处理程序。Authority 指示可信令牌服务所在的位置。然后通过 ClientId 和 ClientSecret 来识别这个客户端。SaveTokens 用于将来自 IdentityServer 的令牌持久保存在 cookie 中（因为稍后将需要它们）。

注解： 我们使用所谓的 授权码 (authorization code) 流程与 PKCE 连接到 OpenID Connect 提供程序。有关协议流的更多信息，请参见[这里](#)。

然后为了确保在每个请求上执行身份验证服务，将 UseAuthentication 添加到 Startup 中的 Configure

```
app.UseStaticFiles();

app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapDefaultControllerRoute()
        .RequireAuthorization();
});
```

注解： RequireAuthorization 方法禁用对整个应用程序的匿名访问。

如果您想在每个控制器或操作方法的基础上指定授权，您还可以使用 [Authorize] 属性。

还要修改主页视图以显示用户的声明以及 cookie 属性:

```
@using Microsoft.AspNetCore.Authentication

<h2>Claims</h2>

<dl>
    @foreach (var claim in User.Claims)
    {
        <dt>@claim.Type</dt>
        <dd>@claim.Value</dd>
    }
</dl>

<h2>Properties</h2>

<dl>
    @foreach (var prop in (await Context.AuthenticateAsync()).Properties.Items)
    {
        <dt>@prop.Key</dt>
        <dd>@prop.Value</dd>
    }
</dl>
```

如果您现在使用浏览器导航到应用程序，则会尝试重定向到 IdentityServer —— 这将导致错误，因为 MVC 客户端尚未注册。

10.3 添加对 OpenID Connect 身份范围的支持

与 OAuth 2.0 类似，OpenID Connect 也使用范围 (scopes) 概念。同样，范围代表您想要保护和客户端想要访问的东西。与 OAuth 相比，OIDC 中的范围不代表 API，而是代表用户 ID、姓名或电子邮件地址等身份数据。

通过修改 Config.cs 中的 IdentityResources 属性，添加对标准 openid (subject id) 和 profile (名字、姓氏等) 范围的支持:

```
public static IEnumerable<IdentityResource> IdentityResources =>
    new List<IdentityResource>
    {
        new IdentityResources.OpenId(),
        new IdentityResources.Profile(),
    };
```

在 startup.cs 中向 IdentityServer 注册身份资源:

```
var builder = services.AddIdentityServer()
    .AddInMemoryIdentityResources(Config.IdentityResources)
    .AddInMemoryApiScopes(Config.ApiScopes)
    .AddInMemoryClients(Config.Clients);
```

注解：所有标准范围及其相应的声明都可以在 [OpenID Connect 规范](#) 中找到。

10.4 添加测试用户

示例 UI 还带有一个内存中的“用户数据库”。您可以通过添加 `AddTestUsers` 扩展方法在 `IdentityServer` 中启用此功能：

```
var builder = services.AddIdentityServer()
    .AddInMemoryIdentityResources(Config.IdentityResources)
    .AddInMemoryApiScopes(Config.ApiScopes)
    .AddInMemoryClients(Config.Clients)
    .AddTestUsers(TestUsers.Users);
```

当您导航到 `TestUsers` 类时，您可以看到定义了两个名为 `alice` 和 `bob` 的用户以及一些身份声明。您可以使用这些用户登录。

10.5 将 MVC 客户端添加到 IdentityServer 配置

最后一步是将 MVC 客户端的新配置条目添加到 `IdentityServer`。

基于 `OpenID Connect` 的客户端与我们目前添加的 `OAuth 2.0` 客户端非常相似。但由于 `OIDC` 中的流程始终是交互式的，我们需要在配置中添加一些重定向 URL。

客户端列表应如下所示：

```
public static IEnumerable<Client> Clients =>
    new List<Client>
    {
        // 机器到机器客户端（来自快速入门 1 开始）
        new Client
        {
            ClientId = "client",
            ClientSecrets = { new Secret("secret".Sha256()) },

            AllowedGrantTypes = GrantTypes.ClientCredentials,
```

(下页继续)

(续上页)

```
// 客户端有权访问的范围
AllowedScopes = { "api1" }

},

// 交互式 ASP.NET Core MVC 客户端
new Client
{
    ClientId = "mvc",
    ClientSecrets = { new Secret("secret".Sha256()) },

    AllowedGrantTypes = GrantTypes.Code,

    // 登录后重定向到哪里
    RedirectUri = { "https://localhost:5002/signin-oidc" },

    // 注销后重定向到哪里
    PostLogoutRedirectUri = { "https://localhost:5002/signout-callback-oidc" },

},

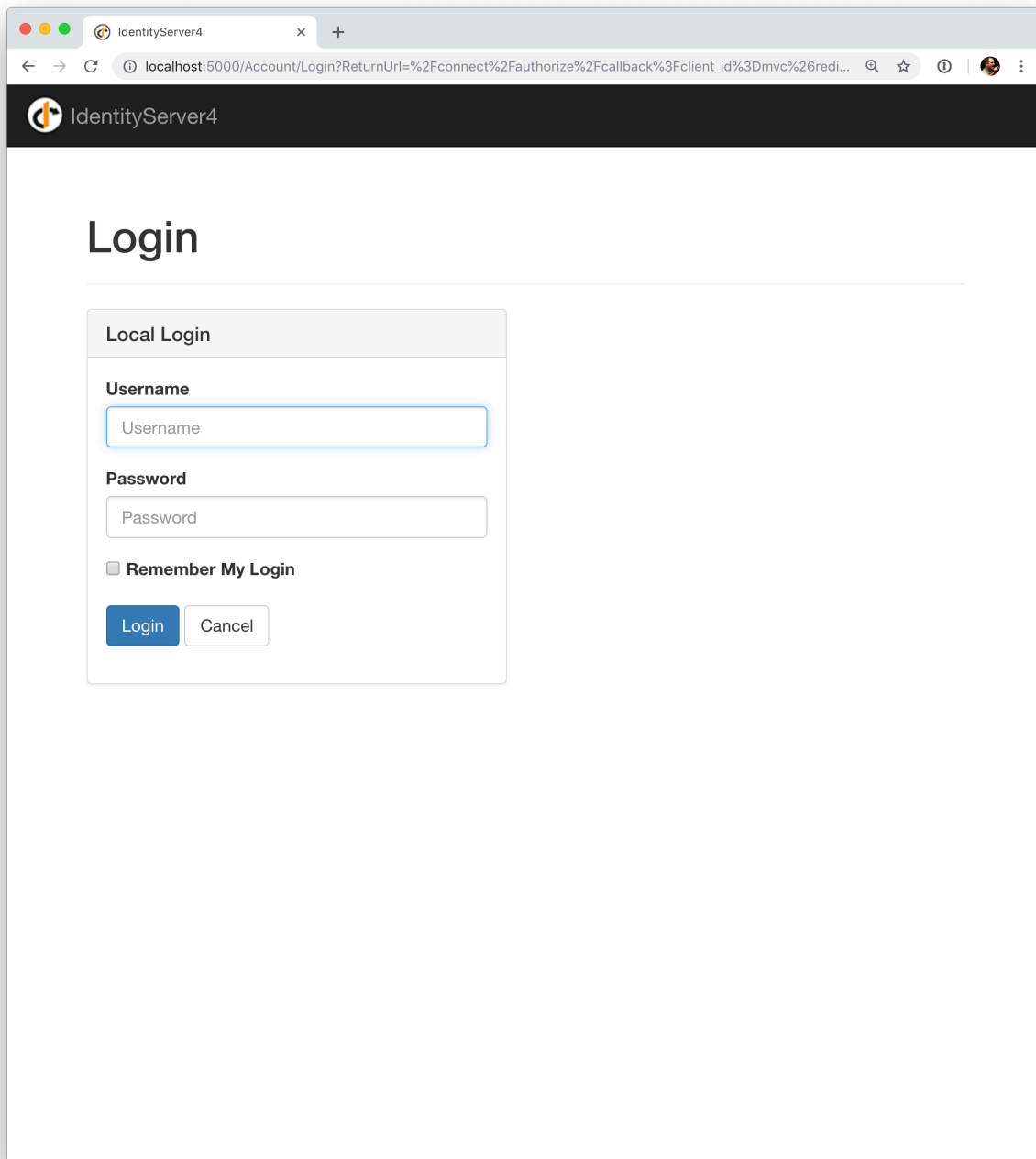
AllowedScopes = new List<string>
{
    IdentityServerConstants.StandardScopes.OpenId,
    IdentityServerConstants.StandardScopes.Profile
}

};
```

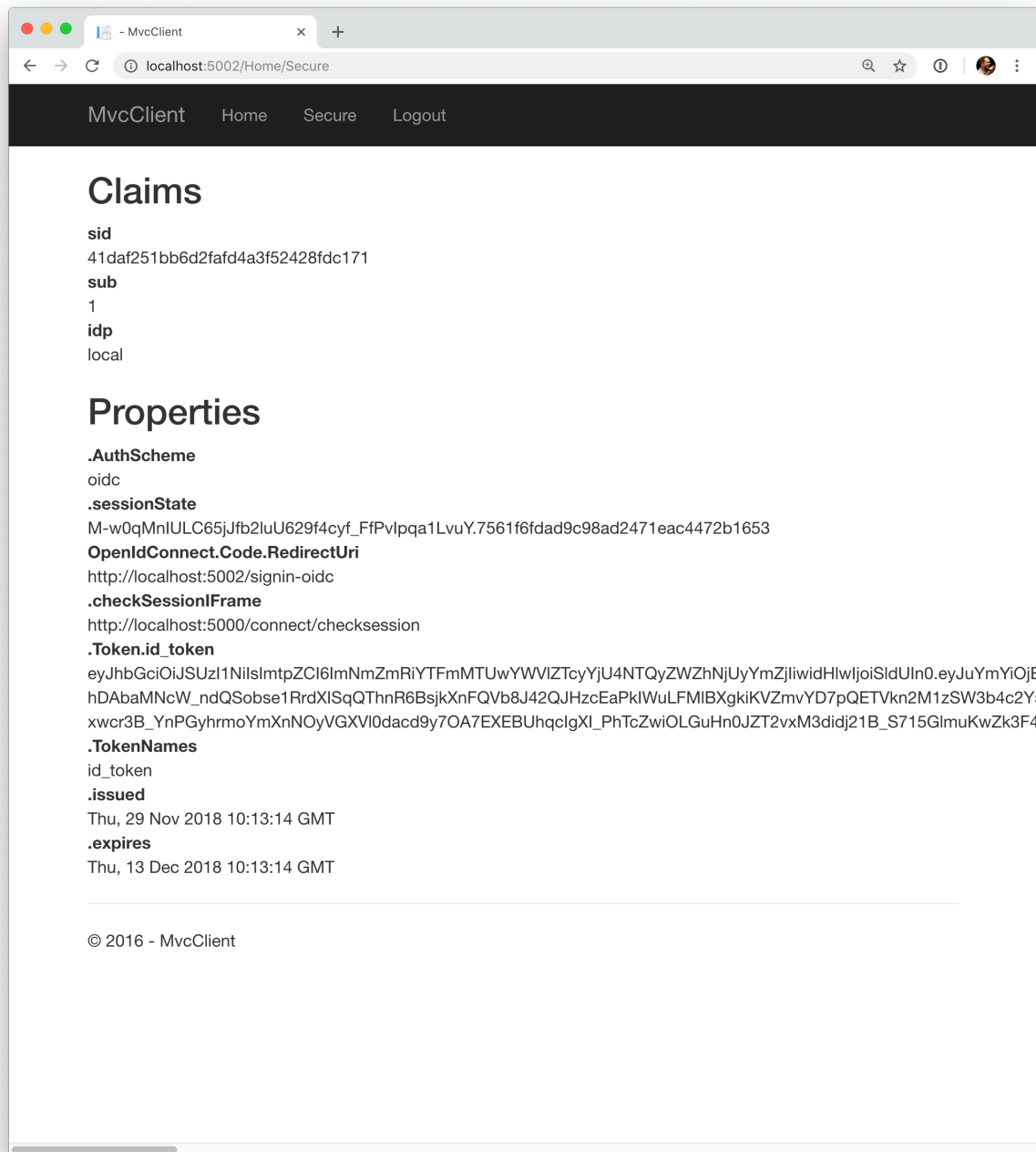
10.6 测试客户端

现在，新的 MVC 客户端的一切都应该就绪了。

通过导航到受保护的控制器操作来触发身份验证握手。您应该会看到重定向到 IdentityServer 的登录页面。



之后，IdentityServer 将重定向回 MVC 客户端，在那里 OpenID Connect 身份验证处理程序处理响应，并通过设置 cookie 在本地登录用户。最后，MVC 视图将显示 cookie 的内容。



如您所见，cookie 有两部分，用户的声明和一些元数据。此元数据还包含 IdentityServer 发布的原始令牌。随意将此令牌复制到 jwt.ms 以检查其内容。

10.7 添加注销

最后一步是向 MVC 客户端添加注销。

使用 IdentityServer 这样的身份验证服务，仅清除本地应用程序 cookie 是不够的。此外，您还需要往返 IdentityServer 以清除中央单点登录会话。

确切的协议步骤在 OpenID Connect 处理程序中实现，只需将以下代码添加到某个控制器即可触发注销：

```
public IActionResult Logout ()
{
    return SignOut ("Cookies", "oidc");
}
```

这将清除本地 cookie，然后重定向到 IdentityServer。IdentityServer 将清除其 cookie，然后为用户提供一个链接以返回到 MVC 应用程序。

10.8 从 UserInfo 端点获取声明

您可能已经注意到，即使我们已将客户端配置为允许检索 profile 身份范围，与该范围关联的声明（例如 name, family_name, website 等）不会出现在返回的令牌中。我们需要告诉客户端应用程序，通过指定需要访问的范围并设置 GetClaimsFromUserInfoEndpoint 选项，来从 UserInfo 端点中提取剩余声明。在以下示例中，我们请求的是 profile 范围，但它可以是客户端有权访问的任何范围（或多个范围）：

```
.AddOpenIdConnect ("oidc", options =>
{
    // ...
    options.Scope.Add ("profile");
    options.GetClaimsFromUserInfoEndpoint = true;
    // ...
});
```

重新启动客户端应用程序、注销并重新登录后，您应该会看到与页面上显示的 profile 身份范围关联的其他用户声明。

s hash

NE2NZYBuxHyXGR3hQQh_Ww

sid

D41C360A757E58F528556410EED249DF

sub

818727

auth time

1601461489

idp

local

amr

pwd

name

Alice Smith

given_name

Alice

family name

Smith

AuthScheme

oidc

Token.access token

```
eylhbGaOiSUzI1NiIsImtpZCI6IhRBNkU2RUQ5NTJCMUEyRjk0MUU3REU5MDdBOQFBRTBliiwiIDhlwjljYXQrand0In0.eyJ1YmYiOiJlMDE0NjE0OTAsImV4cCI6MTY5XSYtLSMmtpDEAlpJ2i45v1nO0oue-cUm8n9udwNND0uP93V2TQdJCOsnU4kXhLRqskvWRN6uc1oCh-
HhyjQwPYw9lQdQK5w7I1wPkwGefnAchH9CxtJCWBgeE5rdE-
GeBdP1kGI3ik88Z2tVLh8NKWmf0Y9N8ICVzmocKUZ71G4FI8HD2W5IPRDacvs adap/rwLwYp0GfDpplWCJGk 95A67l 0ao0F7Lzm8rZaLr4H2K8U1Kawa
```

.sessionState

KvnEv7ATw4XfdJuUqY68Xu2YT7mJNeJYtEeOnmq9uMQ.78528A880E8A4B740ECFD774926BF659

隨意向測試用戶添加更多聲明——以及更多身份資源。

定义身份资源的过程如下:

- 将新的身份资源添加到列表中——为其命名并指定在请求此资源时应返回哪些声明
- 通过客户端配置上的 `AllowedScopes` 属性让客户端访问资源
- 通过将资源添加到客户端中 **OpenID Connect** 处理程序配置的 `Scopes` 集合来请求资源
- （可选）如果身份资源与非标准声明（例如 `myclaim1`）相关联，则在客户端添加 JSON 中出现的声明（从 `UserInfo` 端点返回）和用户 **声明** 之间的 **ClaimAction** 映射

```
using Microsoft.AspNetCore.Authentication // ... .AddOpenIdConnect( "oidc" , options => {  
    // ... options.ClaimActions.MapUniqueJsonKey( "myclaim1" , "myclaim1" ); // ...
```



```
});
```

同样值得注意的是，令牌声明的检索是一个扩展点——`IProfileService`。由于我们使用的是 `AddTestUsers`，因此默认使用 `TestUserProfileService`。您可以在 [此处](#) 检查源代码以了解其工作原理。

10.10 添加对外部身份验证的支持

接下来，我们将添加对外部身份验证的支持。这真的很简单，因为您真正需要的只是一个与 ASP.NET Core 兼容的身份验证处理程序。

ASP.NET Core 本身支持 Google、Facebook、Twitter、Microsoft Account 和 OpenID Connect。此外，您可以在 [此处](#) 找到许多其他身份验证提供程序的实现。

10.11 添加 Google 支持

为了能够使用 Google 进行身份验证，您首先需要向他们注册。这是在他们的开发人员 [控制台](#) 上完成的。创建一个新项目，启用 Google+ API，并通过将 `/signin-google` 路径添加到您的基地址（例如 <https://localhost:5001/signin-google>）来配置您本地 IdentityServer 的回调地址。

开发者控制台将向您显示由 Google 发布的客户端 ID 和密码——下一步你会用到的。

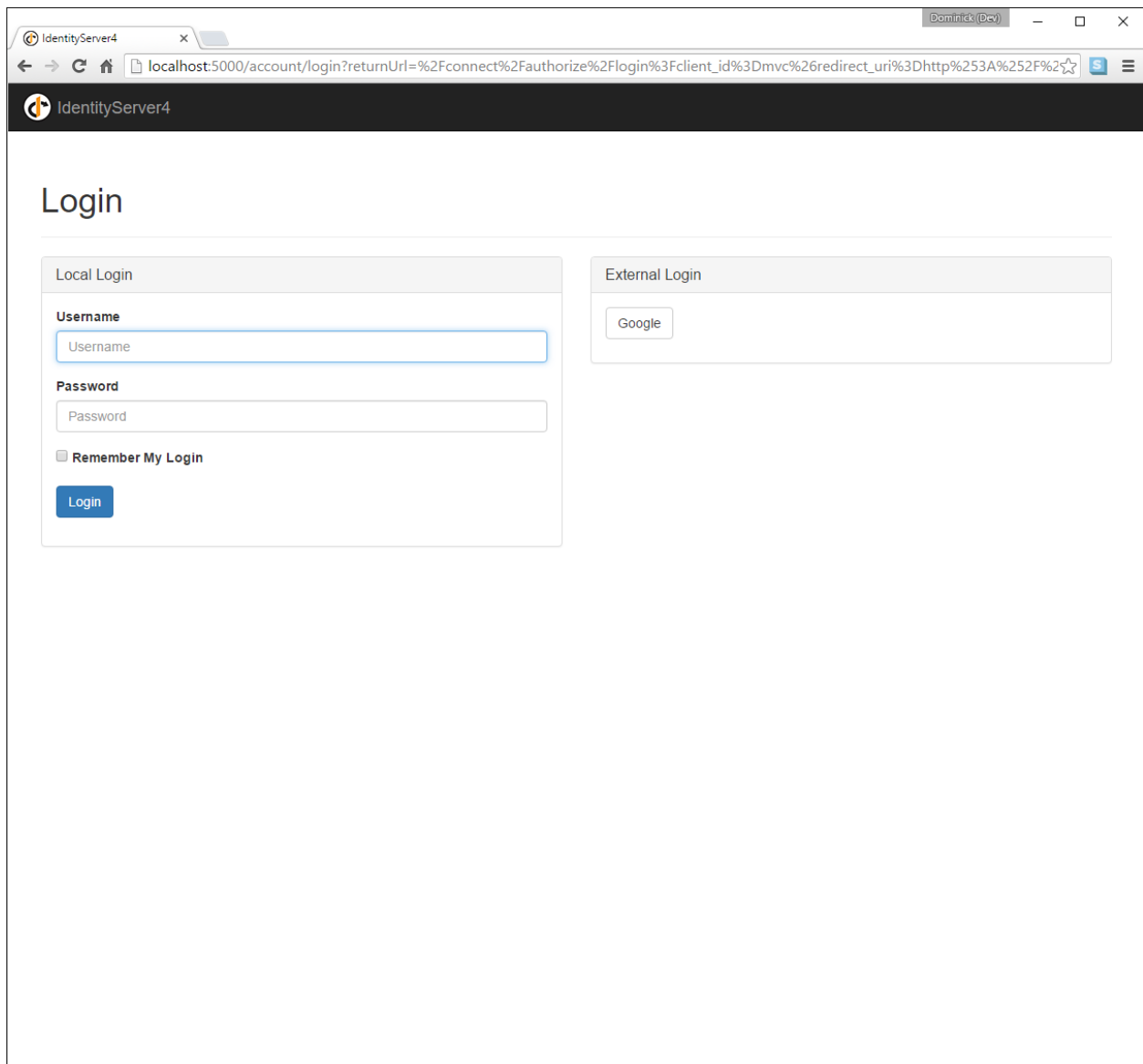
将 Google 身份验证处理程序添加到 IdentityServer 主机的 DI。首先添加 `Microsoft.AspNetCore.Authentication.Google` nuget 包，然后将此代码段添加到 Startup 中的 `ConfigureServices` 中：

```
services.AddAuthentication()
    .AddGoogle("Google", options =>
    {
        options.SignInScheme = IdentityServerConstants.
↪ExternalCookieAuthenticationScheme;

        options.ClientId = "<在此处插入>";
        options.ClientSecret = "<在此处插入>";
    });
```

默认情况下，IdentityServer 专门为外部身份验证的结果配置 cookie 处理程序（使用基于常量 `IdentityServerConstants.ExternalCookieAuthenticationScheme` 的方案）。Google 处理程序的配置将使用该 cookie 处理程序。

现在运行 MVC 客户端并尝试进行身份验证——您将在登录页面上看到一个 Google 按钮：



在使用 MVC 客户端进行身份验证之后，您可以看到这些声明现在来自 Google 数据。

注解： 如果您对在登录页面上自动呈现 Google 按钮的魔法感兴趣，请检查 AccountController 上的 BuildLoginViewModel 方法。

10.12 进一步的实验

您可以添加额外的外部提供程序。我们有一个 [云托管演示](#) 版本的 IdentityServer4, 您可以使用 OpenID Connect 进行集成。

将 OpenId Connect 处理程序添加到 DI:

```
services.AddAuthentication()
    .AddGoogle("Google", options =>
    {
        options.SignInScheme = IdentityServerConstants.
↪ExternalCookieAuthenticationScheme;

        options.ClientId = "<在此处插入>";
        options.ClientSecret = "<在此处插入>";
    })
    .AddOpenIdConnect("oidc", "Demo IdentityServer", options =>
    {
        options.SignInScheme = IdentityServerConstants.
↪ExternalCookieAuthenticationScheme;
        options.SignOutScheme = IdentityServerConstants.SignoutScheme;
        options.SaveTokens = true;

        options.Authority = "https://demo.identityserver.io/";
        options.ClientId = "interactive.confidential";
        options.ClientSecret = "secret";
        options.ResponseType = "code";

        options.TokenValidationParameters = new TokenValidationParameters
        {
            NameClaimType = "name",
            RoleClaimType = "role"
        };
    });
```

现在, 用户应该能够使用云托管的演示身份提供程序。

注解: 快速入门 UI 自动配置外部用户。当外部用户第一次登录时, 会创建一个新的本地用户, 所有外部声明都被复制并与新用户关联。但是, 您处理这种情况的方式完全取决于您。也许您想先显示一些注册 UI。可以在 [这里](#) 找到默认快速入门的源代码。可以在 [此处](#) 找到执行自动配置的控制​​器。

ASP.NET Core 和 API 访问

在之前的快速入门中，我们探讨了 API 访问和用户身份验证。现在我们把这两个部分放在一起。

OpenID Connect 和 OAuth 2.0 结合的美妙之处在于，您可以通过单个协议和与令牌服务的单个交换来实现。

到目前为止，我们只在令牌请求期间请求身份资源，一旦我们开始包括 API 资源，IdentityServer 将返回两个令牌：包含身份验证和会话信息的身份令牌，以及代表登录用户访问 API 的访问令牌。

11.1 修改客户端配置

在 IdentityServer 中更新客户端配置很简单——我们只需要将 api1 资源添加到允许的范围列表中。此外，我们通过 AllowOfflineAccess 属性启用对刷新令牌的支持：

```
new Client
{
    ClientId = "mvc",
    ClientSecrets = { new Secret("secret".Sha256()) },

    AllowedGrantTypes = GrantTypes.Code,

    // 登录后重定向到哪里
    RedirectUri = { "https://localhost:5002/signin-oidc" },

    // 注销后重定向到哪里
    PostLogoutRedirectUri = { "https://localhost:5002/signout-callback-oidc" },
```

(下页继续)

(续上页)

```
AllowOfflineAccess = true,

AllowedScopes = new List<string>
{
    IdentityServerConstants.StandardScopes.OpenId,
    IdentityServerConstants.StandardScopes.Profile,
    "api1"
}
}
```

11.2 修改 MVC 客户端

现在客户端要做的就是通过 scope 参数请求额外的资源。这是在 OpenID Connect 处理程序配置中完成的：

```
services.AddAuthentication(options =>
{
    options.DefaultScheme = "Cookies";
    options.DefaultChallengeScheme = "oidc";
})
.AddCookie("Cookies")
.AddOpenIdConnect("oidc", options =>
{
    options.Authority = "https://localhost:5001";

    options.ClientId = "mvc";
    options.ClientSecret = "secret";
    options.ResponseType = "code";

    options.SaveTokens = true;

    options.Scope.Add("api1");
    options.Scope.Add("offline_access");
});
```

由于启用了 SaveTokens，ASP.NET Core 将自动将结果访问和刷新令牌存储在身份验证会话中。您应该能够检查打印出您之前创建的会话内容的页面上的数据。

11.3 使用访问令牌

您可以使用可以在 `Microsoft.AspNetCore.Authentication` 命名空间中找到的标准 ASP.NET Core 扩展方法来访问会话中的令牌:

```
var accessToken = await HttpContext.GetTokenAsync("access_token");
```

要使用访问令牌访问 API，您需要做的就是检索令牌，并将其设置在您的 `HttpClient` 上:

```
public async Task<IActionResult> CallApi()
{
    var accessToken = await HttpContext.GetTokenAsync("access_token");

    var client = new HttpClient();
    client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", accessToken);
    var content = await client.GetStringAsync("https://localhost:6001/identity");

    ViewBag.Json = JObject.Parse(content).ToString();
    return View("json");
}
```

创建一个名为 `json.cshtml` 的视图，输出如下所示的 json:

```
<pre>@ViewBag.Json</pre>
```

确保 API 正在运行，启动 MVC 客户端并在身份验证后调用 `/home/CallApi`。

11.4 管理访问令牌

到目前为止，典型客户端最复杂的任务是管理访问令牌。你通常想要

- 在登录时请求访问和刷新令牌
- 缓存这些令牌
- 使用访问令牌调用 API，直到它过期
- 使用刷新令牌获取新的访问令牌
- 重新开始

ASP.NET Core 有许多内置工具可以帮助您完成这些任务（如缓存或 sessions），但仍有很多工作要做。随意看看 [这个库](#)，它可以自动执行许多样板任务。

CHAPTER 12

添加 JavaScript 客户端

注解： 对于任何先决条件（例如模板），请先查看概述。

本快速入门将展示如何构建基于浏览器的 JavaScript 客户端应用程序（有时称为“单页应用程序”或“SPA”）。用户将登录到 IdentityServer，使用 IdentityServer 颁发的访问令牌调用 Web API，然后注销 IdentityServer。所有这一切都将由在浏览器中运行的 JavaScript 驱动。

12.1 新的 JavaScript 客户端项目

为 JavaScript 应用程序创建一个新项目。它可以只是一个空的 Web 项目、一个空的 ASP.NET Core 应用程序或其他类似 Node.js 的应用程序。本快速入门将使用 ASP.NET Core 应用程序。

在 `~/src` 目录中创建一个新的“Empty”ASP.NET Core Web 应用程序。您可以使用 Visual Studio 或从命令行执行此操作：

```
md JavaScriptClient
cd JavaScriptClient
dotnet new web
```

正如我们之前所做的那样，对于其他客户端项目，也将此项目添加到您的解决方案中。从包含 `sln` 文件的根文件夹运行它：

```
dotnet sln add .\src\JavaScriptClient\JavaScriptClient.csproj
```

12.2 修改托管

修改 *JavaScriptClient* 项目以在 <https://localhost:5003> 上运行。

12.3 添加静态文件中间件

鉴于此项目旨在运行客户端，我们需要 ASP.NET Core 做的就是提供构成我们应用程序的静态 HTML 和 JavaScript 文件。静态文件中间件就是为此而设计的。

在 *Startup.cs* 中的 *Configure* 方法中注册静态文件中间件（同时删除其他所有内容）：

```
public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
}
```

这个中间件现在将从应用程序的 *~/wwwroot* 文件夹中提供静态文件。这是我们将放置 HTML 和 JavaScript 文件的地方。如果您的项目中不存在该文件夹，请立即创建它。

12.4 引用 oidc-client

在之前基于 ASP.NET Core MVC 的客户端项目的快速入门之一中，我们使用了一个库来处理 OpenID Connect 协议。在 *JavaScriptClient* 项目的这个快速入门中，我们需要一个类似的库，一个只在 JavaScript 中工作并设计为在浏览器中运行的库。*oidc-client* 库 就是这样的一个库。它可以通过 [NPM](#)，[Bower](#)，以及从 [github](#) 的 [直接下载](#) 获得。

NPM

如果你想使用 NPM 下载 *oidc-client*，那么从你的 *JavaScriptClient* 项目目录运行这些命令：

```
npm i oidc-client
copy node_modules\oidc-client\dist\* wwwroot
```

这会在本地下载最新的 *oidc-client* 包，然后将相关的 JavaScript 文件复制到 *~/wwwroot* 中，以便您的应用程序提供它们。

手动下载

如果您只想手动下载 *oidc-client* JavaScript 文件，请浏览到 [GitHub 存储库](#) 并下载 JavaScript 文件。下载后，将它们复制到 `~/wwwroot` 中，以便您的应用程序提供它们。

12.5 添加您的 HTML 和 JavaScript 文件

接下来是将您的 HTML 和 JavaScript 文件添加到 `~/wwwroot`。我们将有两个 HTML 文件和一个特定于应用程序的 JavaScript 文件（除了 *oidc-client.js* 库）。在 `~/wwwroot` 中，添加名为 *index.html* 和 *callback.html* 的 HTML 文件，并添加名为 *app.js* 的 JavaScript 文件。

index.html

这将是您的应用程序的主页。它将只包含用于用户登录、注销和调用 Web API 的按钮的 HTML。它还将包含 `<script>` 标签以包含我们的两个 JavaScript 文件。它还将包含一个 `<pre>` 用于向用户显示消息。

它应该是这样的：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title></title>
</head>
<body>
  <button id="login">Login</button>
  <button id="api">Call API</button>
  <button id="logout">Logout</button>

  <pre id="results"></pre>

  <script src="oidc-client.js"></script>
  <script src="app.js"></script>
</body>
</html>
```

app.js

这将包含我们应用程序的主要代码。首先是添加一个辅助函数来将消息记录到 `<pre>`：

```
function log() {
  document.getElementById('results').innerText = '';

  Array.prototype.forEach.call(arguments, function (msg) {
    if (msg instanceof Error) {
      msg = "Error: " + msg.message;
    }
  })
}
```

(下页继续)

(续上页)

```
    else if (typeof msg !== 'string') {
        msg = JSON.stringify(msg, null, 2);
    }
    document.getElementById('results').innerHTML += msg + '\r\n';
});
}
```

接下来，添加代码以将 click 事件处理程序注册到三个按钮：

```
document.getElementById("login").addEventListener("click", login, false);
document.getElementById("api").addEventListener("click", api, false);
document.getElementById("logout").addEventListener("click", logout, false);
```

接下来，我们可以使用 *oidc-client* 库中的 *UserManager* 类来管理 OpenID Connect 协议。它需要 MVC 客户端中所必需的类似配置（尽管具有不同的值）。添加此代码以配置和实例化 *UserManager*：

