

WEBSERVICE SECURITY

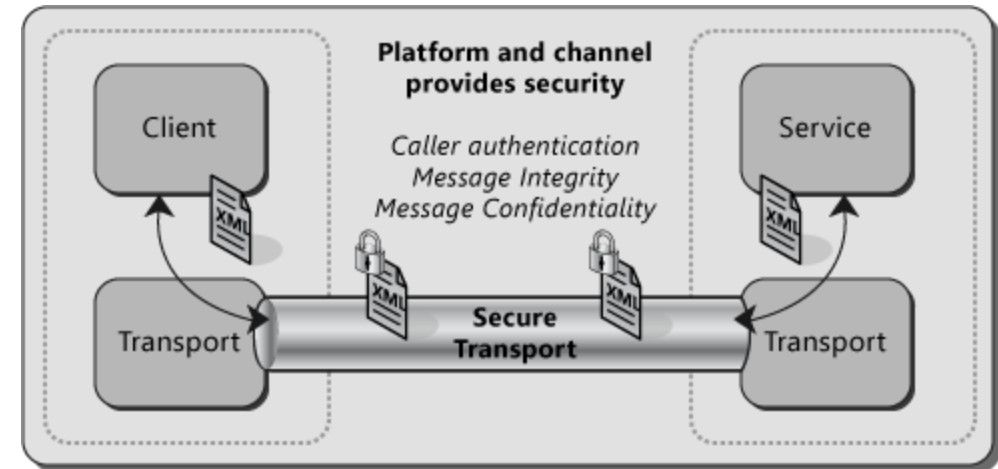
GITHUB



- Sourcen mit Beispielen zum Skript finden sie unter [florianwachs/AspNetWebServicesCourse \(github.com\)](https://github.com/florianwachs/AspNetWebServicesCourse)

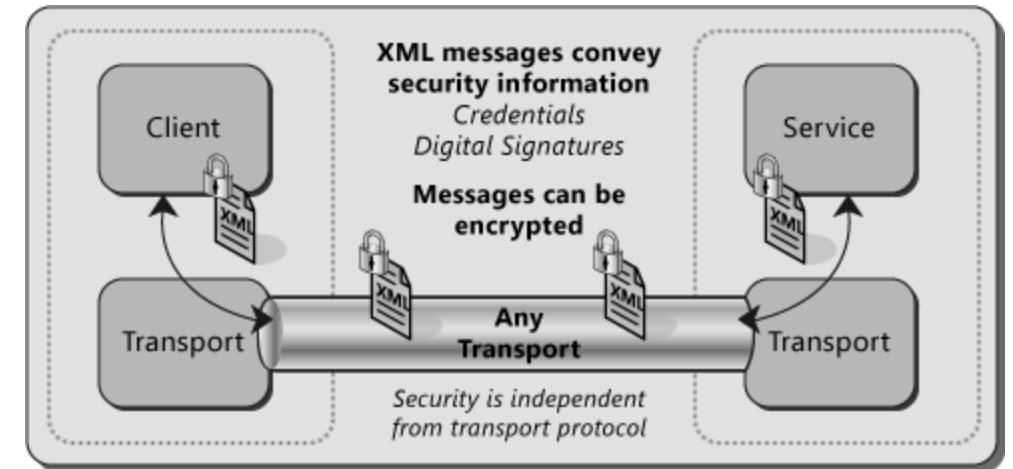
Transport Security

- Sicherheit zwischen zwei Endpunkten
- Bei mehreren „Zwischenstellen“ müssen sichere Verbindungen zwischen diesen aufgebaut werden und die Nachrichten weitergeleitet werden
- **Hop-to-Hop Security**
- Kommunikationspartner müssen nicht WS-Security verstehen
- Unterstützt weniger Authentifizierungsmethoden als Message Security
- Möglichkeit der Hardwarebeschleunigung
- Problem: Endet meist an der Firewall