```
var config = {
    authority: "https://localhost:5001",
    client_id: "js",
    redirect_uri: "https://localhost:5003/callback.html",
    response_type: "code",
    scope: "openid profile api1",
    post_logout_redirect_uri : "https://localhost:5003/index.html",
};
var mgr = new Oidc.UserManager(config);
```

接下来，*UserManager* 提供了一个 *getUser* API 来了解用户是否已登录到 JavaScript 应用程序。它使用 JavaScript Promise 异步返回结果。返回的 *User* 对象有一个 *profile* 属性，其中包含用户的声明。添加此代码以检测用户是否登录到 JavaScript 应用程序：

```
mgr.getUser().then(function (user) {
    if (user) {
        log("User logged in", user.profile);
    }
    else {
        log("User not logged in");
    }
});
```

接下来，我们要实现 *login*、*api* 和 *logout* 函数。*UserManager* 提供了一个 *signinRedirect* 来登录用户，和一个 *signoutRedirect* 来注销用户。我们在上面的代码中获得的 *User* 对象也有一个 *access_token* 属性，可用于对 Web API 进行身份验证。*access_token* 将通过带有 *Bearer* 方案的 *Authorization* 标头传递给 Web API。添加此代码以在我们的应用程序中实现这三个函数：

```

function login() {
    mgr.signinRedirect();
}

function api() {
    mgr.getUser().then(function (user) {
        var url = "https://localhost:6001/identity";

        var xhr = new XMLHttpRequest();
        xhr.open("GET", url);
        xhr.onload = function () {
            log(xhr.status, JSON.parse(xhr.responseText));
        }
        xhr.setRequestHeader("Authorization", "Bearer " + user.access_token);
        xhr.send();
    });
}

function logout() {
    mgr.signoutRedirect();
}

```

注解： 请参阅客户端凭据快速入门 以获取有关如何创建上述代码中使用的 api 的信息。

callback.html

一旦用户登录到 IdentityServer, 这个 HTML 文件就是指定的 redirect_uri 页面。它将完成与 IdentityServer 的 OpenID Connect 协议登录握手。这一切的代码都是由我们之前使用的 UserManager 类提供的。登录完成后, 我们可以将用户重定向回主要的 index.html 页面。添加此代码以完成登录过程:

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <script src="oidc-client.js"></script>
    <script>
        new Oidc.UserManager({response_mode:"query"}).signinRedirectCallback().
        ↪then(function() {
            window.location = "index.html";
        }).catch(function(e) {

```

(下页继续)

(续上页)

```
        console.error(e);
    });
</script>
</body>
</html>
```

12.6 为 JavaScript 客户端向 IdentityServer 添加客户端注册

现在客户端应用程序已准备就绪，我们需要在 IdentityServer 中为这个新的 JavaScript 客户端定义一个配置条目。在 IdentityServer 项目中找到客户端配置（在 *Config.cs* 中）。为我们的新 JavaScript 应用程序，向列表中添加一个新的 *Client*。它应该具有下面列出的配置：

```
// JavaScript 客户端
new Client
{
    ClientId = "js",
    ClientName = "JavaScript Client",
    AllowedGrantTypes = GrantTypes.Code,
    RequireClientSecret = false,

    RedirectUri = { "https://localhost:5003/callback.html" },
    PostLogoutRedirectUri = { "https://localhost:5003/index.html" },
    AllowedCorsOrigins = { "https://localhost:5003" },

    AllowedScopes =
    {
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile,
        "api1"
    }
}
```

12.7 使用 CORS 允许 Ajax 调用 Web API

必需的最后一点配置是在 Web API 项目中配置 CORS。这将允许从 *https://localhost:5003* 到 *https://localhost:6001* 的 Ajax 调用。

配置 CORS

在 *Startup.cs* 的 *ConfigureServices* 中将 CORS 服务添加到依赖注入系统中：

```
public void ConfigureServices(IServiceCollection services)
{
    // ...

    services.AddCors(options =>
    {
        // 这定义了一个名为 ``default`` 的 CORS 策略
        options.AddPolicy("default", policy =>
        {
            policy.WithOrigins("https://localhost:5003")
                .AllowAnyHeader()
                .AllowAnyMethod();
        });
    });
}
```

在 `Configure` 中将 `CORS` 中间件添加到管道中（在路由之后）：

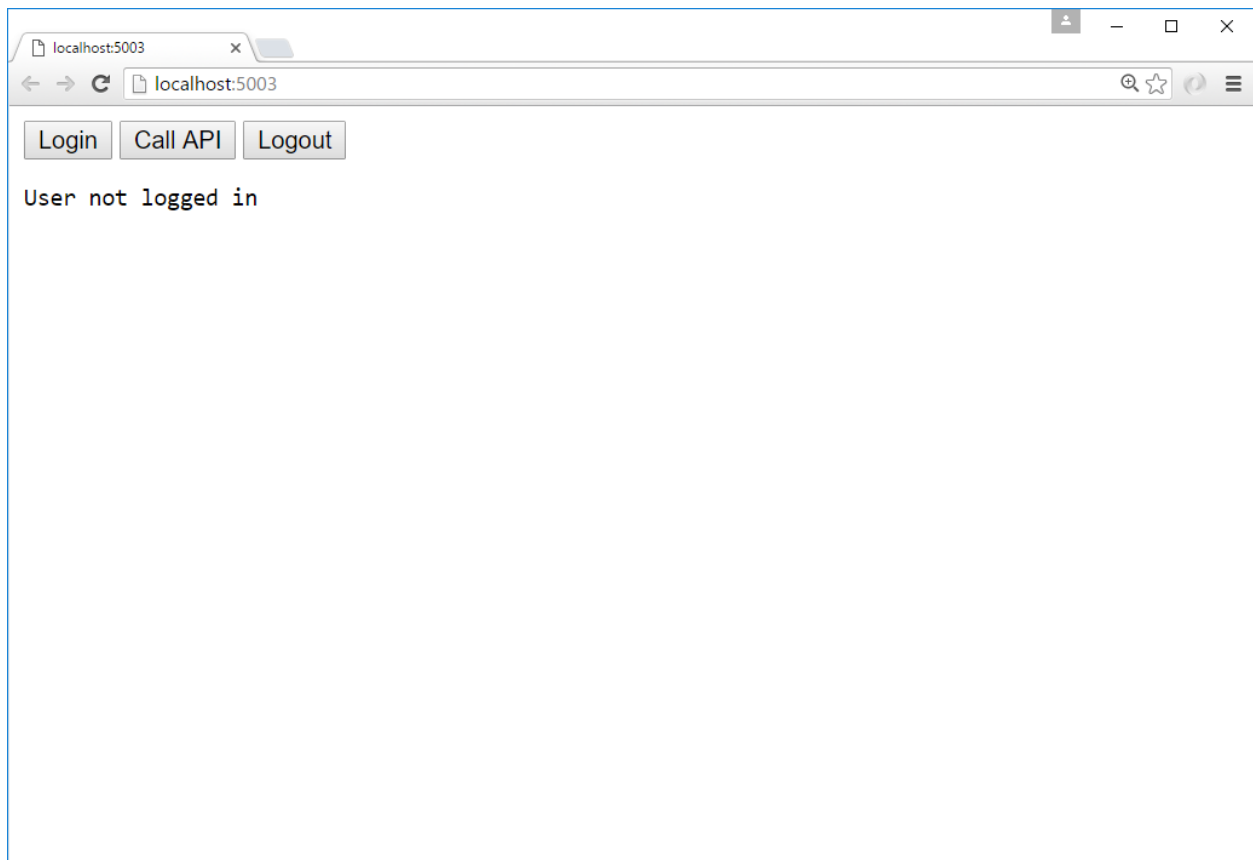
```
public void Configure(IApplicationBuilder app)
{
    app.UseRouting();

    app.UseCors("default");

    // ...
}
```

12.8 运行 JavaScript 应用程序

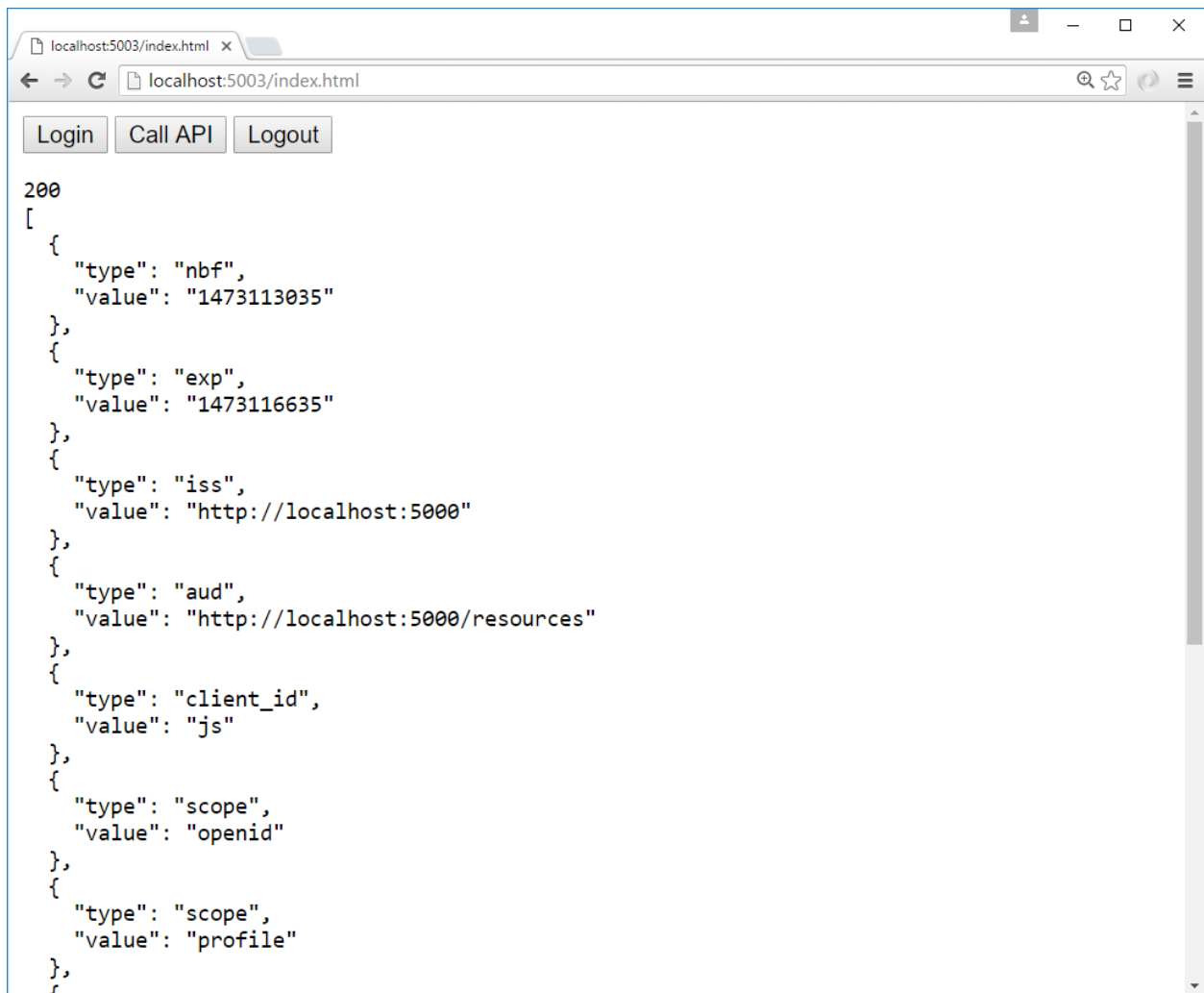
现在您应该能够运行 JavaScript 客户端应用程序：



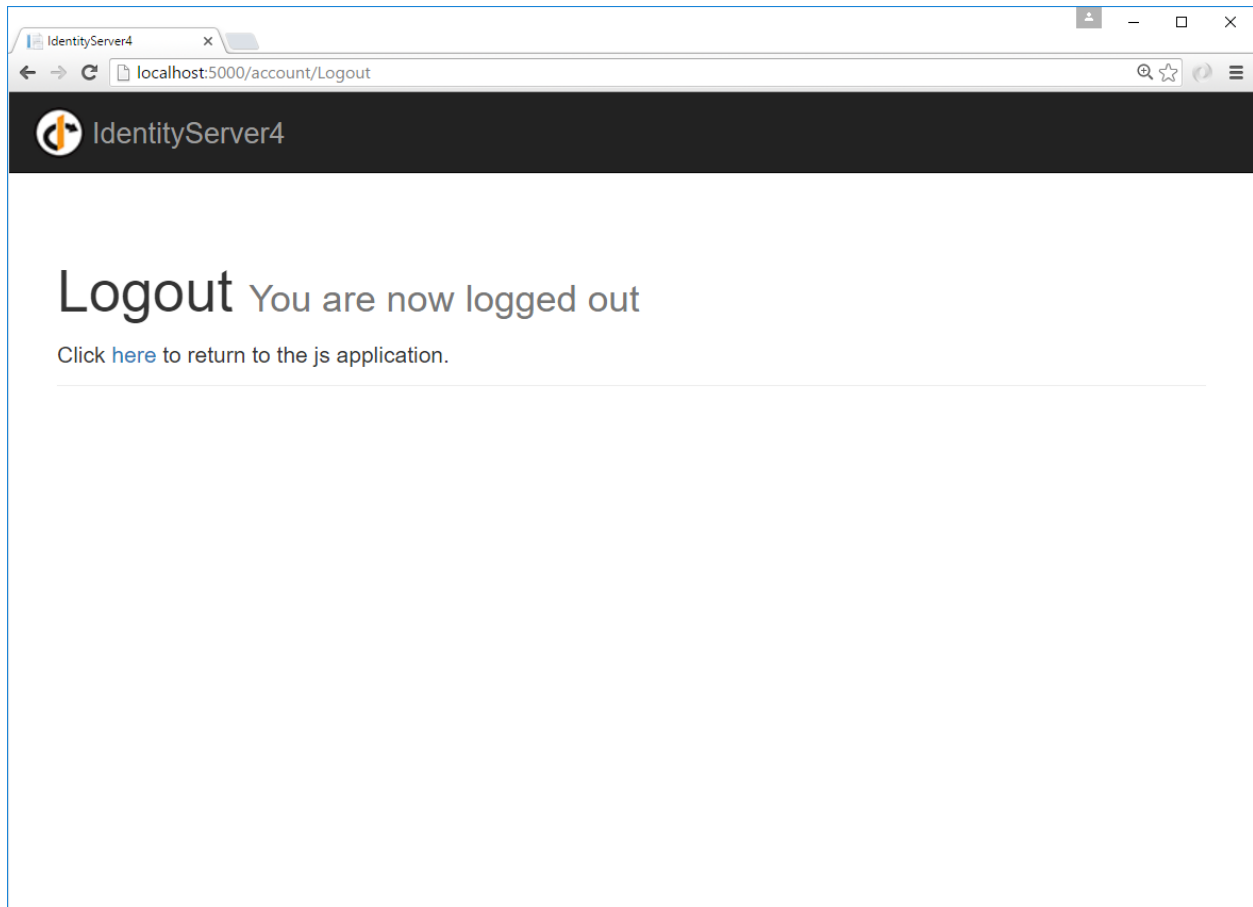
单击“Login”按钮以登录用户。一旦用户返回到 JavaScript 应用程序，您应该会看到他们的个人资料信息：



然后单击“API”按钮调用 Web API:



最后单击“Logout”将用户注销。



您现在已经启动了一个 JavaScript 客户端应用程序，该应用程序使用 IdentityServer 进行登录、注销和验证对 Web API 的调用。

使用 EntityFramework Core 获取配置和操作数据

在之前的快速入门中，我们在代码中创建了客户端和范围数据。在启动时，IdentityServer 将此配置数据加载到内存中。如果我们想修改这个配置数据，我们必须停止和启动 IdentityServer。

IdentityServer 还会生成临时数据，例如授权码、同意选择和刷新令牌。默认情况下，这些也存储在内存中。

要将这些数据移动到在重新启动和跨多个 IdentityServer 实例之间持久的数据库中，我们可以使用 IdentityServer4 Entity Framework 库。

注解：除了手动配置 EF 支持外，还有一个 IdentityServer 模板可以使用 `dotnet new is4ef` 创建一个支持 EF 的新项目。

13.1 IdentityServer4.EntityFramework

IdentityServer4.EntityFramework 使用以下 DbContext 实现所需的存储和服务：

- ConfigurationDbContext —— 用于配置数据，例如客户端、资源和范围
- PersistedGrantDbContext —— 用于临时操作数据，例如授权码和刷新令牌

这些 context 适用于任何 Entity Framework Core 兼容的关系数据库。

您可以在 IdentityServer4.EntityFramework.Storage nuget 包中找到这些 context、它们的实体以及使用它们的 IdentityServer4 存储。

您可以在 `IdentityServer4.EntityFramework` 中找到在您的 `IdentityServer` 中注册它们的扩展方法，我们现在将这样做：

```
dotnet add package IdentityServer4.EntityFramework
```

13.2 使用 SqlServer

对于本快速入门，我们将使用 Visual Studio 附带的 `SqlServer` 的 `LocalDb` 版本。要将 `SQL Server` 支持添加到我们的 `IdentityServer` 项目，您需要以下 `nuget` 包：

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

13.3 数据库架构更改和使用 EF 迁移

`IdentityServer4.EntityFramework.Storage` 包包含从 `IdentityServer` 模型映射的实体类。随着 `IdentityServer` 的模型发生变化，`IdentityServer4.EntityFramework.Storage` 中的实体类也会发生变化。当您使用 `IdentityServer4.EntityFramework.Storage` 并随着时间的推移升级时，您需要负责数据库架构以及随着实体类的更改而对该架构进行必要的更改。管理这些更改的一种方法是使用 **EF 迁移**，我们将在本快速入门中使用它。如果迁移不是您的偏好，那么您可以用任何您认为合适的方式管理架构更改。

注解： 您可以在 `IdentityServer4.EntityFramework.Storage` 存储库中找到 `SqlServer` 的 **最新 SQL 脚本**。

13.4 配置存储

要开始使用这些存储，您需要在 `Startup.cs` 的 `ConfigureServices` 方法中使用 `AddConfigurationStore` 和 `AddOperationalStore` 来替换对 `AddInMemoryClients`、`AddInMemoryIdentityResources`、`AddInMemoryApiScopes`、`AddInMemoryApiResources` 和 `AddInMemoryPersistedGrants` 的任何现有调用。

这些方法每个都需要一个 `DbContextOptionsBuilder`，这意味着你的代码看起来像这样：

```
var migrationsAssembly = typeof(Startup).GetTypeInfo().Assembly.GetName().Name;
const string connectionString = @"Data Source=(LocalDb)\MSSQLLocalDB;
→database=IdentityServer4.Quickstart.EntityFramework-4.0.0;trusted_connection=yes;";

services.AddIdentityServer()
    .AddTestUsers(TestUsers.Users)
    .AddConfigurationStore(options =>
```

(下页继续)

(续上页)

```
{
    options.ConfigureDbContext = b => b.UseSqlServer(connectionString,
        sql => sql.MigrationsAssembly(migrationsAssembly));
})
.AddOperationalStore(options =>
{
    options.ConfigureDbContext = b => b.UseSqlServer(connectionString,
        sql => sql.MigrationsAssembly(migrationsAssembly));
});
```

您可能需要将这些命名空间添加到文件中:

```
using Microsoft.EntityFrameworkCore;
using System.Reflection;
```

由于我们在本快速入门中使用 EF 迁移, 因此调用 `MigrationsAssembly` 用于通知 Entity Framework 宿主项目将包含迁移代码。这是必要的, 因为宿主项目位于与包含 `DbContext` 类的程序集中不同的程序集中。

13.5 添加迁移

一旦 IdentityServer 被配置为使用 Entity Framework, 我们需要生成一些迁移。

要创建迁移, 您需要在您的机器上安装 Entity Framework Core CLI, 和在 IdentityServer 中安装 Microsoft.EntityFrameworkCore.Design nuget 包:

```
dotnet tool install --global dotnet-ef
dotnet add package Microsoft.EntityFrameworkCore.Design
```

要创建迁移, 请在 IdentityServer 项目目录中打开命令提示符并运行以下两个命令:

```
dotnet ef migrations add InitialIdentityServerPersistedGrantDbMigration -c
↳ PersistedGrantDbContext -o Data/Migrations/IdentityServer/PersistedGrantDb
dotnet ef migrations add InitialIdentityServerConfigurationDbMigration -c
↳ ConfigurationDbContext -o Data/Migrations/IdentityServer/ConfigurationDb
```

您现在应该在项目中看到一个 `~/Data/Migrations/IdentityServer` 文件夹, 其中包含新创建的迁移的代码。

13.6 初始化数据库

现在我们有迁移，我们可以编写代码来从迁移创建数据库。我们还可以使用我们在之前的快速入门中已经定义的内存中配置数据为数据库种子。

注解： 本快速入门中使用的方法用于轻松启动和运行 IdentityServer。您应该设计适合您的架构的自己的数据库创建和维护策略。

在 *Startup.cs* 中添加这个方法来帮助初始化数据库：

```
private void InitializeDatabase(IApplicationBuilder app)
{
    using (var serviceScope = app.ApplicationServices.GetService<IServiceScopeFactory>
        .CreateScope())
    {
        serviceScope.ServiceProvider.GetRequiredService<PersistedGrantDbContext>().
            Database.Migrate();

        var context = serviceScope.ServiceProvider.GetRequiredService
            <ConfigurationDbContext>();
        context.Database.Migrate();
        if (!context.Clients.Any())
        {
            foreach (var client in Config.Clients)
            {
                context.Clients.Add(client.ToEntity());
            }
            context.SaveChanges();
        }

        if (!context.IdentityResources.Any())
        {
            foreach (var resource in Config.IdentityResources)
            {
                context.IdentityResources.Add(resource.ToEntity());
            }
            context.SaveChanges();
        }

        if (!context.ApiScopes.Any())
        {
            foreach (var resource in Config.ApiScopes)
            {
```

(下页继续)

(续上页)

```
        context.ApiScopes.Add(resource.ToEntity());  
    }  
    context.SaveChanges();  
}  
}
```

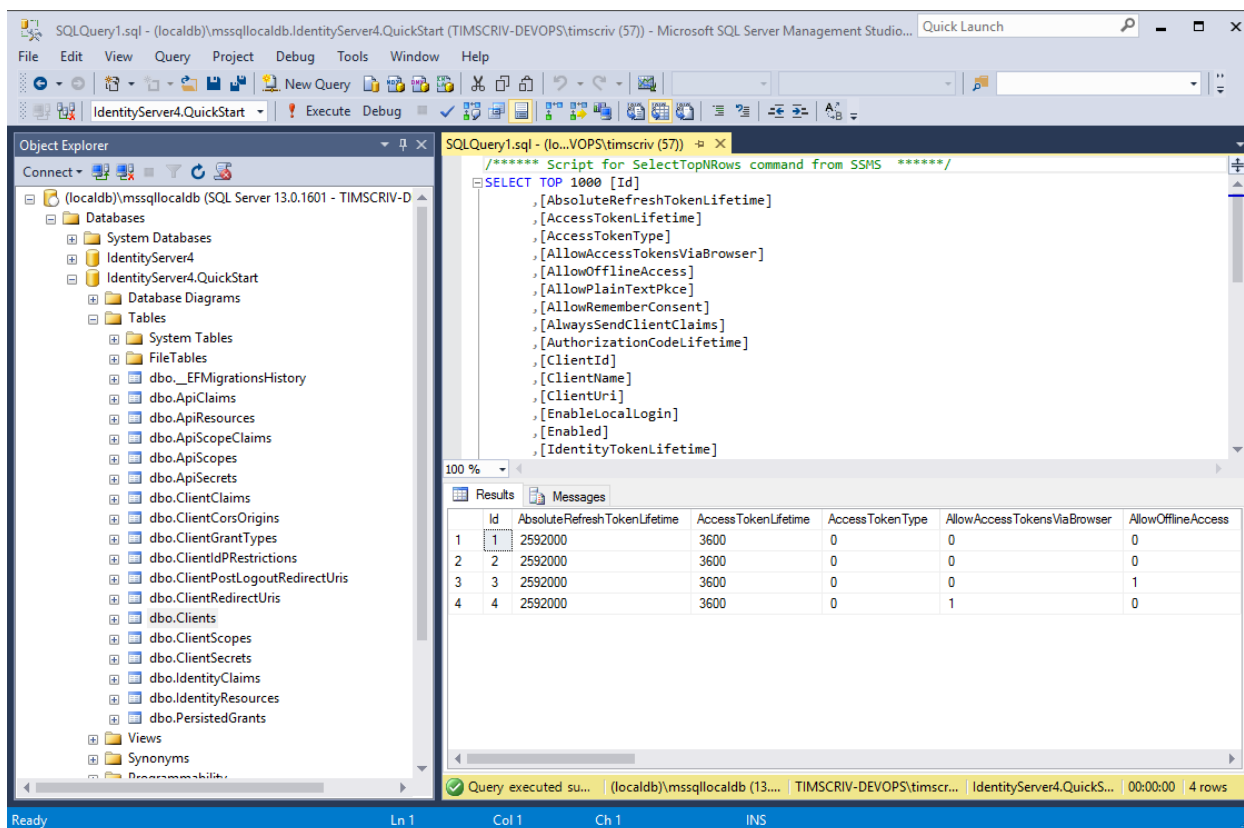
上面的代码可能需要您将以下命名空间添加到您的文件中:

```
using System.Linq;  
using IdentityServer4.EntityFramework.DbContexts;  
using IdentityServer4.EntityFramework.Mappers;
```

然后我们可以从 `Configure` 方法中调用它:

```
public void Configure(IApplicationBuilder app)  
{  
    // 这将进行初始数据库填充  
    InitializeDatabase(app);  
  
    // 已经在这里的其余代码  
    // ...  
}
```

现在, 如果您运行 `IdentityServer` 项目, 则应创建数据库, 并使用快速入门配置数据作为种子。您应该能够使用 `SQL Server Management Studio` 或 `Visual Studio` 来连接和检查数据。



注解： 上面的 InitializeDatabase 辅助 API 可以方便地为数据库设置种子，但是这种方法不适合每次应用程序运行时都执行。填充数据库后，请考虑删除对 API 的调用。

13.7 运行客户端应用程序

您现在应该能够运行任何现有的客户端应用程序，并进行登录、获取令牌和调用 API ——所有这些都基于数据库配置。

使用 ASP.NET Core Identity

注解： 对于任何先决条件（例如模板），请先查看概述。

IdentityServer 旨在提供灵活性，其中一部分是允许您为您的用户及其数据（包括密码）使用您想使用的任何数据库。如果您开始使用新的用户数据库，则 ASP.NET Core Identity 是您可以选择的一个选择。本快速入门展示了如何将 ASP.NET Core Identity 与 IdentityServer 结合使用。

本快速入门使用 ASP.NET Core Identity 的方法是为 IdentityServer 主机创建一个新项目。这个新项目将取代我们在之前的快速入门中建立的先前 IdentityServer 项目。这个新项目的原因是由于使用 ASP.NET Core Identity 时 UI 资产的差异（主要围绕登录和注销的差异）。此解决方案中的所有其他项目（针对客户端和 API）将保持不变。

注解： 本快速入门假设您熟悉 ASP.NET Core Identity 的工作原理。如果不是，建议先 [了解一下](#)。

14.1 新的 ASP.NET Core Identity 项目

第一步是将 ASP.NET Core Identity 的新项目添加到您的解决方案中。我们提供了一个模板，其中包含 ASP.NET Core Identity 和 IdentityServer 所需的最少 UI 资产。您最终将删除 IdentityServer 的旧项目，但您需要迁移一些项目。

首先创建一个将使用 ASP.NET Core Identity 的新 IdentityServer 项目：

```
cd quickstart/src
dotnet new is4aspid -n IdentityServerAspNetIdentity
```

当提示“seed”用户数据库时，选择“Y”作为“yes”。这将使用我们的“alice”和“bob”用户填充用户数据库。他们的密码是“Pass123\$”。

注解：该模板使用 Sqlite 作为用户的数据库，并且在模板中预先创建了 EF 迁移。如果您希望使用不同的数据库提供程序，则需要更改代码中使用的提供程序并重新创建 EF 迁移。

14.2 检查新项目

在您选择的编辑器中打开新项目，并检查生成的代码。请务必查看：

14.2.1 IdentityServerAspNetIdentity.csproj

注意对 *IdentityServer4.AspNetIdentity* 的引用。此 NuGet 包包含 IdentityServer 的 ASP.NET Core Identity 集成组件。

14.2.2 Startup.cs

在 ‘ConfigureServices’ 中注意必要的 `AddDbContext<ApplicationDbContext>` 和 `AddIdentity<ApplicationUser, IdentityRole>` 调用来配置 ASP.NET Core Identity。

另请注意，您在之前的快速入门中所做的大部分相同 IdentityServer 配置已经完成。该模板对客户端和资源使用内存中的样式，它们来自 *Config.cs*。

最后，注意添加了对 `AddAspNetIdentity<ApplicationUser>` 的新调用。`AddAspNetIdentity` 添加了集成层以允许 IdentityServer 访问 ASP.NET Core Identity 用户数据库的用户数据。当 IdentityServer 必须将用户的声明添加到令牌中时，这是必需的。

注意 `AddIdentity<ApplicationUser, IdentityRole>` 必须在 `AddIdentityServer` 之前调用。

14.2.3 Config.cs

Config.cs 包含硬编码的内存客户端和资源定义。为了让相同的客户端和 API 像之前的快速入门一样工作，我们需要将旧 IdentityServer 项目中的配置数据复制到这个项目中。现在这样做，之后 *Config.cs* 应该是这样的：

```
public static class Config
{
    public static IEnumerable<IdentityResource> IdentityResources =>
```

(下页继续)

(续上页)

```

new List<IdentityResource>
{
    new IdentityResources.OpenId(),
    new IdentityResources.Profile(),
};

public static IEnumerable<ApiScope> ApiScopes =>
new List<ApiScope>
{
    new ApiScope("api1", "My API")
};

public static IEnumerable<Client> Clients =>
new List<Client>
{
    // 机器对机器客户端
    new Client
    {
        ClientId = "client",
        ClientSecrets = { new Secret("secret".Sha256()) },

        AllowedGrantTypes = GrantTypes.ClientCredentials,
        // 客户端有权访问的范围
        AllowedScopes = { "api1" }
    },

    // interactive ASP.NET Core MVC client
    new Client
    {
        ClientId = "mvc",
        ClientSecrets = { new Secret("secret".Sha256()) },

        AllowedGrantTypes = GrantTypes.Code,

        // 登录后重定向到哪里
        RedirectUri = { "https://localhost:5002/signin-oidc" },

        // 注销后重定向到哪里
        PostLogoutRedirectUri = { "https://localhost:5002/signout-callback-
oidc" },

        AllowedScopes = new List<string>
        {

```

(下页继续)

(续上页)

```
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile,
        "api1"
    }
}
};
}
```

此时，您不再需要旧的 IdentityServer 项目。

14.2.4 Program.cs 和 SeedData.cs

Program.cs 的 Main 与大多数 ASP.NET Core 项目略有不同。请注意这如何查找名为 */seed* 的命令行参数，该参数用作在 ASP.NET Core Identity 数据库中为用户提供种子的标志。

查看 SeedData 类代码以了解如何创建数据库以及如何创建第一个用户。

14.2.5 AccountController

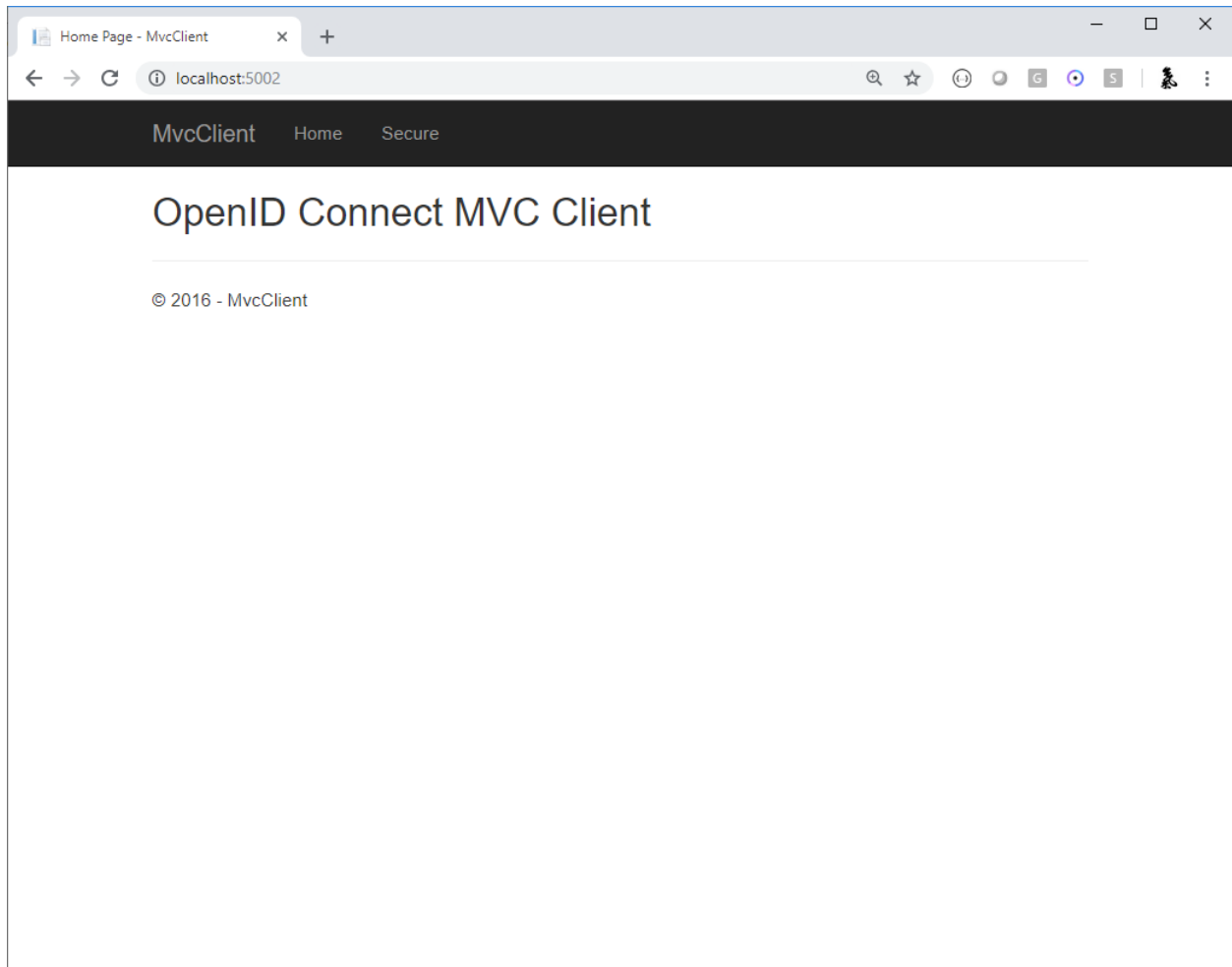
此模板中要检查的最后一个代码是 AccountController。这包含与之前的快速入门和模板略有不同的登录和注销代码。请注意使用 ASP.NET Core Identity 中的 SignInManager<ApplicationUser> 和 UserManager<ApplicationUser> 来验证凭据和管理身份验证会话。

其余大部分代码与之前的快速入门和模板相同。

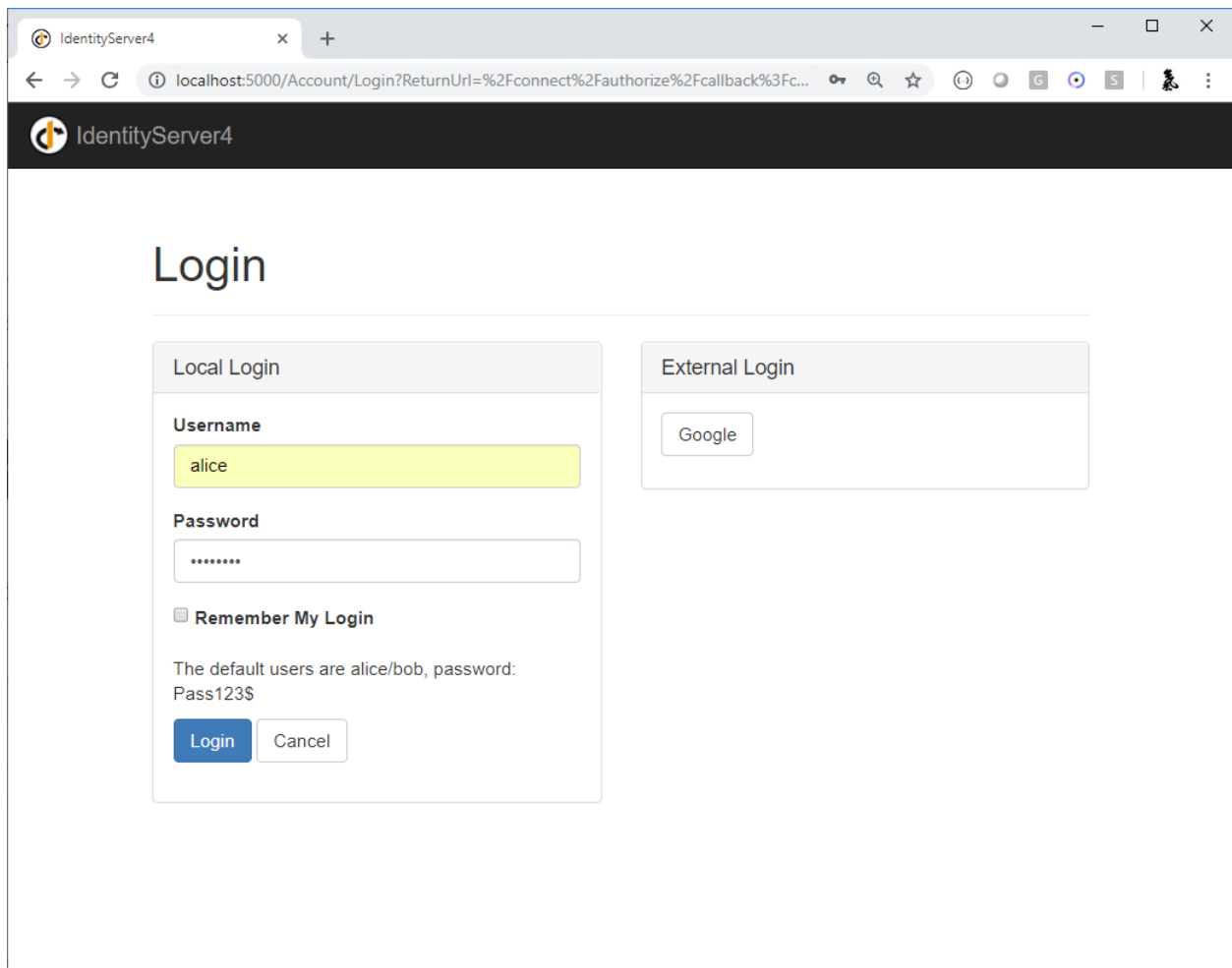
14.3 使用 MVC 客户端登录

此时，您应该能够运行所有现有的客户端和示例。一个例外是 *ResourceOwnerClient* ——密码需要从 password 更新为 Pass123\$。

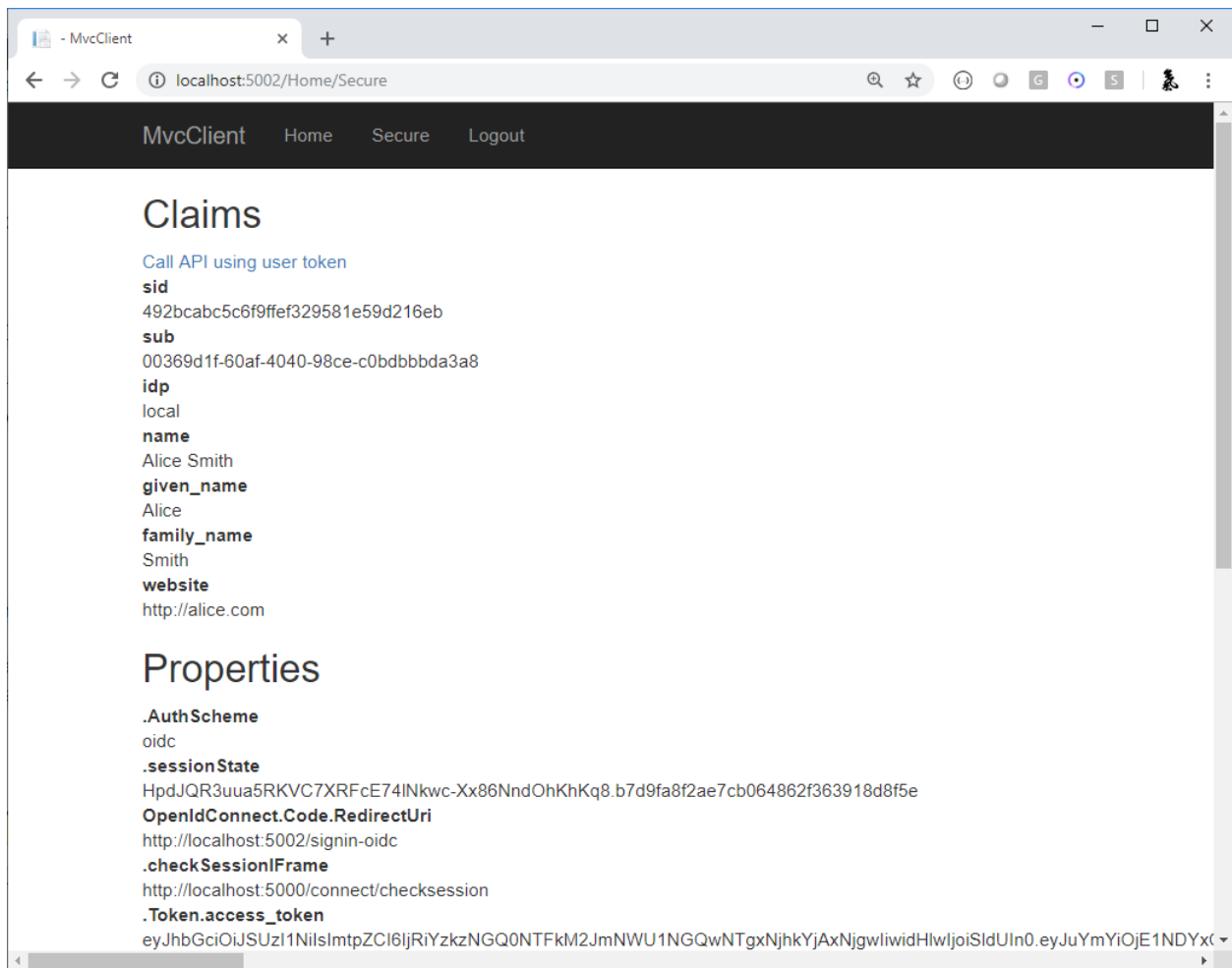
启动 MVC 客户端应用程序，您应该能够单击“安全”链接以登录。



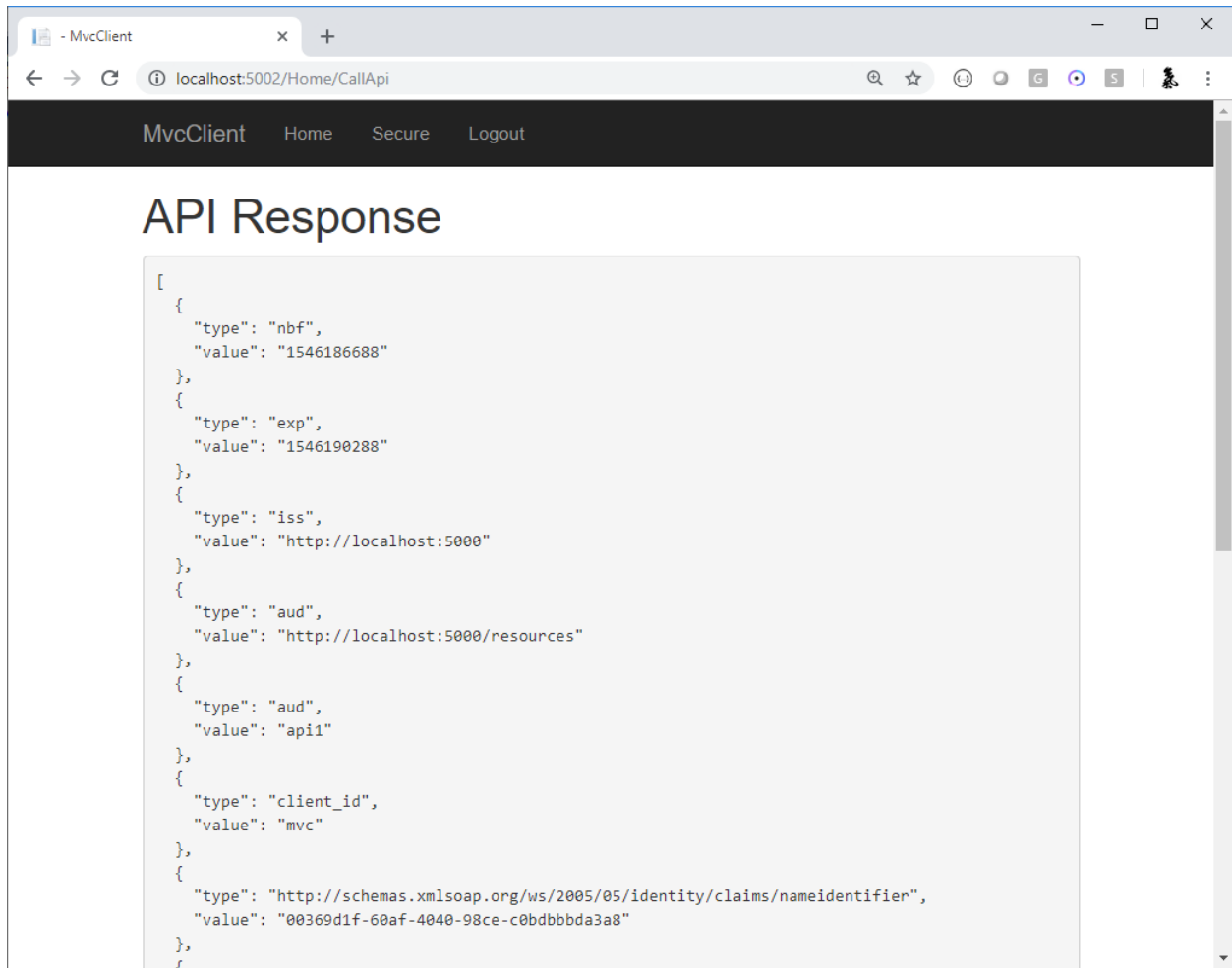
您应该被重定向到 ASP.NET Core Identity 登录页面。使用您新创建的用户登录：



登录后，您会看到正常的同意页面。同意后，您将被重定向回 MVC 客户端应用程序，您的用户声明应在其中列出。



您还应该能够点击 “Call API using application identity” 以代表用户调用 API:



现在，您正在 IdentityServer 中使用来自 ASP.NET Core Identity 的用户。

14.4 缺少了什么？

此模板中的大部分其余代码与我们提供的其他快速入门和模板类似。您会注意到此模板中缺少的一件事是用于用户注册、密码重置的 UI 代码，以及您可能期望从 Visual Studio ASP.NET Core Identity 模板中获得的其他内容。

鉴于使用 ASP.NET Core Identity 的各种要求和不同方法，我们的模板故意不提供这些功能。您应该了解 ASP.NET Core Identity 如何工作得足够好，以便将这些功能添加到您的项目中。或者，您可以基于 Visual Studio ASP.NET Core Identity 模板创建一个新项目，并将您在这些快速入门中了解的 IdentityServer 功能添加到该项目。

IdentityServer 是中间件和服务的组合。所有配置都在您的启动类中完成。

15.1 配置服务

您可以通过调用将 IdentityServer 服务添加到 DI 系统:

```
public void ConfigureServices(IServiceCollection services)
{
    var builder = services.AddIdentityServer();
}
```

您可以选择将选项传递到此调用中。有关选项的详细信息,请参阅[这里](#)。

这将返回一个 `builder` 对象,该对象又具有许多连接附加服务的便捷方法。

15.2 密钥材料

IdentityServer 支持 X.509 证书(原始文件和对 Windows 证书存储的引用)、RSA 密钥和用于令牌签名和验证的 EC 密钥。每个密钥都可以配置一个(兼容的)签名算法,例如 RS256、RS384、RS512、PS256、PS384、PS512、ES256、ES384 或 ES512。

您可以使用以下方法配置密钥材料:

- **AddSigningCredential** 添加一个签名密钥,为各种令牌创建/验证服务提供指定的密钥材料。

- **AddDeveloperSigningCredential** 在启动时创建临时密钥材料。这是针对开发场景的。默认情况下，生成的密钥将持久保存在本地目录中。
- **AddValidationKey** 添加用于验证令牌的密钥。它们将由内部令牌验证器使用，并将显示在发现文档中。

15.3 内存配置存储

各种“in-memory”配置 API 允许从内存中的配置对象列表配置 IdentityServer。这些“in-memory”集合可以在托管应用程序中进行硬编码，也可以从配置文件或数据库中动态加载。但是，按照设计，这些集合仅在托管应用程序启动时创建。

这些配置 API 的使用旨在用于原型设计、开发和/或测试，其中不需要在运行时动态查询数据库以获取配置数据。如果配置很少更改，或者在必须更改值时要求重新启动应用程序并不方便，则这种配置样式也可能适用于生产场景。

- **AddInMemoryClients** 基于 Client 配置对象的内存集合注册 IClientStore 和 ICorsPolicyService 实现。
- **AddInMemoryIdentityResources** 基于 IdentityResource 配置对象的内存集合注册 IResourceStore 实现。
- **AddInMemoryApiScopes** 基于 ApiScope 配置对象的内存中集合注册 IResourceStore 实现。
- **AddInMemoryApiResources** 基于 ApiResource 配置对象的内存集合注册 IResourceStore 实现。

15.4 测试存储

TestUser 类在 IdentityServer 中对用户、他们的凭据和声明进行建模。TestUser 的使用类似于“in-memory”存储的使用，因为它用于原型设计、开发和/或测试。不建议在生产中使用 TestUser。

- **AddTestUsers** 基于 TestUser 对象的集合注册 TestUserStore。TestUserStore 由默认的快速入门 UI 使用。还注册了 IProfileService 和 IResourceOwnerPasswordValidator 的实现。

15.5 额外服务

- **AddExtensionGrantValidator** 添加用于扩展授权的 IExtensionGrantValidator 实现。
- **AddSecretParser** 添加用于解析客户端或 API 资源凭据的 ISecretParser 实现。
- **AddSecretValidator** 添加 ISecretValidator 实现，用于根据凭证存储验证客户端或 API 资源凭证。

- **AddResourceOwnerValidator** 添加 `IResourceOwnerPasswordValidator` 实现，用于验证资源所有者密码凭据授权类型的用户凭据。
- **AddProfileService** 添加 `IProfileService` 实现以连接到您的自定义用户配置文件存储。`DefaultProfileService` 类提供了默认实现，该实现依赖于身份验证 cookie 作为在令牌中发布的唯一声明来源。
- **AddAuthorizeInteractionResponseGenerator** 添加 `IAuthorizeInteractionResponseGenerator` 实现，以自定义授权端点的逻辑，以便在必须向用户显示错误、登录、同意或任何其他自定义页面的 UI 时使用。`AuthorizeInteractionResponseGenerator` 类提供了默认实现，因此如果您需要增强现有行为，请考虑从该现有类派生。
- **AddCustomAuthorizeRequestValidator** 添加 `ICustomAuthorizeRequestValidator` 实现，以在授权端点自定义请求参数验证。
- **AddCustomTokenRequestValidator** 添加 `ICustomTokenRequestValidator` 实现，以自定义令牌端点处的请求参数验证。
- **AddRedirectUriValidator** 添加 `IRedirectUriValidator` 实现，以自定义重定向 URI 验证。
- **AddAppAuthRedirectUriValidator** 添加符合“AppAuth”（原生应用程序的 OAuth 2.0）的重定向 URI 验证器（进行严格验证，但也允许使用随机端口的 <http://127.0.0.1>）。
- **AddJwtBearerClientAuthentication** 添加对使用 JWT bearer 断言的客户端身份验证的支持。
- **AddMutualTlsSecretValidators** 为 mutual TLS 添加 X509 密钥验证器。

15.6 缓存

IdentityServer 经常使用客户端和资源数据。如果这些数据是从数据库或其他外部存储加载的，那么频繁地重新加载相同的数据可能会很昂贵。

- **AddInMemoryCaching** 要使用下面描述的任何缓存，必须在 DI 中注册 `ICache<T>` 的实现。此 API 注册了一个基于 ASP.NET Core 的 `MemoryCache` 的默认内存中实现 `ICache<T>`。
- **AddClientStoreCache** 注册一个 `IClientStore` 装饰器实现，它将维护一个 `Client` 配置对象的内存缓存。缓存持续时间可在 `IdentityServerOptions` 的 `Caching` 配置选项中配置。
- **AddResourceStoreCache** 注册一个 `IResourceStore` 装饰器实现，它将维护一个 `IdentityResource` 和 `ApiResource` 配置对象的内存缓存。缓存持续时间可在 `IdentityServerOptions` 的 `Caching` 配置选项中配置。
- **AddCorsPolicyCache** 注册一个 `ICorsPolicyService` 装饰器实现，它将维护 CORS 策略服务评估结果的内存缓存。缓存持续时间可在 `IdentityServerOptions` 的 `Caching` 配置选项中配置。

可以进一步自定义缓存：

默认缓存依赖于 `ICache<T>` 实现。如果您希望为特定配置对象自定义缓存行为，您可以在依赖注入系统中替换此实现。

`ICache<T>` 本身的默认实现依赖于 .NET 提供的 `IMemoryCache` 接口（和 `MemoryCache` 实现）。如果您希望自定义内存缓存行为，您可以替换依赖注入系统中的 `IMemoryCache` 实现。

15.7 配置管道

您需要通过调用将 `IdentityServer` 添加到管道中：

```
public void Configure(IApplicationBuilder app)
{
    app.UseIdentityServer();
}
```

注解： `UseIdentityServer` 包括对 `UseAuthentication` 的调用，因此不必同时具有这两个调用。

中间件没有额外的配置。

请注意，顺序在管道中很重要。例如，您需要在实现登录屏幕的 UI 框架之前添加 `IdentitySever`。

定义资源

OpenID Connect/OAuth 令牌服务的最终工作是控制对资源的访问。

IdentityServer 中的两种基本资源类型是：

- **identity resources:** 代表关于用户的声明，如用户 ID、显示名称、电子邮件地址等……
- **API resources:** 代表客户想要访问的功能。通常，它们是基于 HTTP 的端点（又名 API），但也可以是消息队列端点或类似端点。

注解： 您可以使用 C# 对象模型定义资源——或从数据存储加载它们。IResourceStore 的实现处理这些低级细节。对于本文档，我们使用内存中实现。

16.1 身份资源

身份资源是一组命名的声明，可以使用 *scope* 参数进行请求。

OpenID Connect 规范 [建议](#) 了几个标准范围名称来声明类型映射，它们可能对您的灵感有用，但您可以自由地自己设计它们。

其中之一实际上是强制性的，*openid* 范围，它告诉提供者在身份令牌中返回 ***sub***（subject id）声明。

这是您在代码中定义 *openid* 范围的方式：

```
public static IEnumerable<IdentityResource> GetIdentityResources()
{
    return new List<IdentityResource>
    {
        new IdentityResource(
            name: "openid",
            userClaims: new[] { "sub" },
            displayName: "您的用户标识符")
    };
}
```

但由于这是规范中的标准范围之一，您可以将其缩短为：

```
public static IEnumerable<IdentityResource> GetIdentityResources()
{
    return new List<IdentityResource>
    {
        new IdentityResources.OpenId()
    };
}
```

注解：有关 IdentityResource 的更多信息，请参阅参考部分。

以下示例显示了一个名为 *profile* 的自定义身份资源，它表示显示名称、电子邮件地址和网站声明：

```
public static IEnumerable<IdentityResource> GetIdentityResources()
{
    return new List<IdentityResource>
    {
        new IdentityResource(
            name: "profile",
            userClaims: new[] { "name", "email", "website" },
            displayName: "您的个人资料数据")
    };
}
```

定义资源后，您可以通过 AllowedScopes 选项（省略其他属性）将其访问权限授予客户端：

```
var client = new Client
{
    ClientId = "client",

    AllowedScopes = { "openid", "profile" }
```

(下页继续)

(续上页)

```
};
```

然后客户端可以使用 `scope` 参数请求资源（其他参数省略）：

```
https://demo.identityserver.io/connect/authorize?client_id=client&scope=openid profile
```

IdentityServer 然后将使用 `scope` 名称创建请求的声明类型列表，并将其呈现给您的 *profile service* 的实现。

16.2 APIs

设计您的 API 表面可能是一项复杂的任务。IdentityServer 提供了一些原语来帮助您解决这个问题。

原始的 OAuth 2.0 规范中有 `scope` 的概念，它只是定义为客户端请求的访问范围。从技术上讲，`scope` 参数是一个空格分隔值列表——您需要提供它的结构和语义。

在更复杂的系统中，通常会引入 *resource* 的概念。这可能是例如物理或逻辑 API。反过来，每个 API 也可能具有 `scope`。某些 `scope` 可能是该资源独有的，而某些 `scope` 可能是共享的。

让我们首先从简单的 `scopes` 开始，然后我们将看看 `resources` 如何帮助构建 `scopes`。

16.2.1 Scopes

让我们对一些非常简单的东西进行建模——一个具有三个逻辑操作 *read*、*write* 和 *delete* 的系统。

您可以使用 `ApiScope` 类定义它们：

```
public static IEnumerable<ApiScope> GetApiScopes()
{
    return new List<ApiScope>
    {
        new ApiScope(name: "read",    displayName: "读取您的数据。"),
        new ApiScope(name: "write",   displayName: "写入您的数据。"),
        new ApiScope(name: "delete",  displayName: "删除您的数据。")
    };
}
```

然后，您可以将 `scopes` 分配给各种客户端，例如：

```
var webViewer = new Client
{
    ClientId = "web_viewer",

    AllowedScopes = { "openid", "profile", "read" }
};
```

(下页继续)

(续上页)

```
var mobileApp = new Client
{
    ClientId = "mobile_app",

    AllowedScopes = { "openid", "profile", "read", "write", "delete" }
}
```

16.2.2 基于 Scopes 的授权

当客户端请求 scope（并且该 scope 是通过配置允许的，而不是通过同意拒绝）时，该 scope 的值将作为 *scope* 类型的声明包含在结果访问令牌中（对于 JWT 和自省），例如：

```
{
    "typ": "at+jwt"
}.
{
    "client_id": "mobile_app",
    "sub": "123",

    "scope": "read write delete"
}
```

访问令牌的使用者可以使用该数据来确保实际上允许客户端调用相应的功能。

注解： 请注意，scope 仅用于授权客户端——而不是用户。IOW ——*write scope* 允许客户端调用与之相关的功能。但是，该客户端很可能只能写入属于当前用户的数据。这个额外的以用户为中心的授权是应用程序逻辑，不包含在 OAuth 中。

您可以通过从 scope 请求派生其他声明来添加有关用户的更多信息。以下 scope 定义告诉配置系统，当授予 *write scope* 时，应将 *user_level* 声明添加到访问令牌：

```
var writeScope = new ApiScope(
    name: "write",
    displayName: "写入您的数据。",
    userClaims: new[] { "user_level" });
```

这会将 *user_level* 声明作为请求的声明类型传递给 profile service，以便访问令牌的使用者可以使用此数据作为授权决策或业务逻辑的输入。

注解： 使用仅 scope 模型时，不会向令牌添加任何 aud（受众）声明，因为此概念不适用。如果您需要 aud 声

明，您可以在选项上启用 `EmitStaticAudience` 设置。这将以 `issuer_name/resources` 格式发出 `aud` 声明。如果您需要更多地控制 `aud` 声明，请使用 `API resources`。

16.2.3 参数化 Scopes

有时 `scopes` 具有特定的结构，例如，`scope` 名称带有附加参数：`transaction:id` 或 `read_patient:patientid`。

在这种情况下，您将创建一个没有参数部分的 `scope` 并将该名称分配给客户端，但另外提供一些逻辑来在运行时使用 `IScopeParser` 接口或通过从我们的默认实现派生来解析 `scope` 的结构，例如：

```
public class ParameterizedScopeParser : DefaultScopeParser
{
    public ParameterizedScopeParser(ILogger<DefaultScopeParser> logger) : base(logger)
    {
    }

    public override void ParseScopeValue(ParseScopeContext scopeContext)
    {
        const string transactionScopeName = "transaction";
        const string separator = ":";
        const string transactionScopePrefix = transactionScopeName + separator;

        var scopeValue = scopeContext.RawValue;

        if (scopeValue.StartsWith(transactionScopePrefix))
        {
            // 进入这里的 scope 类似于 "transaction:something"
            var parts = scopeValue.Split(separator, StringSplitOptions.
↵RemoveEmptyEntries);
            if (parts.Length == 2)
            {
                scopeContext.SetParsedValues(transactionScopeName, parts[1]);
            }
            else
            {
                scopeContext.SetError("transaction scope 缺少 transaction 参数值");
            }
        }
        else if (scopeValue != transactionScopeName)
        {
            // 进入这里的 scope 不像 "transaction"
            base.ParseScopeValue(scopeContext);
        }
    }
}
```

(下页继续)

(续上页)

```

        else
        {
            // 进入这里的 scope
            ↳恰好是 "transaction"，也就是说我们忽略了它并且不将它包含在结果中
            scopeContext.SetIgnore();
        }
    }
}

```

然后，您可以访问整个管道中的解析值，例如，在配置文件服务中：

```

public class HostProfileService : IProfileService
{
    public override async Task GetProfileDataAsync(ProfileDataRequestContext context)
    {
        var transaction = context.RequestedResources.ParsedScopes.FirstOrDefault(x =>
            ↳x.ParsedName == "transaction");
        if (transaction?.ParsedParameter != null)
        {
            context.IssuedClaims.Add(new Claim("transaction_id", transaction.
                ↳ParsedParameter));
        }
    }
}

```

16.2.4 API Resources

当 API 表面变得更大时，像上面使用的那样一个简单的 scope 列表可能是不可行的。

您通常需要引入某种命名空间来组织 scope 名称，也许您还想将它们组合在一起并获得一些更高级别的结构，例如访问令牌中的 *audience* 声明。您可能还会遇到多个资源应支持相同 scope 名称的情况，而有时您明确希望将 scope 与特定资源隔离。

在 IdentityServer 中，ApiResource 类允许一些额外的组织。让我们使用以下 scope 定义：

```

public static IEnumerable<ApiScope> GetApiScopes()
{
    return new List<ApiScope>
    {
        // 发票 API 特有的 scopes
        new ApiScope(name: "invoice.read",    displayName: "读取您的发票。"),
        new ApiScope(name: "invoice.pay",     displayName: "支付您的发票。"),
    }
}

```

(下页继续)

(续上页)

```
// 客户 API 特有的 scopes
new ApiScope(name: "customer.read",    displayName: "读取您的客户信息。"),
new ApiScope(name: "customer.contact", displayName: "允许联系您的客户之一。"),

// 共享 scope
new ApiScope(name: "manage", displayName: "提供对发票和客户数据的管理访问。")
};
}
```

使用 `ApiResource`，您现在可以创建两个逻辑 API 及其对应的 `scope`:

```
public static readonly IEnumerable<ApiResource> GetApiResources()
{
    return new List<ApiResource>
    {
        new ApiResource("invoice", "发票 API")
        {
            Scopes = { "invoice.read", "invoice.pay", "manage" }
        },

        new ApiResource("customer", "客户 API")
        {
            Scopes = { "customer.read", "customer.contact", "manage" }
        }
    };
}
```

使用 API 资源分组可以为您提供以下附加功能

- 支持 JWT *aud* 声明。*audience* 声明的值将是 API 资源的名称
- 支持在所有包含的 `scopes` 内添加通用用户声明
- 通过为资源分配 API 密钥来支持自省
- 支持为资源配置访问令牌签名算法