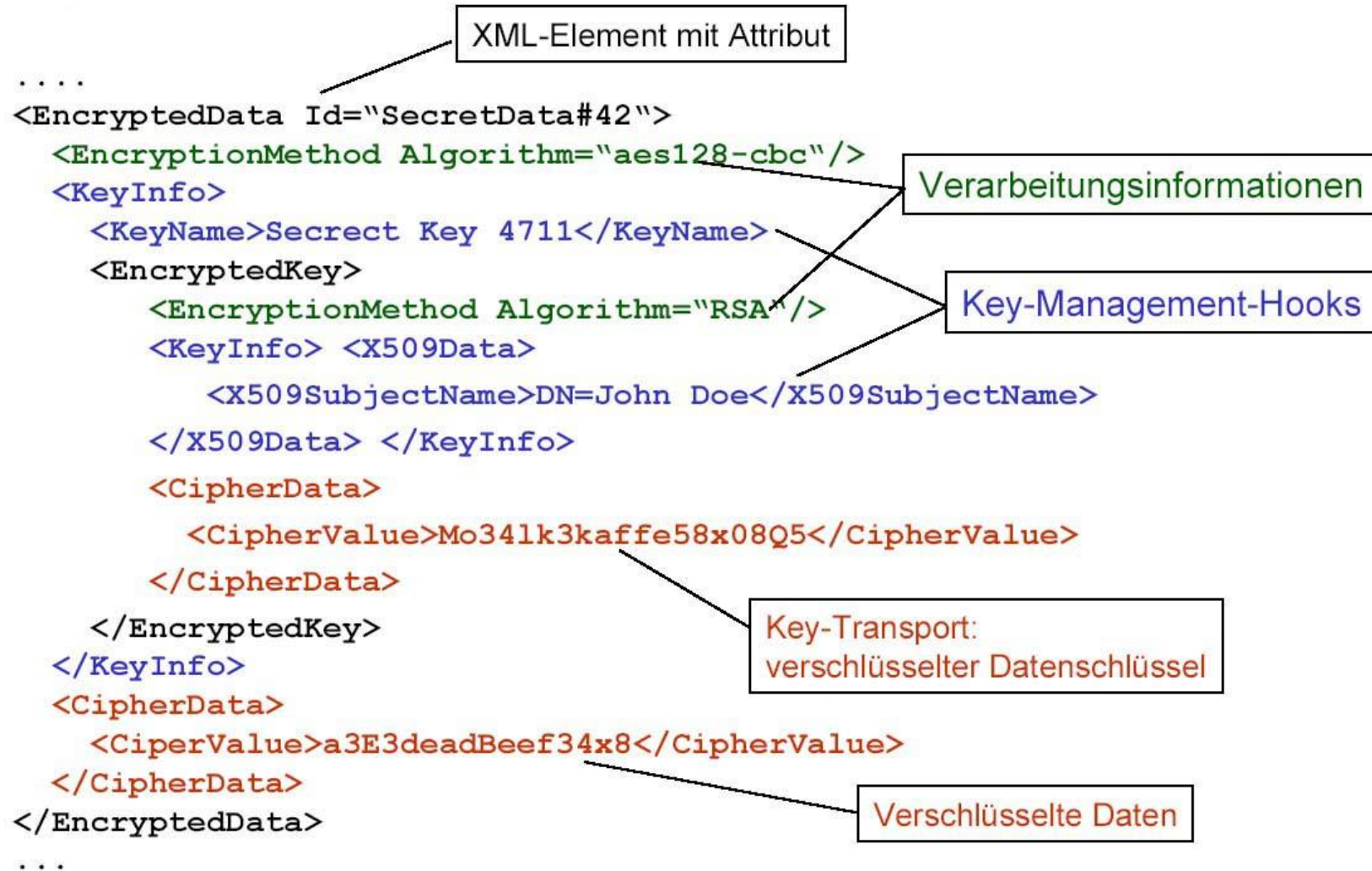


Message Security

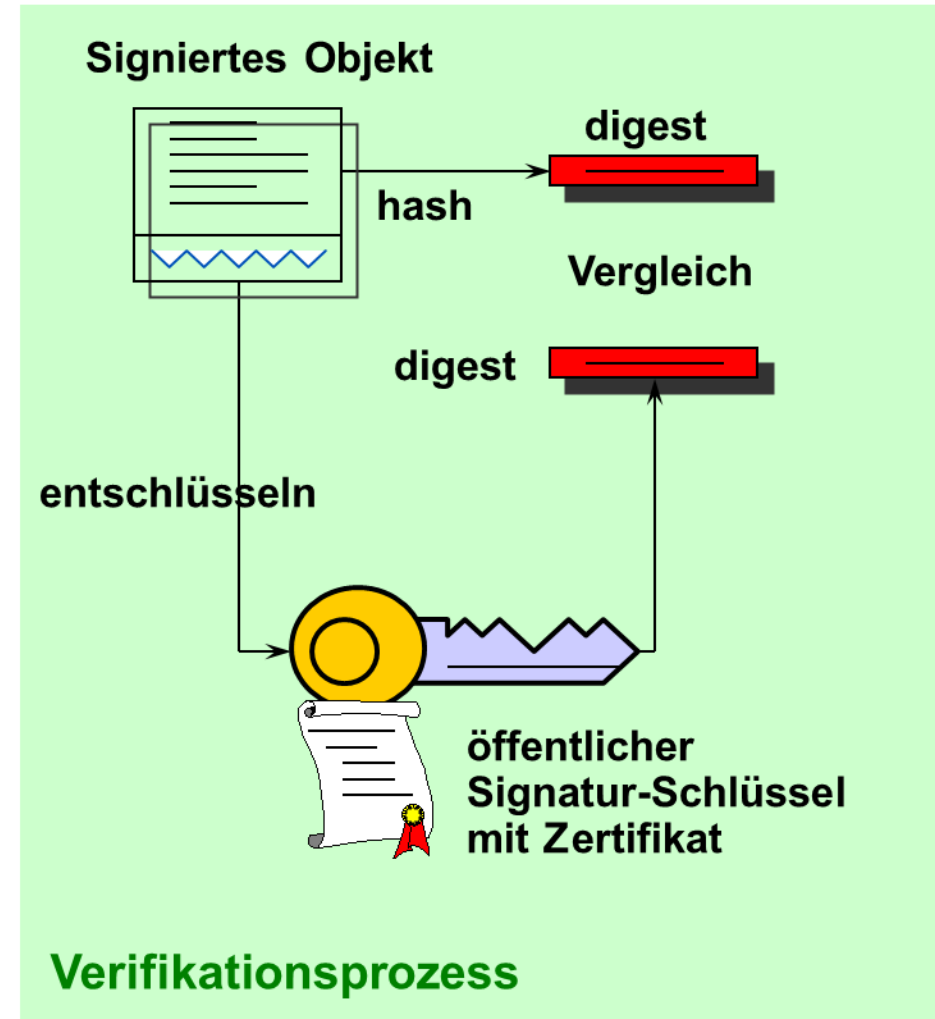
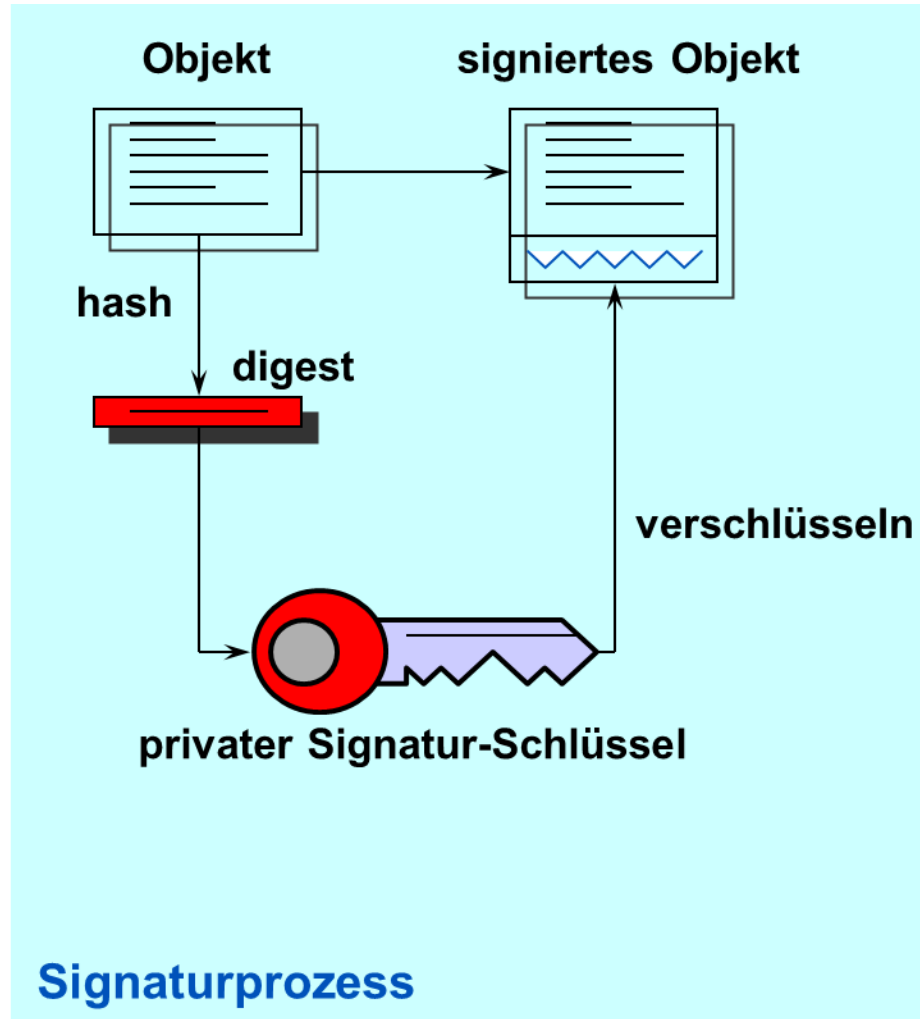
- Sicherstellung Integrität und Vertraulichkeit der Nachricht
- Durch Verschlüsselung und Signierung wird die Nachricht auch beim Transport über unsichere Verbindungen geschützt
- **End-to-End Security**
- Teilweise oder vollständige Verschlüsselung und / oder Signierung von Nachrichten
- Selektive Verschlüsselung einzelner Nachrichten
- Nutzt den WS-Security Standard
- Problem: Bei vielen Nachrichten langsam