让我们看一下上述资源配置的一些示例访问令牌。

客户端请求 `invoice.read` 和 `invoice.pay`:

```
{
    "typ": "at+jwt"
}.
{
    "client_id": "client",
    "sub": "123",
```

(下页继续)

(续上页)

```
"aud": "invoice",  
"scope": "invoice.read invoice.pay"  
}
```

客户端请求 invoice.read 和 customer.read:

```
{  
  "typ": "at+jwt"  
}.  
{  
  "client_id": "client",  
  "sub": "123",  
  
  "aud": [ "invoice", "customer" ]  
  "scope": "invoice.read customer.read"  
}
```

客户端请求 manage:

```
{  
  "typ": "at+jwt"  
}.  
{  
  "client_id": "client",  
  "sub": "123",  
  
  "aud": [ "invoice", "customer" ]  
  "scope": "manage"  
}
```

16.2.5 迁移到 v4 的步骤

如上所述，从 v4 开始，scope 有自己的定义，并且可以选择被 resource 引用。在 v4 之前，scopes 始终包含在 resource 中。

要迁移到 v4，您需要拆分 scope 和 resource 注册，通常首先注册所有 scope（例如，使用 AddInMemoryApiScopes 方法），然后注册 API resources（如果有）。然后 API resources 将按名称引用先前注册的 scope。

CHAPTER 17

定义客户端

客户端表示可以从您的 `IdentityServer` 请求令牌的应用程序。

详细信息各不相同，但您通常会为客户端定义以下通用设置：

- 唯一的客户端 ID
- 密钥（如果需要的话）
- 允许与令牌服务的交互（称为授权类型）
- 身份和/或访问令牌被发送到的网络位置（称为重定向 URI）
- 允许客户端访问的 `scopes`（又名 `resources`）列表

注解：在运行时，通过 `IClientStore` 的实现来检索客户端。这允许从任意数据源（如配置文件或数据库）加载它们。对于本文档，我们将使用客户端存储的内存版本。您可以通过 `AddInMemoryClients` 扩展方法在 `ConfigureServices` 中连接内存存储。

17.1 定义服务器到服务器通信的客户端

在这种情况下，没有交互式用户存在——服务（又名客户端）想要与 API（又名 scope）通信：

```
public class Clients
{
    public static IEnumerable<Client> Get()
    {
        return new List<Client>
        {
            new Client
            {
                ClientId = "service.client",
                ClientSecrets = { new Secret("secret".Sha256()) },

                AllowedGrantTypes = GrantTypes.ClientCredentials,
                AllowedScopes = { "api1", "api2.read_only" }
            }
        };
    }
}
```

17.2 定义用于使用身份验证和委托 API 访问的交互式应用程序

交互式应用程序（例如 Web 应用程序或原生桌面/移动）应用程序使用授权码流程。此流程为您提供最佳安全性，因为访问令牌仅通过后端渠道调用传输（并允许您访问刷新令牌）：

```
var interactiveClient = new Client
{
    ClientId = "interactive",

    AllowedGrantTypes = GrantTypes.Code,
    AllowOfflineAccess = true,
    ClientSecrets = { new Secret("secret".Sha256()) },

    RedirectUri = { "http://localhost:21402/signin-oidc" },
    PostLogoutRedirectUri = { "http://localhost:21402/" },
    FrontChannelLogoutUri = "http://localhost:21402/signout-oidc",

    AllowedScopes =
    {
        IdentityServerConstants.StandardScopes.OpenId,
```

(下页继续)

(续上页)

```
IdentityServerConstants.StandardScopes.Profile,
IdentityServerConstants.StandardScopes.Email,

    "api1", "api2.read_only"
},
};
```

注解: 请参阅 *grant types* 主题，了解有关为您的客户端选择正确的授权类型的更多信息。

17.3 在 appsettings.json 中定义客户端

AddInMemoryClients 扩展方法还支持从 ASP.NET Core 配置文件添加客户端。这允许您直接从 appsettings.json 文件定义静态客户端:

```
"IdentityServer": {
  "IssuerUri": "urn:sso.company.com",
  "Clients": [
    {
      "Enabled": true,
      "ClientId": "local-dev",
      "ClientName": "Local Development",
      "ClientSecrets": [ { "Value": "<插入编码为 Base64 字符串的密钥的 Sha256 哈希>" }
    ↪ ],
      "AllowedGrantTypes": [ "client_credentials" ],
      "AllowedScopes": [ "api1" ],
    }
  ]
}
```

然后将配置部分传递给 AddInMemoryClients 方法:

```
AddInMemoryClients(configuration.GetSection("IdentityServer:Clients"))
```


为了让 IdentityServer 代表用户颁发令牌，该用户必须登录到 IdentityServer。

18.1 Cookie 认证

使用由 ASP.NET Core 的 `cookie` 身份验证 处理程序管理的 `cookie` 跟踪身份验证。

IdentityServer 注册了两个 `cookie` 处理程序（一个用于身份验证会话，另一个用于临时外部 `cookie`）。

默认情况下使用它们，如果您想手动引用它们，您可以从 `IdentityServerConstants` 类（`DefaultCookieAuthenticationScheme` 和 `ExternalCookieAuthenticationScheme`）中获取它们的名称。

仅公开这些 `cookie` 的基本设置（过期和滑动），但如果您需要更多控制，您可以注册自己的 `cookie` 处理程序。当使用来自 ASP.NET Core 的 `AddAuthentication` 时，IdentityServer 使用与 `AuthenticationOptions` 上配置的 `DefaultAuthenticateScheme` 相匹配的任何 `cookie` 处理程序。

注解：除了身份验证 `cookie` 之外，IdentityServer 还将发布一个额外的 `cookie`，默认名称为 “idsrv.session”。此 `cookie` 派生自主身份验证 `cookie`，它用于注销时基于浏览器的 *JavaScript* 客户端 的检查会话端点。它与身份验证 `cookie` 保持同步，并在用户注销时删除。

18.2 覆盖 cookie 处理程序配置

如果您希望使用自己的 cookie 身份验证处理程序，则必须自己配置它。这必须在 DI 中注册 IdentityServer（使用 AddIdentityServer）之后在 ConfigureServices 中完成。例如：

```
services.AddIdentityServer()
    .AddInMemoryClients(Clients.Get())
    .AddInMemoryIdentityResources(Resources.GetIdentityResources())
    .AddInMemoryApiResources(Resources.GetApiResources())
    .AddDeveloperSigningCredential()
    .AddTestUsers(TestUsers.Users);

services.AddAuthentication("MyCookie")
    .AddCookie("MyCookie", options =>
    {
        options.ExpireTimeSpan = ...;
    });
```

注解： IdentityServer 使用自定义方案（通过常量 IdentityServerConstants.DefaultCookieAuthenticationScheme）在内部调用 AddAuthentication 和 AddCookie，因此要覆盖它们，您必须在 AddIdentityServer 之后进行相同的调用。

18.3 登录用户界面和身份管理系统

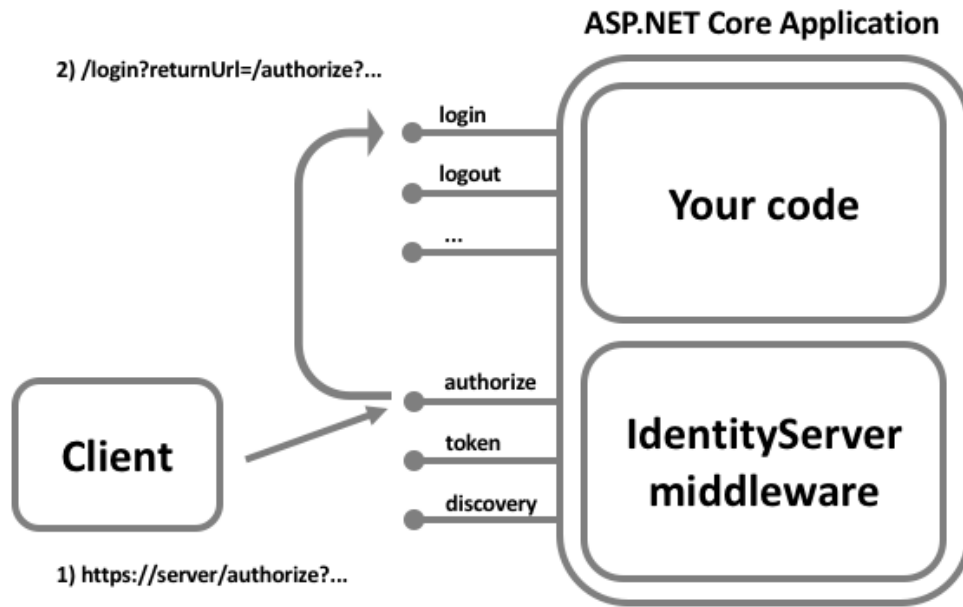
IdentityServer 不提供任何用于用户身份验证的用户界面或用户数据库。这些是您自己应该提供或开发的东西。

如果您需要一个基本 UI（登录、注销、同意和管理授权）的起点，您可以使用我们的 [快速入门 UI](#)。

快速入门 UI 根据内存数据库对用户进行身份验证。您可以使用对真实用户存储的访问权限来替换这些位。我们有使用 *ASP.NET Identity* 的示例。

18.4 登录工作流程

当 IdentityServer 在授权端点收到请求并且用户未通过身份验证时，用户将被重定向到配置的登录页面。您必须通过选项上的 UserInteraction 设置（默认为 /account/login）将登录页面的路径告知 IdentityServer。将传递一个 returnUrl 参数，通知您的登录页面，一旦登录完成，用户应该被重定向到该页面。



注解：通过 `returnUrl` 参数请注意 开放式重定向攻击。您应该验证 `returnUrl` 指的是众所周知的位置。请参阅 API 的 *interaction service* 以验证 `returnUrl` 参数。

18.5 登录上下文

在您的登录页面上，您可能需要有关请求上下文的信息以自定义登录体验（例如客户端、提示参数、IdP 提示或其他内容）。这是通过 *interaction service* 上的 `GetAuthorizationContextAsync` API 提供的。

18.6 颁发 cookie 和声明

ASP.NET Core 的 `HttpContext` 上有与身份验证相关的扩展方法，用于发出身份验证 cookie 并让用户登录。使用的身份验证方案必须与您使用的 cookie 处理程序匹配（见上文）。

当您登录用户时，您必须至少发布一个 `sub` 声明和一个 `name` 声明。IdentityServer 还在 `HttpContext` 上提供了一些 `SignInAsync` 扩展方法，以使其更方便。

您还可以选择发出 `idp` 声明（用于身份提供者名称）、`amr` 声明（用于使用的身份验证方法）和/或 `auth_time` 声明（用于纪元时间用户通过身份验证）。如果您不提供这些，则 IdentityServer 将提供默认值。

CHAPTER 19

使用外部身份提供商登录

ASP.NET Core 有一种灵活的方式来处理外部身份验证。这涉及几个步骤。

注解：如果您使用的是 ASP.NET Identity，许多底层技术细节对您来说是隐藏的。建议您还阅读 [Microsoft 文档](#) 并执行 ASP.NET Identity 快速入门。

19.1 为外部提供商添加身份验证处理程序

与外部提供者对话所需的协议实现封装在 *authentication handler* 中。一些提供商使用专有协议（例如 Facebook 等社交提供商），一些提供商使用标准协议，例如 OpenID Connect、WS-Federation 或 SAML2p。

有关添加外部身份验证和配置它的分步说明，请参阅此快速入门。

19.2 Cookie 的作用

外部身份验证处理程序的一个选项称为 `SignInScheme`，例如：

```
services.AddAuthentication()
    .AddGoogle("Google", options =>
    {
        options.SignInScheme = "要使用的 cookie 处理程序方案";
    })
```

(下页继续)

(续上页)

```
options.ClientId = "...";
options.ClientSecret = "...";
})
```

登录方案指定将临时存储外部身份验证结果的 cookie 处理程序的名称，例如，由外部提供商发送的声明。这是必要的，因为在您完成外部身份验证过程之前，通常会涉及几个重定向。

鉴于这是一种常见的做法，IdentityServer 专门为此外部提供程序工作流注册了一个 cookie 处理程序。该方案通过 IdentityServerConstants.ExternalCookieAuthenticationScheme 常量表示。如果您要使用我们的外部 cookie 处理程序，那么对于上面的 SignInScheme，您需要将值分配为 IdentityServerConstants.ExternalCookieAuthenticationScheme 常量：

```
services.AddAuthentication()
    .AddGoogle("Google", options =>
    {
        options.SignInScheme = IdentityServerConstants.
↪ExternalCookieAuthenticationScheme;

        options.ClientId = "...";
        options.ClientSecret = "...";
    })
```

您也可以注册自己的自定义 cookie 处理程序，如下所示：

```
services.AddAuthentication()
    .AddCookie("你的自定义方案")
    .AddGoogle("Google", options =>
    {
        options.SignInScheme = "你的自定义方案";

        options.ClientId = "...";
        options.ClientSecret = "...";
    })
```

注解：对于特殊场景，您还可以短路外部 cookie 机制，将外部用户直接转发到主 cookie 处理程序。这通常涉及处理外部处理程序上的事件，以确保您从外部身份源执行正确的声明转换。

19.3 触发身份验证处理程序

您可以通过 `HttpContext` 上的 `ChallengeAsync` 扩展方法（或使用 `MVC ChallengeResult`）调用外部身份验证处理程序。

您通常希望将一些选项传递给 `challenge` 操作，例如回调页面的路径和簿记提供者的名称，例如：

```
var callbackUrl = Url.Action("ExternalLoginCallback");

var props = new AuthenticationProperties
{
    RedirectUri = callbackUrl,
    Items =
    {
        { "scheme", provider },
        { "returnUrl", returnUrl }
    }
};

return Challenge(provider, props);
```

19.4 处理回调并登录用户

在回调页面上，您的典型任务是：

- 检查外部提供者返回的身份。
- 决定如何与该用户打交道。根据这是新用户还是回访用户的事实，这可能会有所不同。
- 新用户在被允许进入之前可能需要额外的步骤和 UI。
- 可能会创建一个链接到外部提供商的新内部用户帐户。
- 存储您想要保留的外部声明。
- 删除临时 cookie
- 登录用户

检查外部身份：

```
// 从临时 cookie 中读取外部身份
var result = await HttpContext.AuthenticateAsync(IdentityServerConstants.
    ExternalCookieAuthenticationScheme);
if (result?.Succeeded != true)
{
    throw new Exception("外部认证错误");
}
```

(下页继续)

(续上页)

```

}

// 检索外部用户的声明
var externalUser = result.Principal;
if (externalUser == null)
{
    throw new Exception("外部认证错误");
}

// 检索外部用户的声明
var claims = externalUser.Claims.ToList();

// 尝试确定外部用户的唯一 ID —— 最常见的声明类型是 sub 声明和 NameIdentifier,
// 具体取决于外部提供者, 可能会使用其他一些声明类型
var userIdClaim = claims.FirstOrDefault(x => x.Type == JwtClaimTypes.Subject);
if (userIdClaim == null)
{
    userIdClaim = claims.FirstOrDefault(x => x.Type == ClaimTypes.NameIdentifier);
}
if (userIdClaim == null)
{
    throw new Exception("未知 userid");
}

var externalUserId = userIdClaim.Value;
var externalProvider = userIdClaim.Issuer;

// 使用 externalProvider 和 externalUserId 查找您的用户, 或配置新用户

```

清理和登录:

```

// 为用户颁发身份验证 cookie
await HttpContext.SignInAsync(new IdentityServerUser(user.SubjectId) {
    DisplayName = user.Username,
    IdentityProvider = provider,
    AdditionalClaims = additionalClaims,
    AuthenticationTime = DateTime.Now
});

// 删除外部身份验证期间使用的临时 cookie
await HttpContext.SignOutAsync(IdentityServerConstants.
    ↪ExternalCookieAuthenticationScheme);

// 验证返回 URL 并重定向回授权端点或本地页面

```

(下页继续)

(续上页)

```
if (_interaction.IsValidReturnUrl(returnUrl) || Url.IsLocalUrl(returnUrl))
{
    return Redirect(returnUrl);
}

return Redirect("~/");
```

19.5 状态、URL 长度和 ISecureDataFormat

当重定向到外部提供程序进行登录时，来自客户端应用程序的状态必须经常往返。这意味着在离开客户端之前捕获状态并保留直到用户返回到客户端应用程序。许多协议，包括 OpenID Connect，允许将某种状态作为参数作为请求的一部分传递，身份提供者将在响应中返回该状态。ASP.NET Core 提供的 OpenID Connect 身份验证处理程序利用了协议的这一特性，这就是它实现上述 returnUrl 特性的方式。

在请求参数中存储状态的问题是请求 URL 可能变得太大（超过 2000 个字符的常见限制）。OpenID Connect 身份验证处理程序确实提供了一个可扩展点来将状态存储在您的服务器中，而不是存储在请求 URL 中。您可以通过实现 ISecureDataFormat<AuthenticationProperties> 并在 [OpenIdConnectOptions](#) 上配置它来实现。

幸运的是，IdentityServer 为您提供了一个实现，由在 DI 容器中注册的 IDistributedCache 实现支持（例如标准的 MemoryDistributedCache）。要使用 IdentityServer 提供的安全数据格式实现，只需在配置 DI 时调用 IServiceCollection 上的 AddOidcStateDataFormatterCache 扩展方法。如果没有传递参数，那么所有配置的 OpenID Connect 处理程序将使用 IdentityServer 提供的安全数据格式实现：

```
public void ConfigureServices(IServiceCollection services)
{
    // 配置 OpenIdConnect 处理程序以将状态参数持久化到服务器端 IDistributedCache。
    services.AddOidcStateDataFormatterCache();

    services.AddAuthentication()
        .AddOpenIdConnect("demoidsrv", "IdentityServer", options =>
        {
            // ...
        })
        .AddOpenIdConnect("aad", "Azure AD", options =>
        {
            // ...
        })
        .AddOpenIdConnect("adfs", "ADFS", options =>
        {
            // ...
        });
}
```

(下页继续)

(续上页)

```
}
```

如果只配置特定的方案，则将这些方案作为参数传递：

```
public void ConfigureServices(IServiceCollection services)
{
    // 配置 OpenIdConnect 处理程序以将状态参数持久化到服务器端 IDistributedCache。
    services.AddOidcStateDataFormatterCache("aad", "demoidsrv");

    services.AddAuthentication()
        .AddOpenIdConnect("demoidsrv", "IdentityServer", options =>
        {
            // ...
        })
        .AddOpenIdConnect("aad", "Azure AD", options =>
        {
            // ...
        })
        .AddOpenIdConnect("adfs", "ADFS", options =>
        {
            // ...
        });
}
```

CHAPTER 20

Windows 身份验证

您可以通过多种方式在 ASP.NET Core（以及 IdentityServer）中启用 Windows 身份验证。

- 在 Windows 上使用 IIS 托管（进程内和进程外）
- 在 Windows 上使用 HTTP.SYS 托管
- 在任何使用协商身份验证处理程序的平台上（在 ASP.NET Core 3.0 中添加）

注解：我们只有 IIS 托管的文档。如果您想为文档做出贡献，请打开一个 PR。谢谢！

20.1 在使用 IIS 托管的 Windows 上

当托管在 IIS 中时，典型的 CreateDefaultBuilder 主机设置支持基于 IIS 的 Windows 身份验证。确保在 launchSettings.json 或您的 IIS 配置中启用了 Windows 身份验证。

IIS 集成层会将 Windows 身份验证处理程序配置到可通过身份验证服务调用的 DI 中。通常在 IdentityServer 中，建议禁用自动行为。

这是在 ConfigureServices 中完成的（详细信息取决于进程内与进程外托管）：

```
// 配置 IIS 进程外设置（参见 https://github.com/aspnet/AspNetCore/issues/14882）
services.Configure<IISSOptions>(iis =>
{
    iis.AuthenticationDisplayName = "Windows";
```

(下页继续)

(续上页)

```

        iis.AutomaticAuthentication = false;
    });

    // ..或配置 IIS 进程内设置
    services.Configure<IISServerOptions>(iis =>
    {
        iis.AuthenticationDisplayName = "Windows";
        iis.AutomaticAuthentication = false;
    });

```

您可以通过在 Windows 方案上调用 `ChallengeAsync` 来触发 Windows 身份验证（或者如果您想使用常量：`Microsoft.AspNetCore.Server.IISIntegration.IISDefaults.AuthenticationScheme`）。

这会将 `Www-Authenticate` 标头发送回浏览器，然后浏览器将重新加载当前的 URL，包括 Windows 身份。当您在 Windows 方案上调用 `AuthenticateAsync` 并且返回的主体类型为 `WindowsPrincipal` 时，您可以判断 Windows 身份验证成功。

主体将具有用户和组 SID 以及 Windows 帐户名称等信息。以下代码段显示了如何触发身份验证，如果成功，则将信息转换为 `temp-Cookie` 方法的标准 `ClaimsPrincipal`：

```

private async Task<IActionResult> ChallengeWindowsAsync(string returnUrl)
{
    // 查看是否已请求 Windows 身份验证并成功
    var result = await HttpContext.AuthenticateAsync("Windows");
    if (result?.Principal is WindowsPrincipal wp)
    {
        // 我们将发出外部 cookie，
        // 然后将用户重定向回外部回调，
        // 本质上，将 windows auth 视为与任何其他外部身份验证机制相同
        var props = new AuthenticationProperties()
        {
            RedirectUri = Url.Action("Callback"),
            Items =
            {
                { "returnUrl", returnUrl },
                { "scheme", "Windows" },
            }
        };

        var id = new ClaimsIdentity("Windows");

        // sid 是一个很好的子值
        id.AddClaim(new Claim(JwtClaimTypes.Subject, wp.FindFirst(ClaimTypes.
        ↪PrimarySid).Value));
    }
}

```

(下页继续)

(续上页)

```
// 帐户名称是我们最接近显示名称的名称
id.AddClaim(new Claim(JwtClaimTypes.Name, wp.Identity.Name));

// 添加组作为声明 —— 如果组数太大, 请小心
var wi = wp.Identity as WindowsIdentity;

// 将组 SID 转换为显示名称
var groups = wi.Groups.Translate(typeof(NTAccount));
var roles = groups.Select(x => new Claim(JwtClaimTypes.Role, x.Value));
id.AddClaims(roles);

await HttpContext.SignInAsync(
    IdentityServerConstants.ExternalCookieAuthenticationScheme,
    new ClaimsPrincipal(id),
    props);
return Redirect(props.RedirectUri);
}
else
{
    // 触发 windows auth
    // 因为 windows auth 不支持重定向 uri,
    // 当我们调用 challenge 时会重新触发这个 URL
    return Challenge("Windows");
}
}
```


注销 IdentityServer 就像删除身份验证 cookie 一样简单，但是要进行完整的联合注销，我们还必须考虑将用户从客户端应用程序（甚至可能是上游身份提供商）中注销。

21.1 删除身份验证 cookie

要删除身份验证 cookie，只需在 HttpContext 上使用 SignOutAsync 扩展方法。您将需要传递使用的方案（由 IdentityServerConstants.DefaultCookieAuthenticationScheme 提供，除非您已更改它）：

```
await HttpContext.SignOutAsync(IdentityServerConstants.  
    ↪DefaultCookieAuthenticationScheme);
```

或者您可以使用 IdentityServer 提供的便捷扩展方法：

```
await HttpContext.SignOutAsync();
```

注解：通常，您应该提示用户注销（意味着需要 POST），否则攻击者可能会热链接到您的注销页面，从而导致用户自动注销。

21.2 通知客户端用户已注销

作为注销过程的一部分，您需要确保通知客户端应用程序用户已注销。IdentityServer 支持服务器端客户端（例如 MVC）的 前端渠道 规范，服务器端客户端（例如 MVC）的 后端渠道 规范，以及基于浏览器的 JavaScript 客户端（例如 SPA、React、Angular 等）的 会话管理 规范。

前端渠道服务器端客户端

要通过前端渠道规范从服务器端客户端应用程序中注销用户，IdentityServer 中的 注销页面 必须呈现一个 `<iframe>` 以通知客户端用户已注销。希望收到通知的客户端必须设置 `FrontChannelLogoutUri` 配置值。IdentityServer 跟踪用户登录了哪些客户端，并在 `IIdentityServerInteractionService`（详情）上提供了一个名为 `GetLogoutContextAsync` 的 API。此 API 返回一个带有 `SignOutIFrameUrl` 属性的 `LogoutRequest` 对象，您的注销页面必须将其呈现到 `<iframe>` 中。

后端渠道服务器端客户端

要通过后端渠道规范从服务器端客户端应用程序注销用户，可以使用 `IBackChannelLogoutService` 服务。当您的注销页面通过调用 `HttpContext.SignOutAsync` 删除用户的身份验证 cookie 时，IdentityServer 将自动使用此服务。希望收到通知的客户端必须设置 `BackChannelLogoutUri` 配置值。

基于浏览器的 JavaScript 客户端

鉴于 会话管理 规范是如何设计的，在 IdentityServer 中，您不需要做什么特殊的事情来通知这些客户端用户已注销。但是，客户端必须对 `check_session_iframe` 执行监控，这是由 `oidc-client` JavaScript 库 实现的。

21.3 由客户端应用程序发起的注销

如果注销是由客户端应用程序发起的，那么客户端首先将用户重定向到结束会话端点。结束会话端点的处理可能需要在重定向到注销页面的过程中维护一些临时状态（例如，客户端的注销后重定向 uri）。此状态可能对注销页面有用，并且该状态的标识符通过 `logoutId` 参数传递到注销页面。

交互服务 上的 `GetLogoutContextAsync` API 可用于加载状态。对 `LogoutRequest` 模型上下文类感兴趣的是 `ShowSignoutPrompt`，它指示注销请求是否已通过身份验证，因此不提示用户注销是安全的。

默认情况下，此状态作为通过 `logoutId` 值传递的受保护数据结构进行管理。如果您希望在结束会话端点和注销页面之间使用其他持久性，那么您可以实现 `IMessageStore<LogoutMessage>` 并在 DI 中注册实现。

注销外部身份提供商

当用户从 IdentityServer 注销时，如果他们使用了外部身份提供商登录，则很可能还应该将他们重定向到注销外部提供商。并非所有外部提供商都支持注销，因为这取决于他们支持的协议和功能。

检测用户是否必须重定向到外部身份提供商以进行注销，通常是通过使用在 IdentityServer 的 cookie 中发出的 idp 声明来完成的。设置到此声明中的值是相应身份验证中间件的 AuthenticationScheme。在注销时，将咨询此声明以了解是否需要外部注销。

由于正常注销工作流程已经需要清理和状态管理，因此将用户重定向到外部身份提供者是有问题的。完成 IdentityServer 的正常注销和清理过程的唯一方法是，从外部身份提供商请求在其注销后，将用户重定向回 IdentityServer。并非所有外部提供商都支持注销后重定向，因为这取决于它们支持的协议和功能。

注销时的工作流程是撤销 IdentityServer 的身份验证 cookie，然后重定向到请求注销后重定向的外部提供商。注销后重定向应保持此处描述的必要注销状态（即 logoutId 参数值）。要在外部提供商注销后重定向回 IdentityServer，例如，当使用 ASP.NET Core 的 SignOutAsync API 时，应在 AuthenticationProperties 上使用 RedirectUri：

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Logout (LogoutInputModel model)
{
    // 建立一个模型，以便注销页面知道要显示什么
    var vm = await _account.BuildLoggedOutViewModelAsync (model.LogoutId);

    var user = HttpContext.User;
    if (user?.Identity.IsAuthenticated == true)
```

(下页继续)

```
{
    // 删除本地认证 cookie
    await HttpContext.SignOutAsync();

    // 引发注销事件
    await _events.RaiseAsync(new UserLogoutSuccessEvent(user.GetSubjectId(), user.
↵GetName()));
}

// 检查我们是否需要在上游身份提供商处触发注销
if (vm.TriggerExternalSignout)
{
    // 构建一个返回 URL，以便上游提供商在用户注销后重定向回我们。
    // 这样我们就可以完成单点注销处理。
    string url = Url.Action("Logout", new { logoutId = vm.LogoutId });

    // 这将触发重定向到外部提供商以进行注销
    return SignOut(new AuthenticationProperties { RedirectUri = url }, vm.
↵ExternalAuthenticationScheme);
}

return View("LoggedOut", vm);
}
```

一旦用户从外部提供商注销，然后重定向回来，IdentityServer 上的正常注销处理就应该执行，这包括处理 logoutId 并进行所有必要的清理。

联合注销

联合注销是指用户使用外部身份提供商登录 IdentityServer，然后用户通过 IdentityServer 未知的工作流注销该外部身份提供商的情景。当用户退出时，IdentityServer 会收到通知，以便它可以让用户注销 IdentityServer 和所有使用 IdentityServer 的应用程序。

并非所有外部身份提供商都支持联合注销，但支持联合注销的提供商将提供一种机制来通知客户端用户已注销。此通知通常以来请求的形式出现在外部身份提供商的注销页面的 `<iframe>` 中。IdentityServer 随后必须通知它的所有客户端（如此处所述），通常也是以来请求的形式从外部身份提供商的 `<iframe>` 中的发出通知。

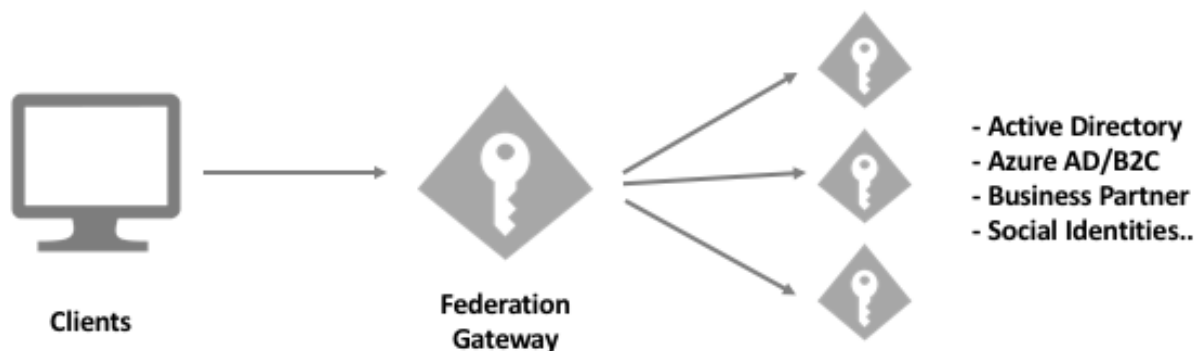
使联合注销成为一种特殊情况（与正常的注销相比）的原因是，联合注销请求不属于 IdentityServer 中的正常注销端点。事实上，每个外部 IdentityProvider 在您的 IdentityServer 主机中都有一个不同的端点。这是因为每个外部身份提供商可能使用不同的协议，并且每个中间件侦听不同的端点。

所有这些因素的最终效果是，没有像我们在正常注销工作流中那样呈现注销页面，这意味着我们错过了 IdentityServer 客户端的注销通知。我们必须为每个联合注销端点添加代码，以呈现实现联合注销所需的通知。

幸运的是 IdentityServer 已经包含了此代码。当请求进入 IdentityServer 并调用外部身份验证提供商的处理程序时，IdentityServer 会检测这些是否是联合注销请求，如果是，它将自动呈现与此处描述的注销相同的 `<iframe>`。简而言之，自动支持联合注销。

联合网关

一种常见的体系结构是所谓的联合网关。在这种方法中，IdentityServer 充当一个或多个外部身份提供商的网关。



这种体系结构有以下优点

- 您的应用程序只需要知道一个令牌服务（网关），并且不需要了解有关连接到外部提供商的所有详细信息。这也意味着您可以添加或更改这些外部提供商，而无需更新您的应用程序。
- 您可以控制网关（与某些外部服务提供商相反）——这意味着您可以对其进行任何更改，并可以保护您的应用程序免受这些外部提供商可能对其自己的服务所做的更改。
- 大多数外部提供商只支持一组固定的声明和声明类型——中间有一个网关允许对来自提供商的响应进行后处理以转换/添加/修改特定于域的身份信息。
- 一些提供商不支持访问令牌（例如社交提供者）——因为网关知道你的 API，它可以根据外部身份发布

访问令牌。

- 一些提供商根据您连接到他们的应用程序数量收费。网关充当外部提供商的单个应用程序。在内部，您可以根据需要连接任意数量的应用程序。
- 一些提供商使用专有协议或对标准协议进行专有修改——使用网关，您只需要处理一个地方。
- 强制每个身份验证（内部或外部）都通过一个地方，这在身份映射方面为您提供了极大的灵活性，为您的所有应用程序提供了稳定的身份，并可以处理新的需求

换句话说——拥有联合网关可以让您身份基础设施进行大量控制。由于您的用户身份是您最重要的资产之一，我们建议您控制网关。

24.1 执行

我们的 [快速启动 UI](#) 利用了以下一些功能。另请查看[外部身份验证快速入门](#) 和 [有关外部提供商](#) 的文档。

- 您可以通过向 IdentityServer 应用程序添加身份验证处理程序来添加对外部身份提供商的支持。
- 您可以通过调用 `IAuthenticationSchemeProvider` 以编程方式查询这些外部提供商。这允许根据注册的外部提供商动态呈现您的登录页面。
- 我们的客户端配置模型允许在每个客户端的基础上限制可用的提供商（使用 `IdentityProviderRestrictions` 属性）。
- 您还可以使用客户端上的 `EnableLocalLogin` 属性来告诉您的 UI 是否应呈现用户名/密码输入。
- 我们的快速入门 UI 通过一次回调（请参阅“`AccountController`”类中的“`ExternalLoginCallback`”）来传递所有外部身份验证调用。这允许单点进行后处理。

在授权请求期间，如果 `IdentityServer` 需要用户同意，浏览器将被重定向到同意页面。

同意用于允许最终用户授予客户端对资源（*identity* 或 *API*）的访问权限。这通常只对第三方客户端是必需的，并且可以在客户端设置上为每个客户端启用/禁用。

25.1 同意页面

为了让用户同意，托管应用程序必须提供同意页面。[快速启动](#) 有一个同意页面的基本实现。

同意页面通常会呈现当前用户的显示名称，请求访问的客户端的显示名称，客户端的 `Logo`，有关客户端的更多信息的链接以及客户端请求访问的资源列表。允许用户表明他们的同意应该被“记住”也是很常见的，这样以后就不会为同一客户端再次提示他们了。

一旦用户提供了同意，同意页面必须通知 `IdentityServer` 同意，然后浏览器必须重定向回授权端点。

25.2 授权上下文

`IdentityServer` 将一个 `returnUrl` 参数（可在用户交互选项上配置）传递到包含授权请求参数的同意页面。这些参数提供同意页面的上下文，可以在交互服务的帮助下读取。`GetAuthorizationContextAsync` API 将返回一个 `AuthorizationRequest` 的实例。

可以使用 `IClientStore` 和 `IResourceStore` 接口获取有关客户端或资源的其他详细信息。

25.3 将同意结果通知 IdentityServer

交互服务上的 `GrantConsentAsync` API 允许同意页面通知 IdentityServer 同意的结果（也可能是拒绝客户端访问）。

IdentityServer 将暂时保留同意的结果。默认情况下，此持久性使用 cookie，因为它只需要持续足够长的时间即可将结果传送回授权端点。这种临时持久性不同于用于“记住我的同意”功能的持久性（授权端点为用户维持“记住我的同意”）。如果您希望在同意页面和授权重定向之间使用其他持久性，那么您可以实现 `IMessageStore<ConsentResponse>` 并在 DI 中注册实现。

25.4 将用户返回到授权端点

一旦同意页面将结果通知 IdentityServer，用户就可以被重定向回 `returnUrl`。您的同意页面应通过验证 `returnUrl` 是否有效来防止打开重定向。这可以通过在交互服务上调用 `IsValidReturnUrl` 来完成。此外，如果 `GetAuthorizationContextAsync` 返回非空结果，那么您也可以相信 `returnUrl` 是有效的。

IdentityServer 默认以 **JWT** (JSON Web 令牌) 格式发布访问令牌。

现在每个相关平台都支持验证 **JWT** 令牌，可以在 [此处](#) 找到一个很好的 **JWT** 库列表。流行的类库如：

- 用于 ASP.NET Core 的 **JWT bearer** 身份验证处理程序
- 用于 Katana 的 **JWT bearer** 身份验证中间件

保护基于 ASP.NET Core 的 API 只是添加 **JWT bearer** 身份验证处理程序的问题：

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
            .AddJwtBearer(options =>
            {
                // 您的 IdentityServer 的基地址
                options.Authority = "https://demo.identityserver.io";

                // 如果您使用的是 API 资源，可以在此处指定名称
                options.Audience = "resource1";

                // 默认情况下 IdentityServer 发出一个 typ 标头，建议额外检查
                options.TokenValidationParameters.ValidTypes = new[] { "at+jwt" };
            });
    }
}
```

(下页继续)

(续上页)

```
}  
}
```

注解: 如果您没有使用受众声明, 您可以通过 `options.TokenValidationParameters.ValidateAudience = false;` 关闭受众检查。有关资源、范围、受众和授权的更多信息, 请参见 [此处](#)。

26.1 验证引用令牌

如果您使用引用令牌, 则需要一个实现 OAuth 2.0 令牌自省 的身份验证处理程序, 例如 这样:

```
services.AddAuthentication("token")  
    .AddOAuth2Introspection("token", options =>  
    {  
        options.Authority = Constants.Authority;  
  
        // 这映射到 API 资源名称和密钥  
        options.ClientId = "resource1";  
        options.ClientSecret = "secret";  
    });
```

26.2 支持 JWT 和引用令牌

您可以设置 ASP.NET Core 以根据传入的令牌分派到正确的处理程序, 有关更多信息, 请参阅 [这篇](#) 博客文章。在这种情况下, 您设置一个默认处理程序和一些转发逻辑, 例如:

```
services.AddAuthentication("token")  
  
    // JWT 令牌  
    .AddJwtBearer("token", options =>  
    {  
        options.Authority = Constants.Authority;  
        options.Audience = "resource1";  
  
        options.TokenValidationParameters.ValidTypes = new[] { "at+jwt" };  
  
        // 如果令牌不包含点, 则为引用令牌  
        options.ForwardDefaultSelector = Selector.ForwardReferenceToken("introspection  
->");
```

(下页继续)

(续上页)

```
    })

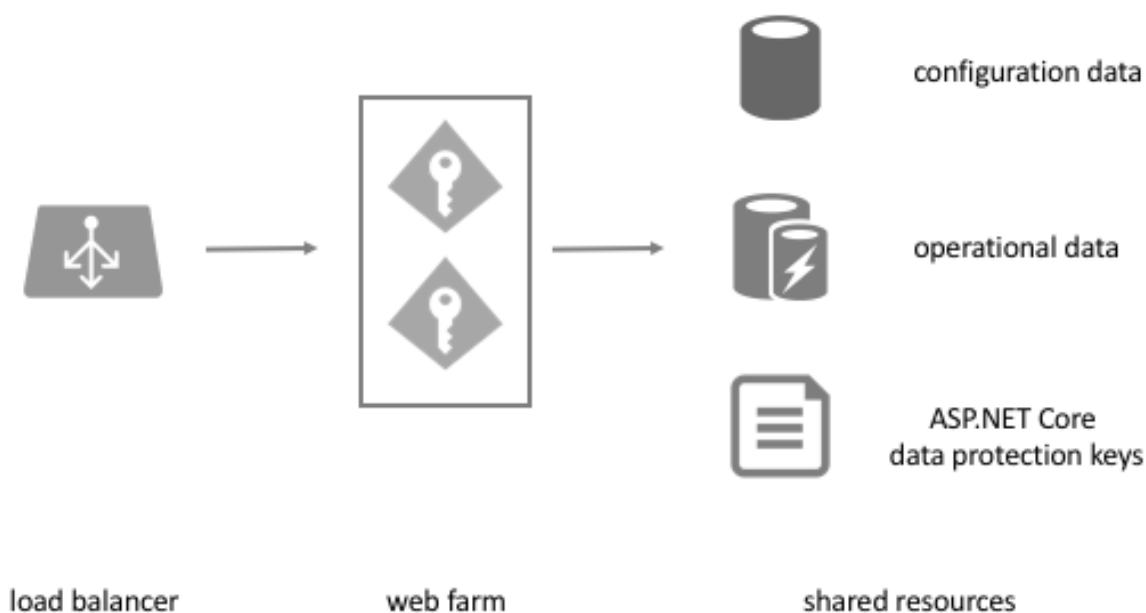
    // 引用令牌
    .AddOAuth2Introspection("introspection", options =>
    {
        options.Authority = Constants.Authority;

        options.ClientId = "resource1";
        options.ClientSecret = "secret";
    });
```


您的身份服务器 只是一个包含 IdentityServer 中间件的标准 ASP.NET Core 应用程序。请先阅读有关发布和部署的 [Microsoft 官方 文档](#)（尤其是有关负载均衡器和代理的 [部分](#)）。

27.1 典型架构

通常，您将设计 IdentityServer 部署以实现高可用性：



IdentityServer 本身是无状态的，不需要服务器关联——但需要在实例之间共享数据。

27.2 配置数据

这通常包括:

- 资源
- 客户端
- 启动配置，例如密钥材料、外部供应商设置等...

您存储该数据的方式取决于您的环境。在配置数据很少更改的情况下，我们建议使用内存存储和代码或配置文件。

在高度动态的环境（例如 SaaS）中，我们建议使用数据库或配置服务来动态加载配置。

IdentityServer 支持开箱即用的代码配置和配置文件（参见 [此处](#)）。对于数据库，我们提供对基于 [Entity Framework Core](#) 的数据库的支持。

您还可以通过实现 `IResourceStore` 和 `IClientStore` 来构建自己的配置存储。

27.3 密钥材料

启动配置的另一个重要部分是您的密钥材料，有关密钥材料和密码学的更多详细信息，请参见[此处](#)。

27.4 操作数据

对于某些操作，IdentityServer 需要一个持久化存储来保持状态，这包括：

- 颁发授权码
- 颁发引用令牌和刷新令牌
- 存储同意

您可以使用传统数据库来存储操作数据，也可以使用具有持久性功能（如 Redis）的缓存。上面提到的 EF Core 实现也支持操作数据。

您还可以通过实现 `IPersistedGrantStore` 来实现对您自己的自定义存储机制的支持——默认情况下，IdentityServer 注入内存版本。

27.5 ASP.NET Core 数据保护

ASP.NET Core 本身需要共享密钥材料来保护敏感数据，如 cookie、状态字符串等。请参阅官方文档 [这里](#)。

如果可能的话，您可以重用上述持久性存储之一，也可以使用一些简单的东西，例如共享文件。

27.6 ASP.NET Core 分布式缓存

某些组件依赖于 ASP.NET Core 分布式缓存。为了在多服务器环境中工作，需要正确设置。[官方文档](#) 描述了几个选项。

以下组件依赖于 `IDistributedCache`：

- `services.AddOidcStateDataFormatterCache()` 配置 `OpenIdConnect` 处理程序以将状态参数持久化到服务器端 `IDistributedCache`。
- `DefaultReplayCache`
- `DistributedDeviceFlowThrottlingService`
- `DistributedCacheAuthorizationParametersMessageStore`

IdentityServer 使用 ASP.NET Core 提供的标准日志记录工具。Microsoft 文档 有一个很好的介绍和内置日志提供程序的描述。

我们大致遵循 Microsoft 的日志级别使用指南：

- Trace 用于仅对开发人员解决问题有价值的信息。这些消息可能包含敏感的应用程序数据，如令牌，不应在生产环境中启用。
- Debug 用于遵循内部流程并理解做出某些决定的原因。在开发和调试期间有短期的用处。
- Information 用于跟踪应用程序的一般流程。这些日志通常具有一些长期价值。
- Warning 用于应用程序流中的异常或意外事件。这些可能包括不会导致应用程序停止但可能需要调查的错误或其他情况。
- Error 用于无法处理的错误和异常。示例：协议请求验证失败。
- Critical 用于需要立即关注的故障。示例：缺少存储实现、无效的密钥材料…

28.1 Serilog 的设置

我们个人非常喜欢 Serilog 和 Serilog.AspNetCore 包。试一试：

```
public class Program
{
    public static int Main(string[] args)
```

(下页继续)

```

{
    Activity.DefaultIdFormat = ActivityIdFormat.W3C;

    Log.Logger = new LoggerConfiguration()
        .MinimumLevel.Debug()
        .MinimumLevel.Override("Microsoft", LogEventLevel.Warning)
        .MinimumLevel.Override("Microsoft.Hosting.Lifetime", LogEventLevel.
↪Information)
        .MinimumLevel.Override("System", LogEventLevel.Warning)
        .MinimumLevel.Override("Microsoft.AspNetCore.Authentication", LogEventLevel.Information)
        .Enrich.FromLogContext()
        .WriteTo.Console(outputTemplate: "[{Timestamp:HH:mm:ss} {Level}] {SourceContext}{NewLine}{Message:lj}{NewLine}{Exception}{NewLine}", theme: AnsiConsoleTheme.Code)
        .CreateLogger();

    try
    {
        Log.Information("Starting host...");
        CreateHostBuilder(args).Build().Run();
        return 0;
    }
    catch (Exception ex)
    {
        Log.Fatal(ex, "Host terminated unexpectedly.");
        return 1;
    }
    finally
    {
        Log.CloseAndFlush();
    }
}

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Microsoft.Extensions.Hosting.Host.CreateDefaultBuilder(args)
        .UseSerilog()
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
}

```

事件

虽然日志记录是更低级别的“printf”样式——但事件表示有关 IdentityServer 中某些操作的更高级别信息。事件是结构化数据，包括事件 ID、成功/失败信息、类别和详细信息。这使得查询和分析它们并提取可用于进一步处理的有用信息变得容易。

事件与诸如 ELK、Seq 或 Splunk 之类的事件存储配合得很好。

29.1 发出事件

默认情况下没有打开事件——但可以在 ConfigureServices 方法中全局配置，例如：

```
services.AddIdentityServer(options =>
{
    options.Events.RaiseSuccessEvents = true;
    options.Events.RaiseFailureEvents = true;
    options.Events.RaiseErrorEvents = true;
});
```

要发出事件，请使用 DI 容器中的 IEventService 并调用 RaiseAsync 方法，例如：

```
public async Task<IActionResult> Login(LoginInputModel model)
{
    if (_users.ValidateCredentials(model.Username, model.Password))
    {
        // 发出带有 subject ID 和 username 的身份验证 cookie
```

(下页继续)

(续上页)

```

        var user = _users.FindByUsername(model.Username);
        await _events.RaiseAsync(new UserLoginSuccessEvent(user.Username, user.
↪SubjectId, user.Username));
    }
    else
    {
        await _events.RaiseAsync(new UserLoginFailureEvent(model.Username, "invalid_
↪credentials"));
    }
}

```

29.2 自定义 sinks

我们的默认事件接收器将简单地将事件类序列化为 JSON，并将其转发到 ASP.NET Core 日志记录系统。如果要连接到自定义事件存储，请实现 `IEventSink` 接口并将其注册到 DI。

以下示例使用 `Seq` 来发出事件：

```

public class SeqEventSink : IEventSink
{
    private readonly Logger _log;

    public SeqEventSink()
    {
        _log = new LoggerConfiguration()
            .WriteTo.Seq("http://localhost:5341")
            .CreateLogger();
    }

    public Task PersistAsync(Event evt)
    {
        if (evt.EventType == EventTypes.Success ||
            evt.EventType == EventTypes.Information)
        {
            _log.Information("{Name} ({Id}), Details: {@details}",
                evt.Name,
                evt.Id,
                evt);
        }
        else
        {
            _log.Error("{Name} ({Id}), Details: {@details}",

```

(下页继续)

(续上页)

```
        evt.Name,
        evt.Id,
        evt);
    }

    return Task.CompletedTask;
}
}
```

将 `Serilog.Sinks.Seq` 包添加到您的主机以使上述代码工作。

29.3 内置事件

IdentityServer 中定义了以下事件:

ApiAuthenticationFailureEvent & ApiAuthenticationSuccessEvent 在内省端点处为成功/失败的 API 身份验证引发。

ClientAuthenticationSuccessEvent & ClientAuthenticationFailureEvent 在令牌端点处为成功/失败的客户端身份验证引发。

TokenIssuedSuccessEvent & TokenIssuedFailureEvent 为尝试请求身份令牌、访问令牌、刷新令牌和授权代码的成功/失败引发。

TokenIntrospectionSuccessEvent & TokenIntrospectionFailureEvent 为成功的令牌自省请求引发。

TokenRevokedSuccessEvent 为成功的令牌吊销请求引发。

UserLoginSuccessEvent & UserLoginFailureEvent 由成功/失败的用户登录的快速入门 UI 引发。

UserLogoutSuccessEvent 为成功的注销请求引发。

ConsentGrantedEvent & ConsentDeniedEvent 在同意 UI 中引发。

UnhandledExceptionEvent 为未处理的异常引发。

DeviceAuthorizationFailureEvent & DeviceAuthorizationSuccessEvent 为成功/失败的设备授权请求引发。

29.4 自定义事件

您可以创建自己的事件并通过我们的基础设施发出它们。

您需要从我们的基础 `Event` 类派生，该类注入上下文信息，如 activity ID、timestamp 等。然后您的派生类可以添加特定于事件上下文的任意数据字段：

```
public class UserLoginFailureEvent : Event
{
    public UserLoginFailureEvent(string username, string error)
        : base(EventCategories.Authentication,
              "User Login Failure",
              EventTypes.Failure,
              EventIds.UserLoginFailure,
              error)
    {
        Username = username;
    }

    public string Username { get; set; }
}
```

密码学、密钥和 HTTPS

IdentityServer 依靠几种加密机制来完成其工作。

30.1 令牌签名和验证

IdentityServer 需要一个非对称密钥对来签名和验证 JWT。此密钥材料可以打包为证书，也可以打包为原始密钥。支持 RSA 和 ECDSA 密钥，支持的签名算法有：RS256、RS384、RS512、PS256、PS384、PS512、ES256、ES384 和 ES512。

您可以同时使用多个签名密钥，但每种算法仅支持一个签名密钥。您注册的第一个签名密钥被视为默认签名密钥。

客户端和 API 资源都可以表达对签名算法的偏好。如果您为多个 API 资源请求一个令牌，则所有资源都需要就至少一种允许的签名算法达成一致。

签名密钥和相应的验证部分的加载是通过 `ISigningCredentialStore` 和 `IValidationKeysStore` 的实现来完成的。如果您想自定义密钥的加载，您可以实现这些接口并将它们注册到 DI。

DI 构建器扩展有几个方便的方法来设置签名和验证密钥——请参阅[此处](#)。

30.2 签名密钥滚动更新

虽然您一次只能使用一个签名密钥，但您可以向发现文档发布多个验证密钥。这对于密钥滚动更新很有用。

简而言之，滚动更新通常是这样工作的：

1. 您请求/创建新的密钥材料
2. 除了当前的验证密钥之外，您还可以发布新的验证密钥。您可以使用 `AddValidationKey` 构建器扩展方法进行此操作。
3. 所有客户端和 API 现在都有机会在下次更新发现文档的本地副本时了解新密钥
4. 经过一段时间（例如 24 小时）后，所有客户端和 API 现在都应该同时接受旧的和新的密钥材料
5. 旧的密钥材料您想保留多久就保留多久，也许您需要验证的长期令牌
6. 当旧密钥材料不再使用时将其报废
7. 所有客户端和 API 将在下次更新发现文档的本地副本时“忘记”旧密钥

这要求客户端和 API 使用发现文档，并且还具有定期刷新其配置的功能。

Brock 写了一篇关于密钥轮换的更详细的 [博客文章](#)，还创建了一个 [商业组件](#)，可以自动处理所有这些细节。

30.3 数据保护

ASP.NET Core 中的 Cookie 身份验证（或 MVC 中的 anti-forgery）使用 ASP.NET Core 数据保护功能。根据您的部署方案，这可能需要额外的配置。有关更多信息，请参阅 [Microsoft 文档](#)。

30.4 HTTPS

我们不强制使用 HTTPS，但对于生产环境，与 IdentityServer 的每次交互都是强制性的。

授权类型

OpenID Connect 和 OAuth 2.0 规范定义了所谓的授权类型（通常也称为流程——或协议流程）。授权类型指定客户端如何与令牌服务交互。

您需要通过 `Client` 配置上的 `AllowedGrantTypes` 属性指定客户端可以使用哪些授权类型。这允许锁定给定客户端所允许的协议交互。

客户端可以配置为使用多种授权类型（例如，用于以用户为中心的操作的授权码流程和用于服务器到服务器通信的客户端凭据）。`GrantTypes` 类可用于从典型的授权类型组合中进行选择：

```
Client.AllowedGrantTypes = GrantTypes.CodeAndClientCredentials;
```

您还可以手动指定授权类型列表：

```
Client.AllowedGrantTypes =  
{  
    GrantType.Code,  
    GrantType.ClientCredentials,  
    "my_custom_grant_type"  
};
```

虽然 `IdentityServer` 支持所有标准授权类型，但对于常见的应用场景，您实际上只需要了解其中的两种。

31.1 机器到机器通信

这是最简单的通信类型。始终代表客户端请求令牌，不存在交互式用户。

在这种情况下，您使用 `client_credentials` 授权类型向令牌端点发送令牌请求。客户端通常必须使用其客户端 ID 和密钥对令牌端点进行身份验证。

有关如何使用它的示例，请参阅客户端凭据快速入门。

31.2 交互式客户端

这是最常见的客户端场景类型：具有交互式用户的 Web 应用程序、SPA 或原生/移动应用程序。

注解：如果您不关心所有技术细节，请随意跳至摘要。

针对此类客户端，设计了 `authorization code` 流程。该流程包含两个物理操作：

- 通过浏览器的前端渠道步骤，所有”交互式“事情都会发生，例如登录页面、同意等。此步骤产生一个代表前端渠道操作结果的授权码。
- 一个后端渠道步骤，其中步骤 1 中的授权码与请求的令牌交换。机密客户端此时需要进行身份验证。

此流程具有以下安全属性：

- 没有数据（除了基本上是随机字符串的授权码）通过浏览器渠道泄露
- 授权码只能使用一次
- 只有在（对于机密客户端——稍后会详细介绍）客户端密钥已知时，授权码才能转换为令牌

这听起来很不错——仍然有一个叫做 [代码替换攻击](#) 的问题。对此，有两种现代缓解技术：

OpenID Connect 混合流程

此流程使用 `code id_token` 的响应类型来向响应添加额外的身份标识。此令牌已签名并受到保护以防止替换。此外，它还通过 `c_hash` 声明包含代码的哈希值。这允许检查您是否确实获得了正确的代码（专家称之为分离签名）。

这解决了问题，但有以下缺点：

- `id_token` 通过前端渠道传输，可能会泄露额外的（个人身份）数据
- 所有缓解步骤（例如加密）都需要由客户端实施。这导致更复杂的客户端库实现。

RFC 7636 - Proof Key for Code Exchange (PKCE)

这实质上为授权码流程引入了每个请求的密钥（请阅读 [此处](#) 的详细信息）。为此，客户端所必须要实现的就是创建一个随机字符串并使用 SHA256 对其进行哈希。

这也解决了替换问题，因为客户端可以证明它是前后渠道上的同一个客户端，并且具有以下附加优势：

- 与混合流程相比，客户端实现非常简单
- 它还解决了公共客户端没有静态密钥的问题
- 不需要额外的前端渠道响应伪像

总结

交互式客户端应使用基于授权码的流程。为了防止代码替换，应该使用混合流程或 PKCE。如果 PKCE 可用，这是解决问题的更简单的方法。

PKCE 已经是对 原生 应用程序和 SPAs 的官方推荐——随着 ASP.NET Core 3 的发布，OpenID Connect 处理程序也默认支持。

以下是如何配置交互式客户端：

```
var client = new Client
{
    ClientId = "...",

    // 为机密客户端设置客户端密钥
    ClientSecret = { ... },

    // ...或为公共客户端关闭
    RequireClientSecret = false,

    AllowedGrantTypes = GrantTypes.Code,
    RequirePkce = true
};
```

31.3 没有浏览器或输入设备受限的交互式客户端

此授权类型在 RFC 8628 中有详细说明。

此流程将用户身份验证和同意外包给外部设备（例如智能手机）。它通常由没有真正的键盘（如电视、游戏机…）并且可以请求身份和 API 资源的设备使用。

31.4 自定义场景

扩展授权允许使用新的授权类型扩展令牌端点。有关更多详细信息，请参阅此内容。

在某些情况下，客户端需要使用 `IdentityServer` 进行身份验证，例如

- 机密应用程序（又名客户端）在令牌端点请求令牌
- 在内省端点验证参考令牌的 API

为此，您可以将密钥列表分配给客户端或 API 资源。

密钥解析和验证是身份服务器中的一个扩展点，开箱即用，它支持共享秘密以及通过基本身份验证标头或 POST 正文传输共享秘密。

32.1 创建共享密钥

以下代码设置哈希共享密钥：

```
var secret = new Secret("secret".Sha256());
```

这个密钥现在可以分配给 客户端或 `ApiResource`。Notice that both do not only support a single secret, but multiple. This is useful for secret rollover and rotation:

```
var client = new Client
{
    ClientId = "client",
    ClientSecrets = new List<Secret> { secret },
}
```

(下页继续)

(续上页)

```

AllowedGrantTypes = GrantTypes.ClientCredentials,
AllowedScopes =
{
    "api1", "api2"
}
};

```

事实上，您还可以为密钥分配描述和到期日期。描述将用于日志记录，以及强制执行密钥生命周期的到期日期：

```

var secret = new Secret(
    "secret".Sha256(),
    "2016 secret",
    new DateTime(2016, 12, 31));

```

32.2 使用共享密钥进行身份验证

您可以将客户端 ID/密钥组合作为 POST 正文的一部分发送：

```

POST /connect/token

client_id=client1&
client_secret=secret&
...

```

.. 或者作为一个基本的认证头：

```

POST /connect/token

Authorization: Basic xxxxx

...

```

您可以使用以下 C# 代码手动创建基本身份验证标头：

```

var credentials = string.Format("{0}:{1}", clientId, clientSecret);
var headerValue = Convert.ToBase64String(Encoding.UTF8.GetBytes(credentials));

var client = new HttpClient();
client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Basic",
    headerValue);

```


`IdentityModel` 库有名为 `TokenClient` 和 `IntrospectionClient` 的辅助类，它们封装了身份验证和协议消息。

32.3 使用非对称密钥进行身份验证

There are other techniques to authenticate clients, e.g. based on public/private key cryptography. IdentityServer includes support for private key JWT client secrets (see [RFC 7523](#) and [here](#)).

Secret extensibility typically consists of three things:

- a secret definition
- a secret parser that knows how to extract the secret from the incoming request
- a secret validator that knows how to validate the parsed secret based on the definition

Secret parsers and validators are implementations of the `ISecretParser` and `ISecretValidator` interfaces. To make them available to `IdentityServer`, you need to register them with the DI container, e.g.:

```
builder.AddSecretParser<JwtBearerClientAssertionSecretParser>()
builder.AddSecretValidator<PrivateKeyJwtSecretValidator>()
```

Our default private key JWT secret validator expects the full (leaf) certificate as base64 on the secret definition or an ESA/EC JSON web key:

[illegible]

(下页继续)

(续上页)

```
new Secret
{
    Type = IdentityServerConstants.SecretTypes.JsonWebKey,
    Value = "{ 'e': 'AQAB', 'kid': 'ZzAjSnraU3bkWGnnAqLapYGpTyNfLbjbzgAPbbW2GEA',
↪ 'kty': 'RSA', 'n': 'wWwQFtSzeRjjerpEM5Rmqz_
↪ DsNaZ9S1Bw6UbZkDLowuuTCjBWUax0vBMMxdy6XjEEK4Oq9lKMvx9JzjmeJf1knoqSNrox3Ka0rnXpNAz6sATvme8p9mTXyp0
↪ S9NF5QWvpXvBeC4GAJx7QaSw4zrUkrc6XyaAiFnLhQEwKJCwUw4NOqIuYvYp_IXhw-5Ti_icDlZS-
↪ 282PcccnBeOcX7vc21pozibIdmZJKqXNsL1Ibx5Nkx1F1jLnekJAmdaACDjYRLL_
↪ 6n3W4wUp19UvzB1lGtXcJKLLkqB6YDiZNu16OSiSprfmrRXvYmvD8m6Fn15aetgKw' }"
    }
},

AllowedGrantTypes = GrantTypes.ClientCredentials,
AllowedScopes = { "api1", "api2" }
};
```

扩展授权

OAuth 2.0 定义了令牌端点的标准授权类型, 例如 `password`、`authorization_code` 和 `refresh_token`。扩展授权是一种添加对非标准令牌发行方案（如令牌转换、委派或自定义凭据）的支持的方法。

您可以通过实现 `IExtensionGrantValidator` 接口来添加对其他授权类型的支持:

```
public interface IExtensionGrantValidator
{
    /// <summary>
    /// 处理自定义授权请求。
    /// </summary>
    /// <param name="request">验证上下文。</param>
    Task ValidateAsync(ExtensionGrantValidationContext context);

    /// <summary>
    /// 返回此验证器可以处理的授权类型
    /// </summary>
    /// <value>
    /// 授权类型。
    /// </value>
    string GrantType { get; }
}
```

`ExtensionGrantValidationContext` 对象使您可以访问:

- 传入的令牌请求——众所周知的验证值, 以及任何自定义值 (通过 `Raw` 集合)
- 结果——错误或成功

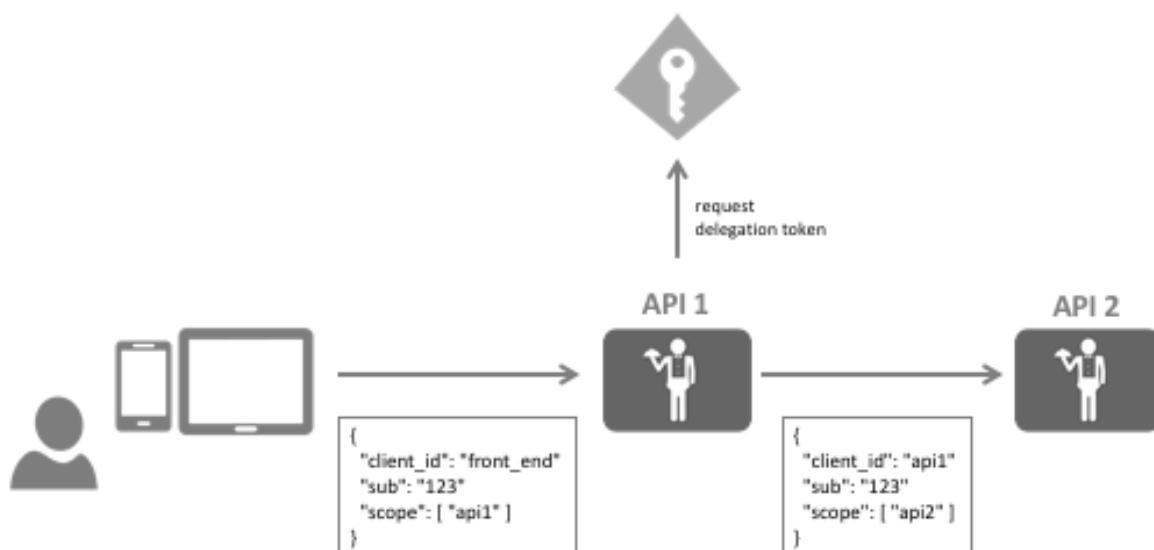
- 自定义响应参数

要注册扩展授权，请将其添加到 DI:

```
builder.AddExtensionGrantValidator<MyExtensionsGrantValidator>();
```

33.1 示例：使用扩展授权的简单委托

想象以下场景——前端客户端使用通过交互流程（例如混合流程）获取的令牌调用中间层 API。这个中间层 API (API 1) 现在想要代表交互式用户调用后端 API (API 2):



换句话说，中间层 API (API 1) 需要一个包含用户身份的访问令牌，但具有后端 API (API 2) 的范围。

注解：您可能听说过 穷人的委托 一词，其中来自前端的访问令牌只是简单地转发到后端。这有一些缺点，例如 API 2 现在必须接受 API 1 范围，这将允许用户直接调用 API 2。此外——您可能希望在令牌中添加一些特定于委托的声明，例如调用路径是通过 API 1 的事实。