Message Security



Message Security| Digitale Signatur



Message Security| Digitale Signatur

```
...  
<Signature>  
  <SignedInfo>  
    <CanonicalizationMethod Algorithm="c14n"/>  
    <SignatureMethod Algorithm="rsa-sha1"/>  
    <Reference URI="http://foo.org/picture.jpg.zip">  
      <Transforms>  
        <Transform Algorithm="UnZip">  
      </Transforms>  
      <DigestMethod Algorithm="sha1"/>  
      <DigestValue>345x3mUrks563X</DigestValue>  
    </Reference>  
  </SignedInfo>  
  <SignatureValue>MC0affe34lkV</SignatureValue>  
  <KeyInfo>  
    <X509Data>  
      <X509SubjectName>DN=John Doe</X509SubjectName>  
    </X509Data>  
  </KeyInfo>  
</Signature>  
...
```

Verarbeitungsinformationen:
verwendete Algorithmen
Kanonisierung

Verarbeitungsinformationen:
Datenquellen für Signatur
Transformationen

Signaturwert

Key-Management:
PKI-Anbindung

.Net Identity

- Eine Menge von Eigenschaften, die einen Benutzer des Systems beschreibt
- Liefert Auskunft darüber,
 - ob ein User authentifiziert ist
 - mit welcher Authentifizierungsmethode sich der User angemeldet hat
 - welchen Namen der User hat
- Je nach Authentifizierungsmethode gibt es unterschiedliche Implementierungen die weitergehende Informationen liefern
 - GenericIdentity
 - WindowsIdentity
 - ClaimsIdentity

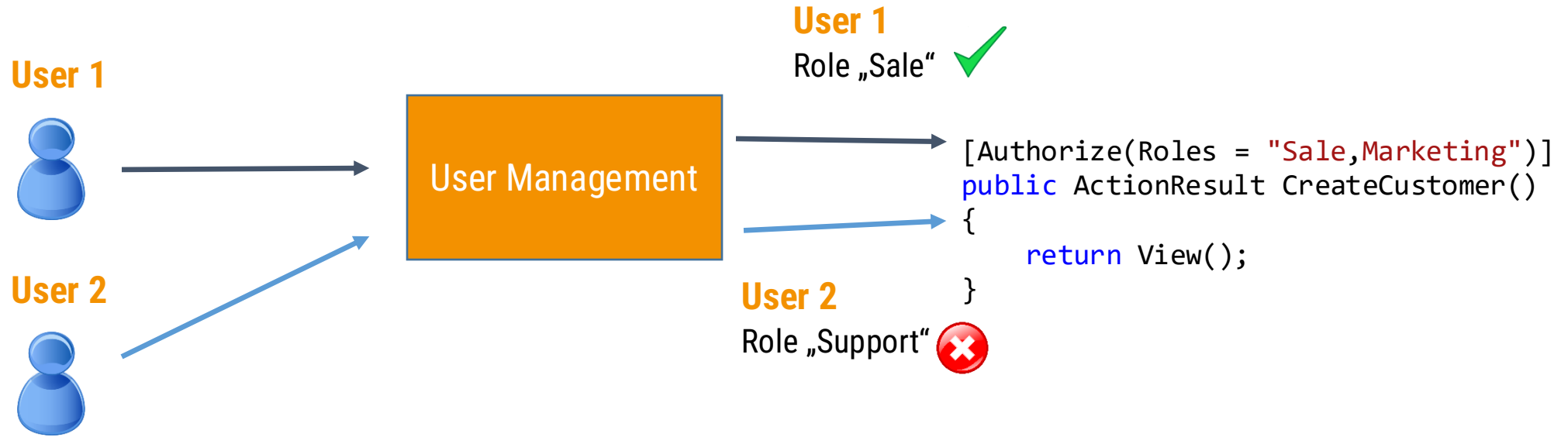
.Net Principal

- Security-Context auf den aus dem Code zugegriffen werden kann
- Liefert Auskunft darüber,
 - In welchen Rollen der Benutzer Mitglied ist
 - Liefert die Identity des Users
- Je nach Authentifizierungsmethode gibt es unterschiedliche Implementierungen
 - GenericPrincipal
 - WindowsPrincipal
 - ClaimsPrincipal

Roles

- Rollen werden Usern in der Regel über ein User Management System zugewiesen
- Z.B: ADMIN, SALESDIRECTOR, SUPPORT
- Über ein Principal kann getestet werden, ob eine Identity (User) teil einer bestimmten Rolle ist
- Ein User kann Teil von mehreren Rollen sein
- Roles werden zunehmend durch Claims ersetzt
- Nachteile
 - „Alles oder nichts“-Ansatz, ein User ist Teil einer Rolle oder nicht
 - Für viele Anwendungsfälle nicht feingranular genug