实现扩展授权

前端会将令牌发送到 API 1，现在这个令牌需要在 IdentityServer 上与 API 2 的新令牌交换。

在网络上，对交换令牌服务的调用可能如下所示：

```
POST /connect/token

grant_type=delegation&
scope=api2&
```

(下页继续)

(续上页)

```
token=...&
client_id=api1.client
client_secret=secret
```

扩展授权验证器的工作是通过验证传入的令牌来处理该请求，并返回代表新令牌的结果：

```
public class DelegationGrantValidator : IExtensionGrantValidator
{
    private readonly ITokenValidator _validator;

    public DelegationGrantValidator(ITokenValidator validator)
    {
        _validator = validator;
    }

    public string GrantType => "delegation";

    public async Task ValidateAsync(ExtensionGrantValidationContext context)
    {
        var userToken = context.Request.Raw.Get("token");

        if (string.IsNullOrEmpty(userToken))
        {
            context.Result = new GrantValidationResult(TokenRequestErrors.
↪InvalidGrant);
            return;
        }

        var result = await _validator.ValidateAccessTokenAsync(userToken);
        if (result.IsError)
        {
            context.Result = new GrantValidationResult(TokenRequestErrors.
↪InvalidGrant);
            return;
        }

        // 获取用户身份
        var sub = result.Claims.FirstOrDefault(c => c.Type == "sub").Value;

        context.Result = new GrantValidationResult(sub, GrantType);
        return;
    }
}
```

不要忘记向 DI 注册验证器。

注册委托客户端

您需要在 IdentityServer 中进行客户端注册，以允许客户端使用此新扩展授权，例如：

```
var client = new Client
{
    ClientId = "api1.client",
    ClientSecrets = new List<Secret>
    {
        new Secret("secret".Sha256())
    },

    AllowedGrantTypes = { "delegation" },

    AllowedScopes = new List<string>
    {
        "api2"
    }
}
```

调用令牌端点

在 API 1 中，您现在可以自己构建 HTTP 负载，或使用 *IdentityModel* 帮助程序库：

```
public async Task<TokenResponse> DelegateAsync(string userToken)
{
    var client = _httpClientFactory.CreateClient();
    // 或者
    // var client = new HttpClient();

    // 向令牌端点发送自定义授权，返回响应
    return await client.RequestTokenAsync(new TokenRequest
    {
        Address = disco.TokenEndpoint,
        GrantType = "delegation",

        ClientId = "api1.client",
        ClientSecret = "secret",

        Parameters =
        {
            { "scope", "api2" },
            { "token", userToken }
        }
    })
```

(下页继续)

(续上页)

```
    });  
}
```

`TokenResponse.AccessToken` 现在将包含委托访问令牌。

资源所有者密码验证

如果要使用 OAuth 2.0 资源所有者密码凭据授予（又名 password），则需要实现并注册 `IResourceOwnerPasswordValidator` 接口：

```
public interface IResourceOwnerPasswordValidator
{
    /// <summary>
    /// 验证资源所有者密码凭据
    /// </summary>
    /// <param name="context">上下文。</param>
    Task ValidateAsync(ResourceOwnerPasswordValidationContext context);
}
```

在上下文中，您会发现已经解析的协议参数，如 `UserName` 和 `Password`，如果您想查看其他输入数据，还会找到原始请求。

然后你的工作是实现密码验证并相应地在上下文中设置 `Result`。请参阅 *GrantValidationResult* 文档。

由于访问令牌的生命周期有限，刷新令牌允许在没有用户交互的情况下请求新的访问令牌。

以下流支持刷新令牌：授权代码、混合和资源所有者密码凭证流。客户端需要通过将 `AllowOfflineAccess` 设置为 `true`，明确授权请求刷新令牌。

35.1 其他客户端设置

AbsoluteRefreshTokenLifetime 刷新令牌的最长生命周期（以秒为单位）。默认为 2592000 秒 / 30 天。
0 允许刷新令牌，当与 `RefreshTokenExpiration = Sliding` 一起使用时，仅在经过 `SlidingRefreshTokenLifetime` 后过期。

SlidingRefreshTokenLifetime 刷新令牌的滑动生命周期（以秒为单位）。默认为 1296000 秒 / 15

RefreshTokenUsage `ReUse` 刷新令牌时刷新令牌句柄将保持不变

`OneTimeOnly` 刷新令牌时将更新刷新令牌句柄

RefreshTokenExpiration `Absolute` 刷新令牌将在固定时间点（由 `AbsoluteRefreshTokenLifetime` 指定）到期。这是默认设置。

`Sliding` 刷新令牌时，刷新令牌的生命周期将更新（按 `SlidingRefreshTokenLifetime` 中指定的数量）。生命周期不会超过 `AbsoluteRefreshTokenLifetime`。

UpdateAccessTokenClaimsOnRefresh 获取或设置一个值，该值指示是否应在刷新令牌请求时更新访问令牌（及其声明）。

注解： 公共客户端（没有客户端密钥的客户端）应该轮换他们的刷新令牌。将 `RefreshTokenUsage` 设置为 `OneTimeOnly`。

35.2 请求刷新令牌

您可以通过向 `scope` 参数添加一个名为 `offline_access` 的范围来请求刷新令牌。

35.3 使用刷新令牌请求访问令牌

要获取新的访问令牌，请将刷新令牌发送到令牌端点。这将产生一个新的令牌响应，其中包含一个新的访问令牌及其过期时间，并且可能还会产生一个新的刷新令牌，具体取决于客户端配置（见上文）。

```
POST /connect/token

client_id=client&
client_secret=secret&
grant_type=refresh_token&
refresh_token=hdh922
```

(删除了表单编码并添加了换行符以提高可读性)

注解： 您可以使用 [IdentityModel](#) 客户端库从 .NET 代码以编程方式访问令牌端点。有关更多信息，请查看 [IdentityModel 文档](#)。

注解： 刷新令牌必须有效，否则会返回 `invalid_grant` 错误。默认情况下，`refresh_token` 只能使用一次。使用已使用的 `refresh_token` 将导致 `invalid_grant` 错误。

35.4 自定义刷新令牌行为

所有刷新令牌处理都在 `DefaultRefreshTokenService`（这是 `IRefreshTokenService` 接口的默认实现）中实现：

```
public interface IRefreshTokenService
{
    /// <summary>
```

(下页继续)

(续上页)

```
    /// 验证刷新令牌。
    /// </summary>
    Task<TokenValidationResult> ValidateRefreshTokenAsync(string token, Client client);

    /// <summary>
    /// 创建刷新令牌。
    /// </summary>
    Task<string> CreateRefreshTokenAsync(ClaimsPrincipal subject, Token accessToken, Client client);

    /// <summary>
    /// 更新刷新令牌。
    /// </summary>
    Task<string> UpdateRefreshTokenAsync(string handle, RefreshToken refreshToken, Client client);
}
```

刷新令牌处理的逻辑相当复杂，我们不建议从头开始实现接口，除非你完全知道自己在做什么。如果要自定义某些行为，更建议从默认实现派生并首先调用基本检查。

您可能想要做的最常见的定制是如何处理刷新令牌重放。这适用于令牌使用已设置为仅一次的情况，但同一令牌被多次发送。这可能指向刷新令牌的重放攻击，或者指向错误的客户端代码，如逻辑错误或竞争条件。

需要注意的是，刷新令牌永远不会在数据库中删除。一旦被使用，ConsumedTime 属性将被设置。如果收到一个已经被消费的令牌，默认服务将调用一个名为 AcceptConsumedTokenAsync 的虚拟方法。

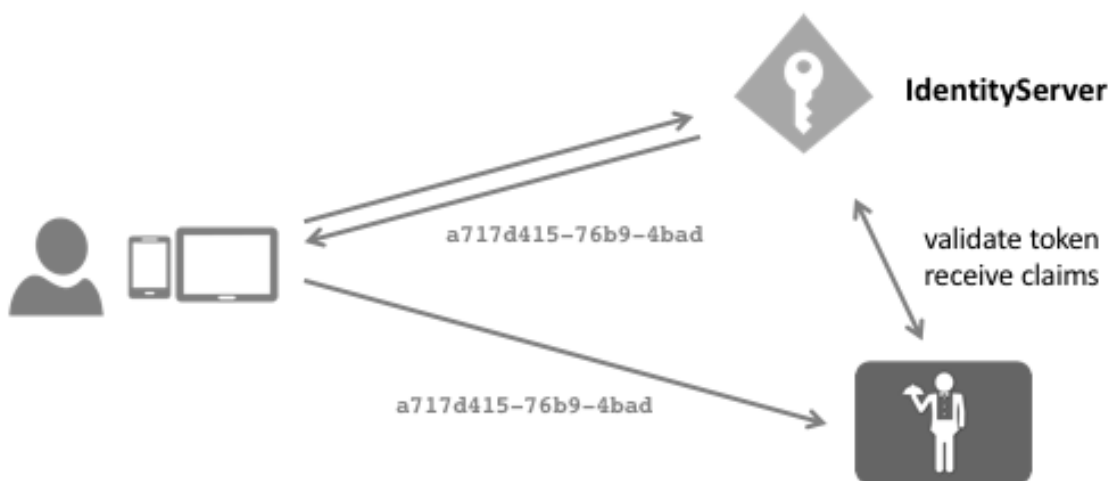
默认实现将拒绝请求，但在这里您可以实现自定义逻辑，如宽限期，或撤销额外的刷新或访问令牌。

引用令牌

访问令牌可以有两种形式 - 独立的或引用的。

JWT 令牌将是一个自包含的访问令牌 - 它是一种受保护的数据结构，具有声明和过期时间。一旦 **API** 了解了密钥材料，它就可以验证自包含的令牌，而无需与发行者进行通信。这使得 **JWT** 难以撤销。它们将一直有效，直到到期。

使用引用令牌时 - **IdentityServer** 会将令牌的内容存储在数据存储中，并且只会将此令牌的唯一标识符返回给客户端。然后，接收此引用的 **API** 必须打开与 **IdentityServer** 的反向通道通信以验证令牌。



您可以使用以下设置切换客户端的令牌类型:

```
client.AccessTokenType = AccessTokenType.Reference;
```

IdentityServer 提供了 OAuth 2.0 自省规范的实现，它允许 API 取消引用令牌。您可以使用我们专用的 [自省处理程序](#) 或使用可以验证 JWT 和引用令牌的 [Identity Server 身份验证处理程序](#)。

自省端点需要身份验证 - 由于自省端点的客户端是一个 API，您可以在 ApiResource 上配置密钥：

```
var api = new ApiResource("api1")
{
    ApiSecrets = { new Secret("secret".Sha256()) }
}
```

有关如何为 API 配置 IdentityServer 身份验证中间件的更多信息，请参阅[此处](#)。

许多授权类型需要在 IdentityServer 中持久化。其中包括授权代码、刷新令牌、参考令牌和记住的用户同意。在 IdentityServer 内部，这些授权的默认存储位于称为持久授权存储的公共存储中。

37.1 持久授权

持久授权是维护授权值的数据类型。它具有以下属性：

Key 存储中持久授权的唯一标识符。

Type 授权的类型。

SubjectId 授予所属的主体 ID。

ClientId 为其创建授权的客户端标识符。

Description 用户分配给授权或被授权设备的描述。

CreationTime 创建授权的日期/时间。

Expiration 授权到期。

ConsumedTime 授权已消耗的日期/时间（见下文）。

Data 授权特定的序列化数据。

注解： Data 属性包含所有值（以及更多）的副本，并且被 IdentityServer 认为是权威的，因此默认情况下，上

述值被认为是信息性的只读值。

存在于存储中，没有 `ConsumedTime` 且仍在 `Expiration` 内的记录，表示授权的有效性。设置这两个值中的任何一个，或者从存储中删除记录，都可以有效地撤销授权。

37.2 授权消费

某些授权类型只能一次性使用（根据定义或配置）。一旦它们被使用，而不是删除记录，`ConsumedTime` 值被设置在数据库中，将它们标记为已被使用。这种软删除允许自定义实现在允许重新使用授权方面具有灵活性（通常在很短的时间内），或用于风险评估和威胁缓解方案（检测到可疑活动）以撤销访问。对于刷新令牌，此类自定义逻辑将在 `IRefreshTokenService` 中执行。

37.3 持久授权服务

直接使用授权存储可能级别太低。因此，提供了一个名为 `IPersistedGrantService` 的更高级别的服务。它将不同的授权类型抽象并聚合到一个概念中，并允许查询和撤销用户的持久授权。

它包含以下 API:

GetAllGrantsAsync 根据主体 ID 获取用户的所有授权。

RemoveAllGrantsAsync 根据主体 id 和可选的客户端 id 和/或会话 id 从存储中删除授权。

访问令牌所有权证明

默认情况下，OAuth 访问令牌称为 *bearer* 令牌。这意味着它们不受客户的约束，任何拥有令牌的人都可以使用它（与现金相比）。

所有权证明 *Proof-of-Possession (short PoP)* 令牌绑定到请求令牌的客户端。如果该令牌泄漏，则其他任何人都无法使用它（与信用卡相比 - 至少在理想世界中如此）。

有关更多历史和动机，请参阅 [此](#) 博客文章。

IdentityServer 通过使用 *Mutual TLS* 机制 支持 PoP 令牌。

IdentityServer 中的双向 TLS 支持允许两个功能：

- 使用 TLS X.509 客户端证书对 IdentityServer 端点进行客户端身份验证
- 使用 TLS X.509 客户端证书将访问令牌绑定到客户端

注解：有关更多信息，请参阅 [“OAuth 2.0 Mutual-TLS 客户端身份验证和证书绑定访问令牌”](#) 规范

设置 MTLS 涉及几个步骤。

39.1 服务器设置

托管层有责任对客户端证书进行实际验证。IdentityServer 然后将使用该信息将证书与客户端相关联，并将证书信息嵌入到访问令牌中。

根据您使用的服务器，这些步骤是不同的。有关更多信息，请参阅 [此](#) 博客文章。

注解：[mkcert](#) 是一个很好的工具，用于为开发目的创建证书。

39.2 ASP.NET Core 设置

根据服务器设置，ASP.NET Core 主机接收客户端证书的方式不同。而对于 IIS 和纯 Kestrel 托管，没有额外的步骤，通常，您在应用程序服务器前面有一个反向代理。

这意味着除了典型的转发头处理之外，您还需要处理包含客户端证书的头。为此，在中间件管道的开头添加对 `app.UseCertificateForwarding()` 的调用。

代理传输证书的确切格式没有标准化，这就是为什么您需要注册一个回调来进行实际的标头解析。Microsoft 文档展示了 Azure Web 应用程序如何工作。

如果您使用的是 Nginx（我们发现它是最灵活的托管选项），您需要在 `ConfigureServices` 中注册以下服务

```
services.AddCertificateForwarding(options => {

    // 标头名称可能会有所不同，具体取决于您的 nginx 配置 options.CertificateHeader = "X-
    SSL-CERT" ;

    options.HeaderConverter = (headerValue) => {

        X509Certificate2 clientCertificate = null;

        if(!string.IsNullOrEmpty(headerValue)) {

            var bytes = Encoding.UTF8.GetBytes(Uri.UnescapeDataString(headerValue));
            clientCertificate = new X509Certificate2(bytes);

        }

        return clientCertificate;

    };

});
```

一旦加载了证书，您还需要设置身份验证处理程序。在这种情况下，我们希望支持自签名证书，因此是 `CertificateType.All` 并且没有吊销检查。这些设置在您的环境中可能会有所不同：

```
services.AddAuthentication()
    .AddCertificate(options =>
    {
        options.AllowedCertificateTypes = CertificateTypes.All;
        options.RevocationMode = X509RevocationMode.NoCheck;
    });
```

39.3 IdentityServer setup

下一步是在 IdentityServer 中启用 MTLS。为此，您需要指定您在最后一步中设置的证书身份验证处理程序的名称（默认为 Certificate），和 MTLS 托管策略。

在 IdentityServer 中，双向 TLS 端点可以通过三种方式配置（假设 IdentityServer 在 `https://identityserver.io` 上运行）：

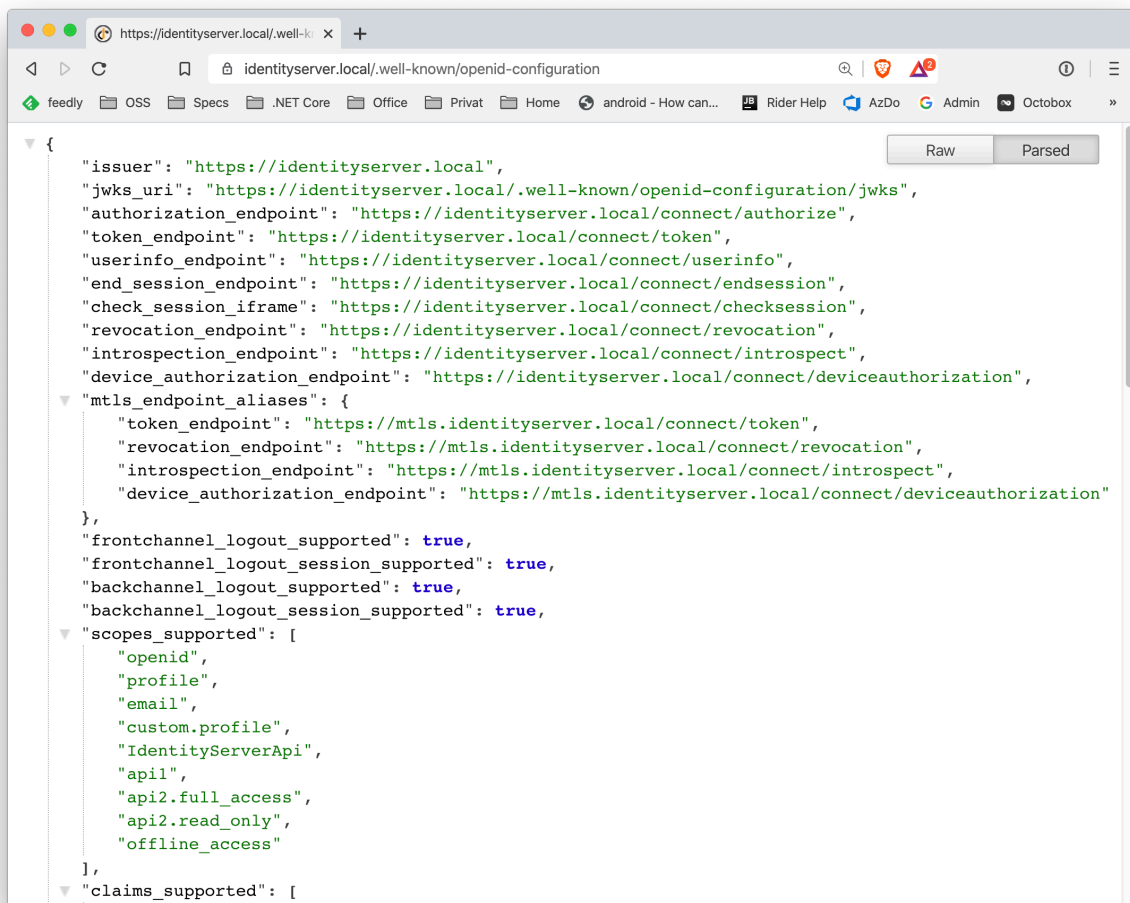
- 基于路径 - 位于路径 `~/connect/mtls` 下的端点，例如 `https://identityserver.io/connect/mtls/token`。
- 基于子域名 - 端点位于主服务器的子域名上，例如 `https://mtls.identityserver.io/connect/token`。
- 基于域名 - 端点位于不同的域名中，例如 `https://identityserver-mtls.io`。

例如：

```
var builder = services.AddIdentityServer(options =>
{
    options.MutualTls.Enabled = true;
    options.MutualTls.ClientCertificateAuthenticationScheme = "Certificate";

    // 使用子域名托管
    options.MutualTls.DomainName = "mtls";
});
```

IdentityServer 的发现文档反映了这些端点：



39.4 客户端认证

客户端可以使用 X.509 客户端证书作为 IdentityServer 中端点的身份验证机制。

为此，您需要将客户端证书与 IdentityServer 中的客户端相关联。使用 *IdentityServer builder* 将服务添加到 DI，其中包含一个默认实现来执行基于指纹或通用名称的操作：

```
builder.AddMutualTlsSecretValidators();
```

最后，对于 *client* 配置 添加到 *ClientSecrets* 集合的机密类型为 *SecretTypes.X509CertificateName*。如果您希望通过证书识别名验证客户端，或者 *SecretTypes.X509CertificateThumbprint* 如果您希望通过证书指纹验证客户端。

例如：


```

new Client
{
    ClientId = "mtls",
    AllowedGrantTypes = GrantTypes.ClientCredentials,
    AllowedScopes = { "api1" }
    ClientSecrets =
    {
        // 基于名称
        new Secret(@"CN=mtls.test, OU=ROO\ballen@roo, O=mkcert development certificate
↪", "mtls.test")
        {
            Type = SecretTypes.X509CertificateName
        },
        // 或基于指纹
        //new Secret("bca0d040847f843c5ee0fa6eb494837470155868", "mtls.test")
        //{
        //    Type = SecretTypes.X509CertificateThumbprint
        //},
    },
}

```

39.4.1 使用客户端证书对 IdentityServer 进行身份验证

编写客户端以连接到 IdentityServer 时, SocketsHttpHandler`` (或 ``HttpClientHandler 如果您使用的是较旧的.NET Framework 版本) 类提供了一种方便的机制来将客户端证书添加到传出请求中。

然后使用 HttpClient 进行 HTTP 调用 (包括使用各种 IdentityModel 扩展方法) 将在 TLS 通道上执行客户端证书认证。

例如:

```

static async Task<TokenResponse> RequestTokenAsync()
{
    var handler = new SocketsHttpHandler();
    var cert = new X509Certificate2("client.p12", "password");
    handler.SslOptions.ClientCertificates = new X509CertificateCollection { cert };

    var client = new HttpClient(handler);

    var disco = await client.GetDiscoveryDocumentAsync(Constants.Authority);
    if (disco.IsError) throw new Exception(disco.Error);

    var response = await client.RequestClientCredentialsTokenAsync(new
↪ClientCredentialsTokenRequest

```

(下页继续)

(续上页)

```
{
    Address = disco
        .TryGetValue(OidcConstants.Discovery.MtlsEndpointAliases)
        .Value<string>(OidcConstants.Discovery.TokenEndpoint)
        .ToString(),

    ClientId = "mtls",
    Scope = "api1"
});

if (response.IsError) throw new Exception(response.Error);
return response;
}
```

39.5 发送方受限访问令牌

每当客户端使用客户端证书向 IdentityServer 进行身份验证时，该证书的指纹将嵌入访问令牌中。

在对 API 进行身份验证时，客户端可以使用 X.509 客户端证书作为发送方受限访问令牌的机制。使用这些受发送方约束的访问令牌要求客户端使用与 IdentityServer 相同的 X.509 客户端证书对 API 进行身份验证。

39.5.1 确认声明

当客户端获得访问令牌并通过双向 TLS 进行身份验证时，IdentityServer 在访问令牌中发出确认声明（或 cnf）。此值是用于向 IdentityServer 进行身份验证的客户端证书指纹的哈希值。

可以在解码访问令牌的屏幕截图中看到此值：



```

C:\ballen\github\IdentityServer\IdentityServer4.Samples\Clients\src\ConsoleMTLSClient>dotnet run
Token response:
{
  "access_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6ImVjZTYzNGJmYTM5ODRhYTE2YTMyYzVmYjY5YWUwYjU3IiwiaWidHlwiOiSlduIn0.eyJ0eXkiOiJpE1NTY0NjU0NjM5ImV4cCI6MTU1NjQ0TA2MywiaXNzIjoiaHR0cHM6Ly9zdHMuaWRlbnRpdHlzZXJ2ZXIubG9jYVwwiLCJhdwQiOiSiaHR0cHM6Ly9zdHMuaWRlbnRpdHlzZXJ2ZXIubG9jYVwwcmVzbnVzIiwiaXNzIjoiSldlcjBjG1lbnRfawQioiJtdGxzIiwic2NvcGU0IjoiSldlcjBmYiOnsiDV0iImYNTYioiJCjZEMkNBNEYwNzk5NDQzMEIENkIGOUQ0ODgzM0EzMDYyN0E1QzI0In19.mje5fxntJLUcDOaDmEwpXdkyzg9JJ015crJ8kwJsrWLJ9yQyUzoeuoa_u8oPy6Qz9H1X4fl3ISrvZ_jmIHvDgeMCLzIW5WlBnz8jVgxXKRSe2bhIkTCiI3CruEfvtu0HonEBu0VuDmqC8uPEStsTQmmUJ608Jx6RBOK63pvh6wrGa98_GcUb-Nv_PUQjWuGS7Rc3LY5q0cKMlSvYVJrzz0PlEkPjeg6N2Re39y87tRmsk2iDrUf1ly2CrNgcMg3ENor60iz29KiyFxxwVw_TVyiAVxk8-sBMFAxBVJR_UkwXJDIz_SQe2w5Xps7RxcvIDhjND1i6JZgE1AHGh_E0w",
  "expires_in": 3600,
  "token_type": "Bearer"
}

Access Token (decoded):
{
  "alg": "RS256",
  "kid": "ece634bfa3984aa16a32c5fb69aa0b57",
  "typ": "JWT"
}
{
  "nbf": 1556465463,
  "exp": 1556469063,
  "iss": "https://sts.identityserver.local",
  "aud": [
    "https://sts.identityserver.local/resources",
    "api1"
  ],
  "client_id": "mTLS",
  "scope": [
    "api1"
  ],
  "cnf": {
    "x5t#S256": "BF6E2CA4F07994430B86BF9D48833A33F27A5C24"
  }
}

```

然后，API 将使用此值来确保 API 中使用的客户端证书与访问令牌中的确认值匹配。

39.5.2 在 API 中验证和接受客户端证书

如上所述，对于 IdentityServer 中的客户端身份验证，在 API 中，Web 服务器应在 TLS 层执行客户端证书验证。

此外，API 托管应用程序将需要一种机制来接受客户端证书，以便获取指纹以执行确认声明验证。下面是如何为访问令牌和客户端证书配置 ASP.NET Core 中的 API 的示例：

```

services.AddAuthentication("token")
    .AddIdentityServerAuthentication("token", options =>
    {
        options.Authority = "https://identityserver.io";
        options.ApiName = "api1";
    })
    .AddCertificate(options =>
    {
        options.AllowedCertificateTypes = CertificateTypes.All;
    });

```

最后，需要一种在身份验证中间件之后运行的机制来验证客户端证书并将指纹与访问令牌中的 cnf 进行比较。

下面是一个检查声明的简单中间件：

```

public class ConfirmationValidationMiddlewareOptions
{
    public string CertificateSchemeName { get; set; } =
↪CertificateAuthenticationDefaults.AuthenticationScheme;
    public string JwtBearerSchemeName { get; set; } = JwtBearerDefaults.
↪AuthenticationScheme;
}

// 此中间件根据当前客户端的 X.509 客户端证书的指纹验证 cnf 声明 (如果存在)
public class ConfirmationValidationMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ConfirmationValidationMiddlewareOptions _options;

    public ConfirmationValidationMiddleware(RequestDelegate next,
↪ConfirmationValidationMiddlewareOptions options = null)
    {
        _next = next;
        _options = options ?? new ConfirmationValidationMiddlewareOptions();
    }

    public async Task Invoke(HttpContext ctx)
    {
        if (ctx.User.Identity.IsAuthenticated)
        {
            var cnfJson = ctx.User.FindFirst("cnf")?.Value;
            if (!String.IsNullOrEmpty(cnfJson))
            {
                var certResult = await ctx.AuthenticateAsync(_options.
↪CertificateSchemeName);
                if (!certResult.Succeeded)
                {
                    await ctx.ChallengeAsync(_options.CertificateSchemeName);
                    return;
                }

                var certificate = await ctx.Connection.GetClientCertificateAsync();
                var thumbprint = Base64UrlTextEncoder.Encode(certificate.
↪GetCertHash(HashAlgorithmName.SHA256));

                var cnf = JObject.Parse(cnfJson);
                var sha256 = cnf.Value<string>("x5t#S256");

                if (String.IsNullOrEmpty(sha256) ||

```

(下页继续)

(续上页)

```

        !thumbprint.Equals(sha256, StringComparison.Ordinal))
    {
        await ctx.ChallengeAsync(_options.JwtBearerSchemeName);
        return;
    }
}

await _next(ctx);
}

```

以下是 API 的示例管道:

```

app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedHeaders.
↪XForwardedProto
});

app.UseCertificateForwarding();
app.UseRouting();
app.UseAuthentication();

app.UseMiddleware<ConfirmationValidationMiddleware>(new
↪ConfirmationValidationMiddlewareOptions
{
    CertificateSchemeName = CertificateAuthenticationDefaults.AuthenticationScheme,
    JwtBearerSchemeName = "token"
});

app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
});

```

一旦上述中间件成功, 那么调用者就已经通过发送者约束的访问令牌进行了身份验证。

39.5.3 自省和确认声明

当访问令牌是 JWT 时，确认声明作为声明包含在令牌中。使用引用令牌时，访问令牌所代表的声明必须通过自省获得。IdentityServer 中的自省端点将为通过双向 TLS 获得的引用令牌返回一个 cnf 声明。

39.6 临时客户端证书

您还可以使用 IdentityServer MTLs 支持来创建受发送方约束的访问令牌，而无需使用客户端证书进行客户端身份验证。这对于您已经拥有不想更改的客户端机密的情况非常有用，例如共享秘密，或更好的私钥 JWT。

尽管如此，如果存在客户端证书，则可以将确认声明嵌入到传出访问令牌中。并且只要客户端使用相同的客户端证书请求令牌并调用 API，这将为您提供所需的所有权证明属性。

为此，在选项中启用以下设置：

```
var builder = services.AddIdentityServer(options =>
{
    // 其他设置

    options.MutualTls.AlwaysEmitConfirmationClaim = true;
});
```

39.6.1 使用临时证书请求令牌

在这种情况下，客户端使用 某些客户端机密（以下示例中的共享机密），但将附加客户端证书附加到令牌请求。由于此证书不需要在令牌服务处与客户端相关联，因此可以即时创建：

```
static X509Certificate2 CreateClientCertificate(string name)
{
    X500DistinguishedName distinguishedName = new X500DistinguishedName($"CN={name}");

    using (RSA rsa = RSA.Create(2048))
    {
        var request = new CertificateRequest(distinguishedName, rsa,
        ↪ HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);

        request.CertificateExtensions.Add(
            new X509KeyUsageExtension(X509KeyUsageFlags.DataEncipherment |
        ↪ X509KeyUsageFlags.KeyEncipherment | X509KeyUsageFlags.DigitalSignature , false));

        request.CertificateExtensions.Add(
            new X509EnhancedKeyUsageExtension(
                new OidCollection { new Oid("1.3.6.1.5.5.7.3.2") }, false));
    }
}
```

(下页继续)

(续上页)

```

        return request.CreateSelfSigned(new DateTimeOffset(DateTime.UtcNow.AddDays(-
↪1)), new DateTimeOffset(DateTime.UtcNow.AddDays(10)));
    }
}

```

然后除了已经设置的客户端密钥之外，还使用此客户端证书：

```

static async Task<TokenResponse> RequestTokenAsync()
{
    var client = new HttpClient(GetHandler(ClientCertificate));

    var disco = await client.GetDiscoveryDocumentAsync("https://identityserver.local
↪");
    if (disco.IsError) throw new Exception(disco.Error);

    var endpoint = disco
        .TryGetValue(OidcConstants.Discovery.MtlsEndpointAliases)
        .Value<string>(OidcConstants.Discovery.TokenEndpoint)
        .ToString();

    var response = await client.RequestClientCredentialsTokenAsync(new
↪ClientCredentialsTokenRequest
    {
        Address = endpoint,

        ClientId = "client",
        ClientSecret = "secret",
        Scope = "api1"
    });

    if (response.IsError) throw new Exception(response.Error);
    return response;
}

static SocketsHttpHandler GetHandler(X509Certificate2 certificate)
{
    var handler = new SocketsHttpHandler();
    handler.SslOptions.ClientCertificates = new X509CertificateCollection {
↪certificate };

    return handler;
}

```


授权请求对象

您可以将授权请求的参数作为单独的查询字符串键/值对提供，而不是将它们打包在签名的 JWT 中。这使得参数可以防篡改，并且您可以对已在前端通道上的客户端进行身份验证。

您可以通过值或通过引用授权端点来传输它们 - 有关更多详细信息，请参阅 [规范](#)。

IdentityServer 要求对请求 JWT 进行签名。我们支持 X509 证书和 JSON Web 密钥，例如：

```
var client = new Client
{
    ClientId = "foo",

    // 将此设置为 true 以仅接受签名请求
    RequireRequestObject = true,

    ClientSecrets =
    {
        new Secret
        {
            // X509 证书 base64 编码
            Type = IdentityServerConstants.SecretTypes.X509CertificateBase64,
            Value = Convert.ToBase64String(cert.Export(X509ContentType.Cert))
        },
        new Secret
        {
            // RSA 密钥作为 JWK
```

(下页继续)

(续上页)

```

        Type = IdentityServerConstants.SecretTypes.JsonWebKey,
        Value =
            "{ 'e': 'AQAB', 'kid': 'ZzAjSnraU3bkWGnnAqLapYGpTyNfLbjbzgAPbbW2GEA', 'kty'
↪ ': 'RSA', 'n': 'wWwQFtSzeRjjerpEM5Rmqz_
↪ DsNaZ9S1Bw6UbZkDLowuuTCjBWUax0vBMMxdy6XjEEK4Oq9lKMvx9JzjmeJf1knoqSNrox3Ka0rnXpNAz6sATvme8p9mTXyp0
↪ S9NF5QWvpXvBeC4GAJx7QaSw4zrUkrc6XyaAiFnLhQEwKJCwUw4NOqIuYvYp_IXhw-5Ti_icDlZS-
↪ 282PcccnBeOcX7vc21pozibIdmZJKqXNsL1Ibx5Nkx1F1jLnkJAmdaACDjYRLL_
↪ 6n3W4wUp19UvzB1lGtXcJKLLkqB6YDiZNu16OSiSprfmrRXvYmvD8m6Fn15aetgKw' }"
    }
}
}

```

注解： Microsoft.IdentityModel.Tokens.JsonWebKeyConverter 有各种帮助程序将密钥转换为 JWK

40.1 通过引用传递请求 JWTs

如果使用 `request_uri` 参数，IdentityServer 将发出一个传出的 HTTP 调用以从指定的 URL 获取 JWT。

您可以自定义用于此传出连接的 HTTP 客户端，例如添加缓存或重试逻辑（例如通过 Polly 库）：

```

builder.AddJwtRequestUriHttpClient(client =>
{
    client.Timeout = TimeSpan.FromSeconds(30);
})
.AddTransientHttpErrorPolicy(policy => policy.WaitAndRetryAsync(new[]
{
    TimeSpan.FromSeconds(1),
    TimeSpan.FromSeconds(2),
    TimeSpan.FromSeconds(3)
}));

```

注解： 默认情况下禁用请求 URI 处理。在 IdentityServer Options <refOptions> 下的“Endpoints”上启用。另请参阅 JAR 规范 中的安全注意事项。

40.2 访问请求对象数据

您可以通过两种方式从请求对象中访问经过验证的数据

- 无论您在哪里可以访问 `ValidatedAuthorizeRequest`, `RequestObjectValues` 字典都会保存值
- 在 UI 代码中, 您可以调用 `IIdentityServerInteractionService.GetAuthorizationContextAsync`, 生成的 `AuthorizationRequest` 对象也包含 `RequestObjectValues` 字典

自定义令牌请求验证和颁发

您可以在令牌端点处将自定义代码作为令牌颁发管道的一部分运行。这允许，例如

- 添加额外的验证逻辑
- 动态更改某些参数（如令牌寿命）

为此，实现（并注册）“`ICustomTokenRequestValidator`” 接口：

```
/// <summary>
/// 允许在令牌请求中插入自定义验证逻辑
/// </summary>
public interface ICustomTokenRequestValidator
{
    /// <summary>
    /// 令牌请求的自定义验证逻辑。
    /// </summary>
    /// <param name="context">上下文。</param>
    /// <returns>
    /// 验证结果
    /// </returns>
    Task ValidateAsync(CustomTokenRequestValidationContext context);
}
```

上下文对象使您可以访问：

- 添加自定义响应参数
- 返回错误和错误描述

- 修改请求参数，例如访问令牌生命周期和类型、客户端声明和确认方法

您可以使用配置构建器上的 `AddCustomTokenRequestValidator` 扩展方法注册验证器的实现。

IdentityServer 中的许多端点将通过基于 JavaScript 的客户端的 Ajax 调用进行访问。鉴于 IdentityServer 很可能托管在与这些客户端不同的源上，这意味着 [跨源资源共享 \(CORS\)](#) 将需要进行配置。

42.1 基于客户端的 CORS 配置

配置 CORS 的一种方法是在客户端配置上使用 AllowedCorsOrigins 集合。只需将客户端的源添加到集合中，IdentityServer 中的默认配置将参考这些值以允许来自源的跨源调用。

注解： 确保在配置 CORS 时使用源（而不是 URL）。例如：`https://foo:123/` 是一个 URL，而 `https://foo:123` 是一个来源。

如果您使用我们提供的“内存中”或基于 EF 的客户端配置，则将使用此默认 CORS 实现。如果您定义自己的 IClientStore，那么您将需要实现您自己的自定义 CORS 策略服务（见下文）。

42.2 自定义 Cors 策略服务

IdentityServer 允许托管应用程序实现 `ICorsPolicyService` 以完全控制 CORS 策略。

要实现的单一方法是：`Task<bool> IsOriginAllowedAsync(string origin)`。如果允许 *origin*，则返回 *true*，否则返回 *false*。

实现后，只需在 DI 中注册实现，IdentityServer 就会使用您的自定义实现。

DefaultCorsPolicyService

如果您只是希望对一组允许的来源进行硬编码，那么您可以使用一个名为 `DefaultCorsPolicyService` 的预构建 `ICorsPolicyService` 实现。这将在 DI 中配置为单例，并使用其 `AllowedOrigins` 集合进行硬编码，或将标志 `AllowAll` 设置为 `true` 以允许所有来源。例如，在 `ConfigureServices` 中：

```
services.AddSingleton<ICorsPolicyService>((container) => {  
    var logger = container.GetRequiredService<ILogger<DefaultCorsPolicyService>>();  
    return new DefaultCorsPolicyService(logger) {  
        AllowedOrigins = { "https://foo", "https://bar" }  
    };  
});
```

注解： 谨慎使用 `AllowAll`。

42.3 将 IdentityServer CORS 策略与 ASP.NET Cores CORS 策略混合使用

IdentityServer 使用来自 ASP.NET Core 的 CORS 中间件来提供其 CORS 实现。托管 IdentityServer 的应用程序可能也需要 CORS 用于其自己的自定义端点。一般来说，两者应该在同一个应用程序中一起工作。

您的代码应使用 ASP.NET Core 中记录的 CORS 功能，而不考虑 IdentityServer。这意味着您应该像往常一样定义策略并注册中间件。如果您的应用程序在 `ConfigureServices` 中定义了策略，那么这些策略应该在您使用它们的相同位置继续工作（在您配置 CORS 中间件的位置或在您的控制器代码中使用 `MVC.EnableCors` 属性的位置）。相反，如果您在使用 CORS 中间件时定义内联策略（通过策略构建器回调），那么它也应该继续正常工作。

使用 ASP.NET Core CORS 服务和 IdentityServer 之间可能存在冲突的一种情况是，如果您决定创建自定义 `ICorsPolicyProvider`。鉴于 ASP.NET Core 的 CORS 服务和中间件的设计，IdentityServer 实现了自己的自定义 `ICorsPolicyProvider` 并将其注册到 DI 系统中。幸运的是，IdentityServer 实现旨在使用装饰器模式来包装任何已在 DI 中注册的现有 `ICorsPolicyProvider`。这意味着您还可以实现 `ICorsPolicyProvider`，但它只需要在 DI 中的 IdentityServer 之前注册（例如在 `ConfigureServices` 中）。

发现文档可以在 <https://baseaddress/.well-known/openid-configuration> 找到。它包含有关 IdentityServer 的端点、密钥材料和功能的信息。

默认情况下，所有信息都包含在发现文档中，但通过使用配置选项，您可以隐藏个别部分，例如：

```
services.AddIdentityServer(options =>
{
    options.Discovery.ShowIdentityScopes = false;
    options.Discovery.ShowApiScopes = false;
    options.Discovery.ShowClaims = false;
    options.Discovery.ShowExtensionGrantTypes = false;
});
```

43.1 扩展发现

您可以向发现文档添加自定义条目，例如：

```
services.AddIdentityServer(options =>
{
    options.Discovery.CustomEntries.Add("my_setting", "foo");
    options.Discovery.CustomEntries.Add("my_complex_setting",
        new
        {
```

(下页继续)

(续上页)

```
        foo = "foo",  
        bar = "bar"  
    });  
});
```

当您添加以 ~/ 开头的自定义值时，它将扩展为 IdentityServer 基地址下方的绝对路径，例如：

```
options.Discovery.CustomEntries.Add("my_custom_endpoint", "~/custom");
```

如果您想完全控制发现（和 jwks）文档的呈现，您可以实现 IDiscoveryResponseGenerator 接口（或派生自我们的默认实现）。

CHAPTER 44

添加更多 API 端点

向托管 IdentityServer 的应用程序添加额外的 API 端点是一种常见的情况。这些端点通常由 IdentityServer 本身保护。

对于简单的场景，我们给你一些帮助。请参阅高级部分以了解更多内部管道。

注解： 您可以通过使用我们的 IdentityServerAuthentication 处理程序或微软的 JwtBearer 处理程序来实现相同的目的。但不建议这样做，因为它需要更多配置并创建对可能导致未来更新冲突的外部库的依赖。

首先将您的 API 注册为 ApiResource，例如：

```
public static IEnumerable<ApiResource> Apis = new List<ApiResource>
{
    // 本地API
    new ApiResource(IdentityServerConstants.LocalApi.ScopeName),
};
```

.. 并让您的客户访问此 API，例如：

```
new Client
{
    // 省略其余部分
    AllowedScopes = { IdentityServerConstants.LocalApi.ScopeName },
}
```

注解: `IdentityServerConstants.LocalApi.ScopeName` 的值为 `IdentityServerApi`。

要为本地 API 启用令牌验证，请将以下内容添加到您的 `IdentityServer` 启动中：

```
services.AddLocalApiAuthentication();
```

为了保护 API 控制器，使用 `LocalApi.PolicyName` 策略用 `Authorize` 属性修饰它：

```
[Route("localApi")]
[Authorize(LocalApi.PolicyName)]
public class LocalApiController : ControllerBase
{
    public IActionResult Get()
    {
        // 省略
    }
}
```

然后，授权客户端可以为 `IdentityServerApi` 范围请求令牌并使用它来调用 API。

44.1 发现

如果需要，您还可以将端点添加到发现文档中，例如像这样：

```
services.AddIdentityServer(options =>
{
    options.Discovery.CustomEntries.Add("local_api", "~/localapi");
})
```

44.2 高级

在幕后，`AddLocalApiAuthentication` 助手做了几件事：

- 添加一个身份验证处理程序，它使用 `IdentityServer` 的内置令牌验证引擎来验证传入的令牌（此处理程序的名称是 `IdentityServerAccessToken` 或 `IdentityServerConstants.LocalApi.AuthenticationScheme`）
- 将身份验证处理程序配置为在值为 `IdentityServerApi` 的访问令牌中要求范围声明
- 设置一个授权策略来检查值 `IdentityServerApi` 的范围声明

这涵盖了最常见的场景。您可以通过以下方式自定义此行为：

- 通过调用 `services.AddAuthentication().AddLocalApi(...)` 自己添加身份验证处理程序
 - 通过这种方式，您可以自己指定所需的范围名称，或者（完全不指定范围）接受来自当前 IdentityServer 实例的任何令牌
- 使用自定义策略或代码在您的控制器中进行您自己的范围验证/授权，例如：

```
services.AddAuthorization(options =>
{
    options.AddPolicy(IdentityServerConstants.LocalApi.PolicyName, policy =>
    {
        policy.AddAuthenticationSchemes(IdentityServerConstants.LocalApi.
↪AuthenticationScheme);
        policy.RequireAuthenticatedUser();
        // 定制需求
    });
});
```

44.3 声明转换

您可以提供回调以在验证后转换传入令牌的声明。要么使用辅助方法，例如：

```
services.AddLocalApiAuthentication(principal =>
{
    principal.Identities.First().AddClaim(new Claim("additional_claim", "additional_
↪value"));

    return Task.FromResult(principal);
});
```

…或者如果您手动添加身份验证处理程序，则在选项上实现事件。

除了对 OpenID Connect 和 OAuth 2.0 的内置支持之外，IdentityServer4 还允许添加对其他协议的支持。

您可以添加这些额外的协议端点作为中间件或使用例如 MVC 控制器。在这两种情况下，您都可以访问 ASP.NET Core DI 系统，该系统允许重用我们的内部服务，例如访问客户端定义或密钥材料。

添加 WS-Federation 支持的示例可以在 [这里](#) 找到。

45.1 典型的身份验证工作流程

身份验证请求通常是这样工作的：

- 身份验证请求到达协议端点
- 协议端点进行输入验证
- 重定向到登录页面，返回 URL 设置回协议端点（如果用户是匿名的）
 - 通过 `IIdentityServerInteractionService` 访问当前请求的详细信息
 - 用户身份验证（本地或通过外部身份验证中间件）
 - 用户登录
 - 重定向回协议端点
- 创建协议响应（令牌创建并重定向回客户端）

45.2 有用的 IdentityServer 服务

为了实现上述工作流程，需要与 IdentityServer 的一些交互点。

访问配置并重定向到登录页面

您可以通过注入 IdentityServerOptions 来访问 IdentityServer 配置类到您的代码中。这，例如具有登录页面的配置路径:

```
var returnUrl = Url.Action("Index");
returnUrl = returnUrl.AddQueryString(Request.QueryString.Value);

var loginUrl = _options.UserInteraction.LoginUrl;
var url = loginUrl.AddQueryString(_options.UserInteraction.LoginReturnUrlParameter, returnUrl);

return Redirect(url);
```

登录页面与当前协议请求的交互

IIdentityServerInteractionService 支持将协议返回 URL 转换为经过解析和验证的上下文对象:

```
var context = await _interaction.GetAuthorizationContextAsync(returnUrl);
```

默认情况下，交互服务仅理解 OpenID Connect 协议消息。要扩展支持，您可以编写自己的 IReturnUrlParser:

```
public interface IReturnUrlParser
{
    bool IsValidReturnUrl(string returnUrl);
    Task<AuthorizationRequest> ParseAsync(string returnUrl);
}
```

.. 然后在 DI 中注册解析器:

```
builder.Services.AddTransient<IReturnUrlParser, WsFederationReturnUrlParser>();
```

这允许登录页面获取诸如客户端配置和其他协议参数之类的信息。

访问用于创建协议响应的配置和密钥材料

通过将 IKeyMaterialService 注入您的代码，您可以访问配置的签名凭证和验证密钥:

```
var credential = await _keys.GetSigningCredentialsAsync();
var key = credential.Key as Microsoft.IdentityModel.Tokens.X509SecurityKey;

var descriptor = new SecurityTokenDescriptor
```

(下页继续)

(续上页)

```
{
    AppliesToAddress = result.Client.ClientId,
    Lifetime = new Lifetime(DateTime.UtcNow, DateTime.UtcNow.AddSeconds(result.Client.
↪IdentityTokenLifetime)),
    ReplyToAddress = result.Client.RedirectUris.First(),
    SigningCredentials = new X509SigningCredentials(key.Certificate, result.
↪RelyingParty.SignatureAlgorithm, result.RelyingParty.DigestAlgorithm),
    Subject = outgoingSubject,
    TokenIssuerName = _contextAccessor.HttpContext.GetIdentityServerIssuerUri(),
    TokenType = result.RelyingParty.TokenType
};
```


`IdentityServerTools` 类是为 `IdentityServer` 编写可扩展性代码时可能需要的有用内部工具的集合。要使用它，请将其注入您的代码中，例如控制器：

```
public MyController(IdentityServerTools tools)
{
    _tools = tools;
}
```

`IssueJwtAsync` 方法允许使用 `IdentityServer` 令牌创建引擎创建 JWT 令牌。`IssueClientJwtAsync` 是一个更简单的版本，用于为服务器到服务器的通信创建令牌（例如，当您必须从代码中调用受 `IdentityServer` 保护的 API 时）：

```
public async Task<IActionResult> MyAction()
{
    var token = await _tools.IssueClientJwtAsync(
        clientId: "client_id",
        lifetime: 3600,
        audiences: new[] { "backend.api" });

    // 更多的代码
}
```


CHAPTER 47

发现端点

发现端点可用于检索有关您的 IdentityServer 的元数据 - 它返回诸如发行者名称、密钥材料、支持的范围等信息。有关更多详细信息，请参阅 [规范](#)。发现端点可通过相对于基地址的 `/.well-known/openid-configuration` 获得，例如：

```
https://demo.identityserver.io/.well-known/openid-configuration
```

注解： 您可以使用 IdentityModel 客户端库从 .NET 代码以编程方式访问发现端点。有关更多信息，请查看 IdentityModel [文档](#)。

授权端点

授权端点可用于通过浏览器请求令牌或授权码。此过程通常涉及最终用户的身份验证和可选的同意。

注解： IdentityServer 支持 OpenID Connect 和 OAuth 2.0 授权请求参数的子集。有关完整列表，请参阅[此处](https://openid.net/specs/openid-connect-core-1_0.html#AuthRequest) `<https://openid.net/specs/openid-connect-core-1_0.html#AuthRequest>`。

client_id 客户端的标识符（必需）。

request 您可以提供一个子集或全部作为 JWT，而不是将所有参数作为单独的查询字符串参数提供

request_uri 包含请求参数的预打包 JWT 的 URL

scope 一个或多个已注册的范围（必需）

redirect_uri 必须与该客户端允许的重定向 URI 之一完全匹配（必需）

response_type id_token 请求身份令牌（仅允许身份范围）

token 请求访问令牌（仅允许资源范围）

id_token token 请求身份令牌和访问令牌

code 请求授权码

code id_token 请求授权码和身份令牌

code id_token token 请求授权码、身份令牌和访问令牌

response_mode form_post 将令牌响应作为表单发布而不是片段编码重定向发送（可选）

state identityserver 将在令牌响应上回显状态值，这是用于客户端和提供者之间的往返状态，关联请求和响应以及 CSRF/重放保护。（推荐的）

nonce identityserver 将回显身份令牌中的 nonce 值，这是为了重放保护）

对于通过隐式授权的身份令牌是必需的。

prompt none 请求期间不会显示 UI。如果这是不可能的（例如，因为用户必须登录或同意），则返回错误 login 即使用户已经登录并且有一个有效的会话，也会显示登录 UI

code_challenge 发送 PKCE 的代码质询

code_challenge_method plain 表示 challenge 使用纯文本（不推荐）S256 表示 challenge 是用 SHA256 散列的

login_hint 可用于在登录页面预填用户名字段

ui_locales 提供有关登录 UI 所需显示语言的提示

max_age 如果用户的登录会话超过最大期限（以秒为单位），将显示登录 UI

acr_values 允许传入额外的身份验证相关信息 - identityserver 特殊情况下以下专有 acr_values:

idp:name_of_idp 绕过登录/主域屏幕并将用户直接转发到选定的身份提供者（如果每个客户端配置允许）

tenant:name_of_tenant 可用于将租户名称传递给登录 UI

例子

```
GET /connect/authorize?
  client_id=client1&
  scope=openid email api1&
  response_type=id_token token&
  redirect_uri=https://myapp/callback&
  state=abc&
  nonce=xyz
```

（删除了 URL 编码，并添加了换行符以提高可读性）

注解： 您可以使用 [IdentityModel](#) 客户端库以编程方式创建授权请求.NET 代码。有关更多信息，请查看 [IdentityModel](#) 文档。

令牌端点

令牌端点可用于以编程方式请求令牌。它支持 `password`、`authorization_code`、`client_credentials`、`refresh_token` 和 `urn:ietf:params:oauth:grant-type:device_code` 授权类型。此外，可以扩展令牌端点以支持扩展授权类型。

注解： IdentityServer 支持 OpenID Connect 和 OAuth 2.0 令牌请求参数的子集。有关完整列表，请参阅[此处](http://openid.net/specs/openid-connect-core-1_0.html#TokenRequest) `<http://openid.net/specs/openid-connect-core-1_0.html#TokenRequest>`‘_。

client_id 客户端标识符（必需 - 在正文中或作为授权标头的一部分。）

client_secret 客户端机密在帖子正文中，或作为基本身份验证标头。可选的。

grant_type `authorization_code`、`client_credentials`、`password`、`refresh_token`、`urn:ietf:params:oauth:grant-type:device_code` 或自定义

scope 一个或多个注册范围。如果未指定，则将发布所有明确允许的范围的令牌。

redirect_uri `authorization_code` 授权类型所需

code 授权码（“`authorization_code`” 授权类型需要）

code_verifier PKCE 证明密钥

username 资源所有者用户名（`password` 授权类型需要）

password 资源所有者密码（`password` 授权类型需要）

acr_values 允许为 `password` 授权类型传递额外的身份验证相关信息 - identityserver 特殊情况，以下专有 `acr_values`：

`idp:name_of_idp` 绕过登录/主域屏幕并将用户直接转发到选定的身份提供者（如果每个客户端配置允许）

`tenant:name_of_tenant` 可用于将租户名称传递给令牌端点

refresh_token 刷新令牌（refresh_token 授予类型所需）

device_code 设备代码（urn:ietf:params:oauth:grant-type:device_code 授权类型需要）

49.1 例子

```
POST /connect/token
CONTENT-TYPE application/x-www-form-urlencoded

client_id=client1&
client_secret=secret&
grant_type=authorization_code&
code=hdh922&
redirect_uri=https://myapp.com/callback
```

（删除了表单编码并添加了换行符以提高可读性）

注解： 您可以使用 [IdentityModel](#) 客户端库从.NET 代码以编程方式访问令牌端点。有关更多信息，请查看 [IdentityModel 文档](#)。

用户信息端点

用户信息端点可用于检索有关用户的身份信息（请参阅 [规范](#)）。

调用者需要发送代表用户的有效访问令牌。根据授予的范围，用户信息端点将返回映射的声明（至少需要 `openid` 范围）。

50.1 例子

```
GET /connect/userinfo
Authorization: Bearer <access_token>
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "sub": "248289761001",
  "name": "Bob Smith",
  "given_name": "Bob",
  "family_name": "Smith",
  "role": [
    "user",
    "admin"
  ]
}
```

注解: 您可以使用 [IdentityModel](#) 客户端库从.NET 代码以编程方式访问 userinfo 端点。有关更多信息, 请查看 [IdentityModel 文档](#)。

设备授权端点

设备授权端点可用于请求设备和用户代码。该端点用于启动设备流授权过程。

注解： 结束会话端点的 URL 可通过发现文档 获得。

client_id 客户端标识符（必需）

client_secret 客户端密钥在 POST body 中，或作为基本身份验证标头。可选的。

scope 一个或多个已注册的范围。如果未指定，则将发布所有明确允许的范围的令牌。

51.1 例子

```
POST /connect/deviceauthorization

client_id=client1&
client_secret=secret&
scope=openid api1
```

（删除了表单编码并添加了换行符以提高可读性）

注解： 您可以使用 `IdentityModel` 客户端库从 .NET 代码以编程方式访问设备授权端点。有关更多信息，请查

看 [IdentityModel 文档](#)。

自省端点

自省端点是 [RFC 7662](#) 的实现。

它可用于验证参考令牌（或 JWT，如果消费者不支持适当的 JWT 或加密库）。自省端点需要身份验证 - 由于自省端点的客户端是一个 API，您可以在 ApiResource 上配置密钥。

52.1 例子

```
POST /connect/introspect
Authorization: Basic xxxyyy

token=<token>
```

成功的响应将返回状态代码 200 以及活动或非活动令牌:

```
{
  "active": true,
  "sub": "123"
}
```

未知或过期的令牌将被标记为不活动:

```
{
  "active": false,
}
```

无效请求将返回 400，未经授权请求将返回 401。

注解： 您可以使用 [IdentityModel](#) 客户端库从 .NET 代码以编程方式访问自省端点。有关更多信息，请查看 [IdentityModel](#) 文档。

撤销端点

此端点允许撤销访问令牌（仅限参考令牌）和刷新令牌。它实现了令牌撤销规范‘[RFC 7009](https://tools.ietf.org/html/rfc7009)’[<https://tools.ietf.org/html/rfc7009>](https://tools.ietf.org/html/rfc7009)’。

token 要撤销的令牌（必需）

token_type_hint access_token 或 refresh_token （可选）

53.1 例子

```
POST /connect/revocation HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

token=45ghiukldjahdnhzdauz&token_type_hint=refresh_token
```

注解： 您可以使用 [IdentityModel](#) 客户端库从.NET 代码以编程方式访问吊销端点。有关更多信息，请查看 [IdentityModel](#) 文档。

结束会话端点

结束会话端点可用于触发单点注销（请参阅 [规范](#)）。

要使用结束会话端点，客户端应用程序会将用户的浏览器重定向到结束会话 URL。用户在会话期间通过浏览器登录的所有应用程序都可以参与注销。

注解：结束会话端点的 URL 可通过发现文档 获得。

54.1 参数

id_token_hint

当用户被重定向到端点时，会提示他们是否真的要注销。客户端发送从身份验证收到的原始 *id_token* 可以绕过此提示。这作为名为 *id_token_hint* 的查询字符串参数传递。

post_logout_redirect_uri

如果传递了有效的 *id_token_hint*，则客户端还可以发送 *post_logout_redirect_uri* 参数。这可用于允许用户在注销后重定向回客户端。该值必须与客户端预配置的 *PostLogoutRedirectUris* 之一 (*client docs*) 匹配。

state

如果传递了有效的 *post_logout_redirect_uri*，则客户端还可以发送 *state* 参数。在用户重定向回客户端后，这将作为查询字符串参数返回给客户端。这通常由客户端用于跨重定向来回状态。

54.2 例子

```
GET /connect/endsession?id_token_
→hint=eyJhbGciOiJSUzI1NiIsImtpZCI6IjdlOGFkZmMzMjU1OTYyNzI0ZDY4NWZmYmIwOThjNDEyIiwidHlwIjoiSldUIIn0.
→eyJ0eXkiOiJ0e00TE3NjUzMjEsImV4cCI6MTQ5MTc2NTYyMSwiaXNzIjoiaHR0cDovL2xvY2FsaG9zdDo1MDAwIiwiaXVkiIjoian
→STzOWoeVYmtZdRAeRT95cMYEmClixWkmGwVH2Yyiks9BETotbSZiSfgE5kRh72kghN78N3-
→RgCTUmM2edB3bZx4H5ut3wWsBnZtQ2JLfTwJAjaLE9Ykt68ovNJySbm8hjZhHzPWKh55jzshivQvTX0GdtlbcDoEA1oNONxHkp
→rAALhFPkyKnVc-uB8IHtGNSyRWLFhwVqAdS3fRNO7iIs5hYRxeFSU7a5ZuUqZ6RRi-bcDhI-
→djKO5uAwihfbbpYcaY_TxXWoCmq8N8uAw9zqFsQUwcXymfOAi2UF3eFZt02hBu-shKA&post_logout_
→redirect_uri=http%3A%2F%2Flocalhost%3A7017%2Findex.html
```

注解：您可以使用 `IdentityModel` 客户端库以编程方式创建 `end_session` 请求.NET 代码。有关更多信息，请查看 `IdentityModel` 文档。

IdentityServer Options

- **IssuerUri** 设置将出现在发现文档中的颁发者名称和颁发的 JWT 令牌。建议不要设置此属性，它会根据客户端使用的主机名推断颁发者名称。
- **LowerCaseIssuerUri** 设置为 `false` 以保留 `IssuerUri` 的原始大小写。默认为 `true`。
- **AccessTokenJwtType** 指定用于访问令牌的 JWT 类型标头的值（默认为 `at+jwt`）。
- **EmitScopesAsSpaceDelimitedStringInJwt** 指定 JWT 中的作用域是作为数组还是字符串发出
- **EmitStaticAudienceClaim** 发出带有发行者/资源格式的 `aud` 声明。默认为 `false`。

55.1 Endpoints

允许启用/禁用单个端点，例如令牌，授权，用户信息等。

默认情况下，所有端点都已启用，但您可以通过禁用不需要的端点来锁定服务器。

- **EnableJwtRequestUri** 在授权端点上启用 JWT `request_uri` 处理。默认为 `false`。

55.2 Discovery

允许启用/禁用发现文档的各个部分，例如端点、范围、声明、授权类型等。

`CustomEntries` 字典允许向发现文档添加自定义元素。

55.3 Authentication

- **CookieAuthenticationScheme** 设置主机配置的用于交互用户的 cookie 认证方案。如果未设置，将从主机的默认身份验证方案中推断出该方案。当 `AddPolicyScheme` 在主机中用作默认方案时，通常使用此设置。
- **CookieLifetime** 身份验证 cookie 生存期（仅在使用 IdentityServer 提供的 cookie 处理程序时才有效）。
- **CookieSlidingExpiration** 指定 cookie 是否应该滑动（仅在使用 IdentityServer 提供的 cookie 处理程序时才有效）。
- **CookieSameSiteMode** 为内部 cookie 指定 SameSite 模式。
- **RequireAuthenticatedUserForSignOutMessage** 指示用户是否必须经过身份验证才能接受参数以结束会话端点。默认为 `false`。
- **CheckSessionCookieName** 用于检查会话端点的 cookie 的名称。
- **CheckSessionCookieDomain** 用于检查会话端点的 cookie 的域。
- **CheckSessionCookieSameSiteMode** 用于检查会话端点的 cookie 的 SameSite 模式。
- **RequireCspFrameSrcForSignout** 如果设置，将需要在结束会话回调端点上发出 frame-src CSP 标头，该端点将 `iframe` 呈现给客户端以进行前端通道注销通知。默认为 `true`。

55.4 Events

允许配置是否以及哪些事件应该提交到注册的事件接收器。有关事件的更多信息，请参见此处。

55.5 InputLengthRestrictions

允许对各种协议参数（如客户端 ID、范围、重定向 URI 等）设置长度限制。

55.6 UserInteraction

- **LoginUrl, LogoutUrl, ConsentUrl, ErrorUrl, DeviceVerificationUrl** 设置登录、注销、同意、错误和设备验证页面的 URL。
- **LoginReturnUrlParameter** 设置传递给登录页面的返回 URL 参数的名称。默认为 *returnUrl*。
- **LogoutIdParameter** 设置传递给注销页面的注销消息 id 参数的名称。默认为 *logoutId*。
- **ConsentReturnUrlParameter** 设置传递给同意页面的返回 URL 参数的名称。默认为 *returnUrl*。
- **ErrorIdParameter** 设置传递给错误页面的错误消息 id 参数的名称。默认为 *errorId*。
- **CustomRedirectReturnUrlParameter** 设置从授权端点传递给自定义重定向的返回 URL 参数的名称。默认为 *returnUrl*。
- **DeviceVerificationUserCodeParameter** 设置传递给设备验证页面的用户代码参数的名称。默认为 *userCode*。
- **CookieMessageThreshold** IdentityServer 和一些 UI 页面之间的某些交互需要 cookie 来传递状态和上下文（上面的任何页面都具有可配置的“message id”参数）。由于浏览器对 cookie 的数量和大小有限制，此设置用于防止创建过多的 cookie。该值设置将创建的任何类型的消息 cookie 的最大数量。一旦达到限制，最旧的消息 cookie 将被清除。这有效地指示了用户在使用 IdentityServer 时可以打开多少个选项卡。

55.7 Caching

只有在启动时在服务配置中启用了相应的缓存时，这些设置才适用。

- **ClientStoreExpiration** 从客户端存储加载的客户端配置的缓存持续时间。
- **ResourceStoreExpiration** 从资源存储加载的身份和 API 资源配置的缓存持续时间。

55.8 CORS

IdentityServer 为其某些端点支持 CORS。底层 CORS 实现由 ASP.NET Core 提供，因此它会自动注册到依赖注入系统中。

- **CorsPolicyName** 将评估 CORS 请求到 IdentityServer 的 CORS 策略的名称（默认为 *IdentityServer4*）。处理这个的策略提供者是根据在依赖注入系统中注册的 *ICorsPolicyService* 实现的。如果您希望自定义允许连接的 CORS 源集，则建议您提供 *ICorsPolicyService* 的自定义实现。
- **CorsPaths** IdentityServer 中支持 CORS 的端点。默认为发现、用户信息、令牌和撤销端点。
- **PreflightCacheDuration** *Nullable<TimeSpan>* 指示要在预检 *Access-Control-Max-Age* 响应标头中使用的值。默认为 *null* 表示没有在响应中设置缓存头。

55.9 CSP (内容安全政策)

IdentityServer 在适当的情况下为某些响应发出 CSP 标头。

- **Level** 要使用的 CSP 级别。默认情况下使用 CSP 级别 2，但如果必须支持旧浏览器，则将其更改为 `CspLevel.One` 以适应它们。
- **AddDeprecatedHeader** 指示是否还应发出旧的 `X-Content-Security-Policy` CSP 标头（除了基于标准的标头值）。默认为 `true`。

55.10 Device Flow

- **DefaultUserCodeType** 要使用的用户代码类型，除非在客户端级别设置。默认为 *Numeric*，一个 9 位代码。
- **Interval** 定义令牌端点上允许的最小轮询间隔。默认为 5。

55.11 Mutual TLS

- **Enabled** 指定是否应启用 MTLS 支持。默认为 `false`。
- **ClientCertificateAuthenticationScheme** 指定 X.509 客户端证书的身份验证处理程序的名称。默认为 `"Certificate"`。
- **DomainName** 指定用于运行 MTLS 端点的子域或完整域的名称（如果未设置，将使用基于路径的端点）。使用简单字符串（例如 `mtls`）设置子域，使用完整域名（例如 `identityserver-mtls.io`）设置完整域名。使用完整域名时，还需要将 `IssuerName` 设置为固定值。
- **AlwaysEmitConfirmationClaim** 指定如果存在客户端证书，是否为访问令牌发出 `cnf` 声明。通常，只有当客户端使用客户端证书进行身份验证时，才会发出 `cnf` 声明，将此设置为 `true`，无论身份验证方法如何，都将设置声明。（默认为 `false`）。

Identity Resource

This class models an identity resource.

Enabled Indicates if this resource is enabled and can be requested. Defaults to true.

Name The unique name of the identity resource. This is the value a client will use for the scope parameter in the authorize request.

DisplayName This value will be used e.g. on the consent screen.

Description This value will be used e.g. on the consent screen.

Required Specifies whether the user can de-select the scope on the consent screen (if the consent screen wants to implement such a feature). Defaults to false.

Emphasize Specifies whether the consent screen will emphasize this scope (if the consent screen wants to implement such a feature). Use this setting for sensitive or important scopes. Defaults to false.

ShowInDiscoveryDocument Specifies whether this scope is shown in the discovery document. Defaults to true.

UserClaims List of associated user claim types that should be included in the identity token.

This class models an OAuth scope.

Enabled Indicates if this resource is enabled and can be requested. Defaults to true.

Name The unique name of the API. This value is used for authentication with introspection and will be added to the audience of the outgoing access token.

DisplayName This value can be used e.g. on the consent screen.

Description This value can be used e.g. on the consent screen.

UserClaims List of associated user claim types that should be included in the access token.

57.1 Defining API scope in appsettings.json

The `AddInMemoryApiResource` extension method also supports adding clients from the ASP.NET Core configuration file:

```
"IdentityServer": {  
  "IssuerUri": "urn:sso.company.com",  
  "ApiScopes": [  
    {  
      "Name": "IdentityServerApi"  
    },  
    {
```

(下页继续)

(续上页)

```
        "Name": "resource1.scope1"
    },
    {
        "Name": "resource2.scope1"
    },
    {
        "Name": "scope3"
    },
    {
        "Name": "shared.scope"
    },
    {
        "Name": "transaction",
        "DisplayName": "Transaction",
        "Description": "A transaction"
    }
]
}
```

Then pass the configuration section to the `AddInMemoryApiScopes` method:

```
AddInMemoryApiScopes(configuration.GetSection("IdentityServer:ApiScopes"))
```

API Resource

This class models an API resource.

Enabled Indicates if this resource is enabled and can be requested. Defaults to true.

Name The unique name of the API. This value is used for authentication with introspection and will be added to the audience of the outgoing access token.

DisplayName This value can be used e.g. on the consent screen.

Description This value can be used e.g. on the consent screen.

ApiSecrets The API secret is used for the introspection endpoint. The API can authenticate with introspection using the API name and secret.

AllowedAccessTokenSigningAlgorithms List of allowed signing algorithms for access token. If empty, will use the server default signing algorithm.

UserClaims List of associated user claim types that should be included in the access token.

Scopes List of API scope names.

58.1 Defining API resources in appsettings.json

The `AddInMemoryApiResource` extensions method also supports adding API resources from the ASP.NET Core configuration file:

```
"IdentityServer": {
  "IssuerUri": "urn:sso.company.com",
  "ApiResources": [
    {
      "Name": "resource1",
      "DisplayName": "Resource #1",

      "Scopes": [
        "resource1.scope1",
        "shared.scope"
      ]
    },
    {
      "Name": "resource2",
      "DisplayName": "Resource #2",

      "UserClaims": [
        "name",
        "email"
      ],

      "Scopes": [
        "resource2.scope1",
        "shared.scope"
      ]
    }
  ]
}
```

Then pass the configuration section to the `AddInMemoryApiResource` method:

```
AddInMemoryApiResources(configuration.GetSection("IdentityServer:ApiResources"))
```

The `Client` class models an OpenID Connect or OAuth 2.0 client - e.g. a native application, a web application or a JS-based application.

59.1 Basics

Enabled Specifies if client is enabled. Defaults to `true`.

ClientId Unique ID of the client

ClientSecrets List of client secrets - credentials to access the token endpoint.

RequireClientSecret Specifies whether this client needs a secret to request tokens from the token endpoint (defaults to `true`)

RequireRequestObject Specifies whether this client needs to wrap the authorize request parameters in a JWT (defaults to `false`)

AllowedGrantTypes Specifies the grant types the client is allowed to use. Use the `GrantTypes` class for common combinations.

RequirePkce Specifies whether clients using an authorization code based grant type must send a proof key (defaults to `true`).

AllowPlainTextPkce Specifies whether clients using PKCE can use a plain text code challenge (not recommended - and default to `false`)

RedirectUris Specifies the allowed URIs to return tokens or authorization codes to

AllowedScopes By default a client has no access to any resources - specify the allowed resources by adding the corresponding scopes names

AllowOfflineAccess Specifies whether this client can request refresh tokens (be requesting the `offline_access` scope)

AllowAccessTokensViaBrowser Specifies whether this client is allowed to receive access tokens via the browser. This is useful to harden flows that allow multiple response types (e.g. by disallowing a hybrid flow client that is supposed to use `code id_token` to add the `token` response type and thus leaking the token to the browser.

Properties Dictionary to hold any custom client-specific values as needed.

59.2 Authentication/Logout

PostLogoutRedirectUri Specifies allowed URIs to redirect to after logout. See the [OIDC Connect Session Management spec](#) for more details.

FrontChannelLogoutUri Specifies logout URI at client for HTTP based front-channel logout. See the [OIDC Front-Channel spec](#) for more details.

FrontChannelLogoutSessionRequired Specifies if the user's session id should be sent to the FrontChannelLogoutUri. Defaults to true.

BackChannelLogoutUri Specifies logout URI at client for HTTP based back-channel logout. See the [OIDC Back-Channel spec](#) for more details.

BackChannelLogoutSessionRequired Specifies if the user's session id should be sent in the request to the BackChannelLogoutUri. Defaults to true.

EnableLocalLogin Specifies if this client can use local accounts, or external IdPs only. Defaults to *true*.

IdentityProviderRestrictions Specifies which external IdPs can be used with this client (if list is empty all IdPs are allowed). Defaults to empty.

UserSsoLifetime *added in 2.3* The maximum duration (in seconds) since the last time the user authenticated. Defaults to `null`. You can adjust the lifetime of a session token to control when and how often a user is required to reenter credentials instead of being silently authenticated, when using a web application.

59.3 Token

IdentityTokenLifetime Lifetime to identity token in seconds (defaults to 300 seconds / 5 minutes)

AllowedIdentityTokenSigningAlgorithms List of allowed signing algorithms for identity token. If empty, will use the server default signing algorithm.

AccessTokenLifetime Lifetime of access token in seconds (defaults to 3600 seconds / 1 hour)

AuthorizationCodeLifetime Lifetime of authorization code in seconds (defaults to 300 seconds / 5 minutes)

AbsoluteRefreshTokenLifetime Maximum lifetime of a refresh token in seconds. Defaults to 2592000 seconds / 30 days

SlidingRefreshTokenLifetime Sliding lifetime of a refresh token in seconds. Defaults to 1296000 seconds / 15 days

RefreshTokenUsage *ReUse* the refresh token handle will stay the same when refreshing tokens
OneTime the refresh token handle will be updated when refreshing tokens. This is the default.

RefreshTokenExpiration *Absolute* the refresh token will expire on a fixed point in time (specified by the *AbsoluteRefreshTokenLifetime*). This is the default.

Sliding when refreshing the token, the lifetime of the refresh token will be renewed (by the amount specified in *SlidingRefreshTokenLifetime*). The lifetime will not exceed *AbsoluteRefreshTokenLifetime*.

UpdateAccessTokenClaimsOnRefresh Gets or sets a value indicating whether the access token (and its claims) should be updated on a refresh token request.

AccessTokenType Specifies whether the access token is a reference token or a self contained JWT token (defaults to *Jwt*).

IncludeJwtId Specifies whether JWT access tokens should have an embedded unique ID (via the *jti* claim). Defaults to *true*.

AllowedCorsOrigins If specified, will be used by the default CORS policy service implementations (In-Memory and EF) to build a CORS policy for JavaScript clients.

Claims Allows settings claims for the client (will be included in the access token).

AlwaysSendClientClaims If set, the client claims will be sent for every flow. If not, only for client credentials flow (default is *false*)

AlwaysIncludeUserClaimsInIdToken When requesting both an id token and access token, should the user claims always be added to the id token instead of requiring the client to use the userinfo endpoint. Default is *false*.

ClientClaimsPrefix If set, the prefix client claim types will be prefixed with. Defaults to *client_*. The intent is to make sure they don't accidentally collide with user claims.

PairWiseSubjectSalt Salt value used in pair-wise subjectId generation for users of this client.

59.4 Consent Screen

RequireConsent Specifies whether a consent screen is required. Defaults to *false*.

AllowRememberConsent Specifies whether user can choose to store consent decisions. Defaults to *true*.

ConsentLifetime Lifetime of a user consent in seconds. Defaults to null (no expiration).

ClientName Client display name (used for logging and consent screen)

ClientUri URI to further information about client (used on consent screen)

LogoUri URI to client logo (used on consent screen)

59.5 Device flow

UserCodeType Specifies the type of user code to use for the client. Otherwise falls back to default.

DeviceCodeLifetime Lifetime to device code in seconds (defaults to 300 seconds / 5 minutes)

GrantValidationResult

The `GrantValidationResult` class models the outcome of grant validation for extensions grants and resource owner password grants.

The most common usage is to either new it up using an identity (success case):

```
context.Result = new GrantValidationResult(  
    subject: "818727",  
    authenticationMethod: "custom",  
    claims: optionalClaims);
```

...or using an error and description (failure case):

```
context.Result = new GrantValidationResult(  
    TokenRequestErrors.InvalidGrant,  
    "invalid custom credential");
```

In both case you can pass additional custom values that will be included in the token response.

在创建令牌或处理对 `userinfo` 或自省端点的请求时，`IdentityServer` 通常需要有关用户的身份信息。默认情况下，`IdentityServer` 只有身份验证 cookie 中的声明可用于此身份数据。

将用户所需的所有可能声明都放入 cookie 是不切实际的，因此 `IdentityServer` 定义了一个扩展点，允许根据用户需要动态加载声明。此扩展点是 `IProfileService`，开发人员通常会实现此接口以访问包含用户身份数据的自定义数据库或 API。

61.1 IProfileService APIs

GetProfileDataAsync 预期为用户加载声明的 API。它传递了一个 `ProfileDataRequestContext` 的实例。

IsActiveAsync 预期指示当前是否允许用户获取令牌的 API。它传递了一个 `IsActiveContext` 的实例。

61.2 ProfileDataRequestContext

为用户声明的请求建模，并且是返回这些声明的工具。它包含以下属性：

Subject 对用户建模的 `ClaimsPrincipal`。

Client 正在为其请求声明的客户端。

RequestedClaimTypes 正在请求的声明类型的集合。

Caller 请求声明的上下文的标识符（例如身份令牌、访问令牌或用户信息端点）。常量 `IdentityServerConstants.ProfileDataCallers` 包含不同的常量值。

IssuedClaims 将返回的 Claim 列表。这预计将由自定义 `IProfileService` 实现填充。

AddRequestedClaims `ProfileDataRequestContext` 上的扩展方法来填充 `IssuedClaims`，但首先根据 `RequestedClaimTypes` 过滤声明。

61.3 请求的范围和声明映射

客户端请求的范围控制用户声明在令牌中返回给客户端的内容。`GetProfileDataAsync` 方法负责根据 `ProfileDataRequestContext` 上的 `RequestedClaimTypes` 集合动态获取这些声明。

`RequestedClaimTypes` 集合是根据资源上定义的用户声明填充的，该用户声明对范围进行建模。如果请求身份令牌并且请求的范围是身份资源，则 `RequestedClaimTypes` 中的声明将根据 `IdentityResource` 中定义的用户声明类型填充。如果请求访问令牌并且请求的范围是 *API* 资源，则 `RequestedClaimTypes` 中的声明将根据 `ApiResource` 和（或）`Scope`。

61.4 IsActiveContext

对请求建模以确定当前是否允许用户获取令牌。它包含以下属性：

Subject 对用户建模的 `ClaimsPrincipal`。

Client 正在为其请求声明的 客户端。

Caller 请求声明的上下文的标识符（例如身份令牌、访问令牌或用户信息端点）。常量 `IdentityServerConstants.ProfileDataCallers` 包含不同的常量值。

IsActive 指示是否允许用户获取令牌的标志。这预计由自定义 `IProfileService` 实现分配。

IdentityServer Interaction Service

The `IIdentityServerInteractionService` interface is intended to provide services to be used by the user interface to communicate with IdentityServer, mainly pertaining to user interaction. It is available from the dependency injection system and would normally be injected as a constructor parameter into your MVC controllers for the user interface of IdentityServer.

62.1 IIdentityServerInteractionService APIs

GetAuthorizationContextAsync Returns the `AuthorizationRequest` based on the `returnUrl` passed to the login or consent pages.

IsValidReturnUrl Indicates if the `returnUrl` is a valid URL for redirect after login or consent.

GetErrorContextAsync Returns the `ErrorMessage` based on the `errorId` passed to the error page.

GetLogoutContextAsync Returns the `LogoutRequest` based on the `logoutId` passed to the logout page.

CreateLogoutContextAsync Used to create a `logoutId` if there is not one presently. This creates a cookie capturing all the current state needed for signout and the `logoutId` identifies that cookie. This is typically used when there is no current `logoutId` and the logout page must capture the current user's state needed for sign-out prior to redirecting to an external identity provider for signout. The newly created `logoutId` would need to be round-tripped to the external identity provider at signout time, and then used on the signout callback page in the same way it would be on the normal logout page.

GrantConsentAsync Accepts a `ConsentResponse` to inform IdentityServer of the user's consent to a particular `AuthorizationRequest`.

DenyAuthorizationAsync Accepts a `AuthorizationError` to inform IdentityServer of the error to return to the client for a particular `AuthorizationRequest`.

GetAllUserGrantsAsync Returns a collection of `Grant` for the user. These represent a user's consent or a client's access to a user's resource.

RevokeUserConsentAsync Revokes all of a user's consents and grants for a client.

RevokeTokensForCurrentSessionAsync Revokes all of a user's consents and grants for clients the user has signed into during their current session.

62.2 AuthorizationRequest

Client The client that initiated the request.

RedirectUri The URI to redirect the user to after successful authorization.

DisplayMode The display mode passed from the authorization request.

UiLocales The UI locales passed from the authorization request.

IdP The external identity provider requested. This is used to bypass home realm discovery (HRD). This is provided via the "idp:" prefix to the `acr_values` parameter on the authorize request.

Tenant The tenant requested. This is provided via the "tenant:" prefix to the `acr_values` parameter on the authorize request.

LoginHint The expected username the user will use to login. This is requested from the client via the `login_hint` parameter on the authorize request.

PromptMode The prompt mode requested from the authorization request.

AcrValues The acr values passed from the authorization request.

ValidatedResources The `ResourceValidationResult` which represents the validated resources from the authorization request.

Parameters The entire parameter collection passed to the authorization request.

RequestObjectValues The validated contents of the request object (if present).

62.3 ResourceValidationResult

Resources The resources of the result.

ParsedScopes The parsed scopes represented by the result.

RawScopeValues The original (raw) scope values represented by the validated result.

62.4 ErrorMessage

DisplayMode The display mode passed from the authorization request.

UiLocales The UI locales passed from the authorization request.

Error The error code.

RequestId The per-request identifier. This can be used to display to the end user and can be used in diagnostics.

62.5 LogoutRequest

ClientId The client identifier that initiated the request.

PostLogoutRedirectUri The URL to redirect the user to after they have logged out.

SessionId The user's current session id.

SignOutIFrameUrl The URL to render in an `<iframe>` on the logged out page to enable single sign-out.

Parameters The entire parameter collection passed to the end session endpoint.

ShowSignoutPrompt Indicates if the user should be prompted for signout based upon the parameters passed to the end session endpoint.

62.6 ConsentResponse

ScopesValuesConsented The collection of scopes the user consented to.

RememberConsent Flag indicating if the user's consent is to be persisted.

Description Optional description the user can set for the grant (e.g. the name of the device being used when consent is given). This can be presented back to the user from the *persisted grant service*.

Error Error, if any, for the consent response. This will be returned to the client in the authorization response.

ErrorDescription Error description. This will be returned to the client in the authorization response.

62.7 Grant

SubjectId The subject id that allowed the grant.

ClientId The client identifier for the grant.

Description The description the user assigned to the client or device being authorized.

Scopes The collection of scopes granted.

CreationTime The date and time when the grant was granted.

Expiration The date and time when the grant will expire.

Device Flow Interaction Service

The `IDeviceFlowInteractionService` interface is intended to provide services to be used by the user interface to communicate with `IdentityServer` during device flow authorization. It is available from the dependency injection system and would normally be injected as a constructor parameter into your MVC controllers for the user interface of `IdentityServer`.

63.1 `IDeviceFlowInteractionService` APIs

`GetAuthorizationContextAsync` Returns the `DeviceFlowAuthorizationRequest` based on the `userCode` passed to the login or consent pages.

`DeviceFlowInteractionResult` Completes device authorization for the given `userCode`.

63.2 `DeviceFlowAuthorizationRequest`

`ClientId` The client identifier that initiated the request.

`ScopesRequested` The scopes requested from the authorization request.

63.3 DeviceFlowInteractionResult

IsError Specifies if the authorization request errored.

ErrorDescription Error description upon failure.

Entity Framework Support

An EntityFramework-based implementation is provided for the configuration and operational data extensibility points in IdentityServer. The use of EntityFramework allows any EF-supported database to be used with this library.

The code for this library is located [here](#) (with the underlying storage code [here](#)) and the NuGet package is [here](#).

The features provided by this library are broken down into two main areas: configuration store and operational store support. These two different areas can be used independently or together, based upon the needs of the hosting application.

64.1 Configuration Store support for Clients, Resources, and CORS settings

If client, identity resource, API resource, or CORS data is desired to be loaded from a EF-supported database (rather than use in-memory configuration), then the configuration store can be used. This support provides implementations of the `IClientStore`, `IResourceStore`, and the `ICorsPolicyService` extensibility points. These implementations use a `DbContext`-derived class called `ConfigurationDbContext` to model the tables in the database.

To use the configuration store support, use the `AddConfigurationStore` extension method after the call to `AddIdentityServer`:

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    const string connectionString = @"Data Source=(LocalDb)\MSSQLLocalDB;
    ↪database=IdentityServer4.EntityFramework-2.0.0;trusted_connection=yes;";
```

(下页继续)

(续上页)

```
var migrationsAssembly = typeof(Startup).GetTypeInfo().Assembly.GetName().Name;

services.AddIdentityServer()
    // this adds the config data from DB (clients, resources, CORS)
    .AddConfigurationStore(options =>
    {
        options.ConfigureDbContext = builder =>
            builder.UseSqlServer(connectionString,
                sql => sql.MigrationsAssembly(migrationsAssembly));
    });
}
```

To configure the configuration store, use the `ConfigurationStoreOptions` `options` object passed to the configuration callback.

64.2 ConfigurationStoreOptions

This options class contains properties to control the configuration store and `ConfigurationDbContext`.

ConfigureDbContext Delegate of type `Action<DbContextOptionsBuilder>` used as a callback to configure the underlying `ConfigurationDbContext`. The delegate can configure the `ConfigurationDbContext` in the same way if EF were being used directly with `AddDbContext`, which allows any EF-supported database to be used.

DefaultSchema Allows setting the default database schema name for all the tables in the `ConfigurationDbContext`

```
options.DefaultSchema = "myConfigurationSchema";
```

If you need to change the schema for the Migration History Table, you can chain another action to the `UseSqlServer`:

```
options.ConfigureDbContext = b =>
    b.UseSqlServer(connectionString,
        sql => sql.MigrationsAssembly(migrationsAssembly).MigrationsHistoryTable(
            ↪ "MyConfigurationMigrationTable", "myConfigurationSchema"));
```

64.3 Operational Store support for persisted grants

If *persisted grants* are desired to be loaded from a EF-supported database (rather than the default in-memory database), then the operational store can be used. This support provides implementations of the `IPersistedGrantStore` extensibility point. The implementation uses a `DbContext`-derived class called `PersistedGrantDbContext` to model the table in the database.

To use the operational store support, use the `AddOperationalStore` extension method after the call to `AddIdentityServer`:

```
public IServiceCollection ConfigureServices(IServiceCollection services)
{
    const string connectionString = @"Data Source=(LocalDb)\MSSQLLocalDB;
    ↪database=IdentityServer4.EntityFramework-2.0.0;trusted_connection=yes;";
    var migrationsAssembly = typeof(Startup).GetTypeInfo().Assembly.GetName().Name;

    services.AddIdentityServer()
        // this adds the operational data from DB (codes, tokens, consents)
        .AddOperationalStore(options =>
        {
            options.ConfigureDbContext = builder =>
                builder.UseSqlServer(connectionString,
                    sql => sql.MigrationsAssembly(migrationsAssembly));

            // this enables automatic token cleanup. this is optional.
            options.EnableTokenCleanup = true;
            options.TokenCleanupInterval = 3600; // interval in seconds (default is
    ↪3600)
        });
}
```

To configure the operational store, use the `OperationalStoreOptions` options object passed to the configuration callback.

64.4 OperationalStoreOptions

This options class contains properties to control the operational store and `PersistedGrantDbContext`.

ConfigureDbContext Delegate of type `Action<DbContextOptionsBuilder>` used as a callback to configure the underlying `PersistedGrantDbContext`. The delegate can configure the `PersistedGrantDbContext` in the same way if EF were being used directly with `AddDbContext`, which allows any EF-supported database to be used.

DefaultSchema Allows setting the default database schema name for all the tables in the

PersistedGrantDbContext.

EnableTokenCleanup Indicates whether expired grants will be automatically cleaned up from the database. The default is `false`.

TokenCleanupInterval The token cleanup interval (in seconds). The default is 3600 (1 hour).

注解: The token cleanup feature does *not* remove persisted grants that are *consumed* (see *persisted grants*).

64.5 Database creation and schema changes across different versions of IdentityServer

It is very likely that across different versions of IdentityServer (and the EF support) that the database schema will change to accommodate new and changing features.

We do not provide any support for creating your database or migrating your data from one version to another. You are expected to manage the database creation, schema changes, and data migration in any way your organization sees fit.

Using EF migrations is one possible approach to this. If you do wish to use migrations, then see the *EF quickstart* for samples on how to get started, or consult the Microsoft [documentation on EF migrations](#).

We also publish [sample SQL scripts](#) for the current version of the database schema.

ASP.NET Identity Support

An ASP.NET Identity-based implementation is provided for managing the identity database for users of IdentityServer. This implementation implements the extensibility points in IdentityServer needed to load identity data for your users to emit claims into tokens.

The repo for this support is located [here](#) and the NuGet package is [here](#).

To use this library, configure ASP.NET Identity normally. Then use the `AddAspNetIdentity` extension method after the call to `AddIdentityServer`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddIdentityServer()
        .AddAspNetIdentity<ApplicationUser>();
}
```

`AddAspNetIdentity` requires as a generic parameter the class that models your user for ASP.NET Identity (and the same one passed to `AddIdentity` to configure ASP.NET Identity). This configures IdentityServer to use the ASP.NET Identity implementations of `IUserClaimsPrincipalFactory`, `IResourceOwnerPasswordValidator`, and `IProfileService`. It also configures some of ASP.NET Identity's options for use with IdentityServer (such as claim types to use and authentication cookie settings).

When using your own implementation of `IUserClaimsPrincipalFactory`, make sure that you register it before

calling the `IdentityServer.AddAspNetIdentity` extension method.

Here are some online, remote and classroom training options to learn more about ASP.NET Core identity & IdentityServer4.

66.1 Identity & Access Control for modern Applications (using ASP.NET Core 2 and IdentityServer4)

That's our own three day flagship course (including extensive hands-on labs) that we deliver as part of conferences, on-sites and remote.

The agenda and dates for public training can be found [here](#), [contact](#) us for private workshops.

66.2 PluralSight courses

There are some good courses on PluralSight around identity, ASP.NET Core and IdentityServer.

new

- [Securing Angular Apps with OpenID and OAuth2](#)
- [ASP.NET Core Identity Management Playbook](#)
- [Getting Started with ASP.NET Core and OAuth](#)
- [Securing ASP.NET Core with OAuth2 and OpenID Connect](#)

- [Understanding ASP.NET Core Security \(Centralized Authentication with a Token Service\)](#)

older

- [Introduction to OAuth2, OpenID Connect and JSON Web Tokens \(JWT\)](#)
- [Web API v2 Security](#)
- [Using OAuth to Secure Your ASP.NET API](#)
- [OAuth2 and OpenID Connect Strategies for Angular and ASP.NET](#)

67.1 Team posts

67.1.1 2020

- [Flexible Access Token Validation in ASP.NET Core](#)
- [Resource Access in IdentityServer4 v4 and going forward](#)
- [Automatic Token Management for ASP.NET Core and Worker Services 1.0](#)
- [Mutual TLS and Proof-of-Possession Tokens: Summary](#)
- [Mutual TLS and Proof-of-Possession Access Tokens –Part 1: Setup](#)
- [Hardening OpenID Connect/OAuth Authorize Requests \(and Responses\)](#)
- [Hardening Refresh Tokens](#)
- [OAuth 2.0: The long Road to Proof-of-Possession Access Tokens](#)
- [Outsourcing IdentityServer4 Token Signing to Azure Key Vault](#)
- [Using ECDSA in IdentityServer4](#)

67.1.2 2019

- Scope and claims design in IdentityServer
- Try Device Flow with IdentityServer4
- The State of the Implicit Flow in OAuth2
- An alternative way to secure SPAs (with ASP.NET Core, OpenID Connect, OAuth 2.0 and ProxyKit)
- Automatic OAuth 2.0 Token Management in ASP.NET Core
- Encrypting Identity Tokens in IdentityServer4

67.1.3 2018

- IdentityServer4 Update
- IdentityServer and Swagger
- Removing Shared Secrets for OAuth Client Authentication
- Creating Your Own IdentityServer4 Storage Library

67.1.4 2017

- Platforms where you can run IdentityServer4
- Optimizing Tokens for size
- Identity vs Permissions
- Bootstrapping OpenID Connect: Discovery
- Extending IdentityServer4 with WS-Federation Support
- Announcing IdentityServer4 RC1
- Getting Started with IdentityServer 4
- IdentityServer 4 SharePoint Integration using WS-Federation

67.2 Community posts

- Blazor WebAssembly authentication and authorization with IdentityServer4
- Additional API Endpoints to IdentityServer 4
- Securing Hangfire Dashboard using an OpenID Connect server (IdentityServer 4)
- OAuth 2.0 - OpenID Connect & IdentityServer

- [Running IdentityServer4 in a Docker Container](#)
- [Connecting Zendesk and IdentityServer 4 SAML 2.0 Identity Provider](#)
- [IdentityServer localization using ui_locales](#)
- [Self-issuing an IdentityServer4 token in an IdentityServer4 service](#)
- [IdentityServer4 on the ASP.NET Team Blog](#)
- [Angular2 OpenID Connect Implicit Flow with IdentityServer4](#)
- [Full Server Logout with IdentityServer4 and OpenID Connect Implicit Flow](#)
- [IdentityServer4, ASP.NET Identity, Web API and Angular in a single Project](#)
- [Secure your .NETCore web applications using IdentityServer 4](#)
- [ASP.NET Core IdentityServer4 Resource Owner Password Flow with custom UserRepository](#)
- [Secure ASP.NET Core MVC with Angular using IdentityServer4 OpenID Connect Hybrid Flow](#)
- [Adding an external Microsoft login to IdentityServer4](#)
- [Implementing Two-factor authentication with IdentityServer4 and Twilio](#)
- [Security Experiments with gRPC and ASP.NET Core 3.0](#)
- [ASP.NET Core OAuth Device Flow Client with IdentityServer4](#)
- [Securing a Vue.js app using OpenID Connect Code Flow with PKCE and IdentityServer4](#)
- [Using an OData Client with an ASP.NET Core API](#)
- [OpenID Connect back-channel logout using Azure Redis Cache and IdentityServer4](#)
- [Single Sign Out in IdentityServer4 with Back Channel Logout](#)

68.1 2020

- January [NDC London] –Implementing OpenID Connect and OAuth 2.0 –Tips from the Trenches
- January [NDC London] –OpenID Connect & OAuth 2.0 –Security Best Practices

68.2 2019

- October [TDC] –Securing Web Applications and APIs with ASP.NET Core 3.0
- January [NDC] –Securing Web Applications and APIs with ASP.NET Core 2.2 and 3.0
- January [NDC] –Building Clients for OpenID Connect/OAuth 2-based Systems

68.3 2018

- 26/09 [DevConf] –Authorization for modern Applications
- 17/01 [NDC London] –IdentityServer v2 on ASP.NET Core v2 - an Update
- 17/01 [NDC London] –Implementing authorization for web apps and APIs (aka PolicyServer announcement)
- 17/01 [DotNetRocks] –IdentityServer and PolicyServer on DotNetRocks

68.4 2017

- 14/09 [Microsoft Learning] –Introduction to IdentityServer for ASP.NET Core - Brock Allen
- 14/06 [NDC Oslo] –Implementing Authorization for Web Applications and APIs
- 22/02 [NDC Mini Copenhagen] –IdentityServer4: New & Improved for ASP.NET Core - Dominick Baier
- 02/02 [DotNetRocks] –IdentityServer4 on DotNetRocks
- 16/01 [NDC London] –IdentityServer4: New and Improved for ASP.NET Core
- 16/01 [NDC London] –Building JavaScript and mobile/native Clients for Token-based Architectures

68.5 2016

- The history of .NET identity and IdentityServer Channel9 interview
- Authentication & secure API access for native & mobile Applications - Dominick Baier
- ASP.NET Identity 3 - Brock Allen
- Introduction to IdentityServer3 - Brock Allen

68.6 2015

- Securing Web APIs –Patterns & Anti-Patterns - Dominick Baier
- Authentication and authorization in modern JavaScript web applications –how hard can it be? - Brock Allen

68.7 2014

- Unifying Authentication & Delegated API Access for Mobile, Web and the Desktop with OpenID Connect and OAuth 2 - Dominick Baier