Roles



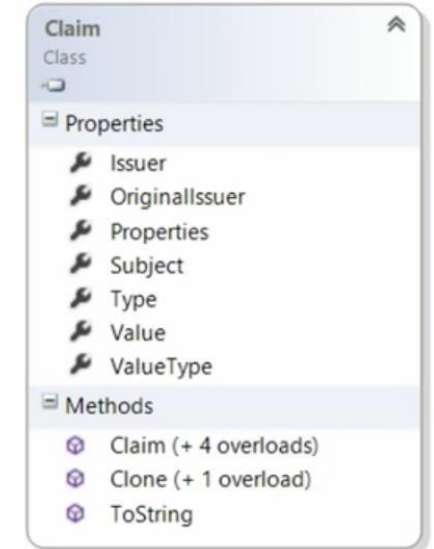
Erweiterung

Die Berechtigung einer neuen Rolle hat häufig Code-Änderungen zur Folge. Alternativ könnte man eine „CreateCustomer“-Rolle erfinden, was aber selten vorkommt, da hier eine große Menge an Rollen im System definiert und verwaltet werden muss

```
[Authorize(Roles = "Sale, Marketing, Support")]  
public ActionResult CreateCustomer()  
{  
    return View();  
}
```

Claims

- Standard für alle neuen Microsoft Autorisierungssysteme
- Ein Claim ist ein Informationsfragment, welches mit andern Claims zusammen ein Gesamtbild der Identität eines Benutzers formt
- Ein Claim wird von einem **Issuer** ausgestellt. Einem Claim kann man nur soweit trauen, wie man dem Austeller des Claims traut. In einem Windows-Netzwerk können Claims vom Active Directory Domänencontroller ausgestellt werden
- Im folgenden typische Claims
 - Name
 - Email
 - Age
 - **Roles**
- Claims treten meist als Einheit mit Tokens auf
- Ein Token ist eine Menge von Claims, die von dem ausstellenden Issuer signiert und kodiert werden
- Claims sind Key / Value Pairs



Claims

Token

Claims:
Department: Customer Management
Email: user1@company.com
AccessLevel: E2

User 1



Credentials

User 2



Credentials

Authorization
Server

Token

Claims:
Department: Support
Email: user2@company.com
AccessLevel: A1

Authorization Server

Der User sendet seine Credentials (Username / Password, Certificates, ...) an eine Autorisierungskomponente. Diese kann, muss aber kein eigenständiger Server sein. Diese Komponente liefert dem Aufrufer einen signierten Token zurück. Dieser enthält alle Claims in kodierter Form. Meist haben diese Token auch noch ein Ablaufdatum. In der Regel speichert der Client (Webseite, Single Page Applikation, WPF-Anwendung) über den sich der User anmeldet diesen Token

Claims

Token

Claims:
Department: Customer Management
Email: user1@company.com
AccessLevel: E2

User 1



User 2



Token

Claims:
Department: Support
Email: user2@company.com
AccessLevel: A1



Resource Server

Der Resource Server repräsentiert unseren Webservice. Dieser erhält im Request einen Token vom Client. Die Signatur des Token wird geprüft, um sicherzustellen dass die enthaltenen Claims von einer vertrauenswürdigen Stelle ausgestellt wurden

Authorize

Zusätzlich zu Roles unterstützt das Authorize-Attribute Policies. Diese können beim Start der Applikation konfiguriert werden und sich aus Claims zusammensetzen

```
[Authorize(Policy = "CanCreateCustomer")]  
[HttpPost]  
public IActionResult CreateCustomer(Customer p)  
{  
}
```

```
services.AddAuthorization(options =>  
{  
    options.AddPolicy("CanCreateCustomer", policy =>  
        policy.RequireClaim("Department", "Customer Management"));  
});
```

Policies

Beschreiben wir in ASP.NET Identity

Token

- Typische tokenbasierte Systeme
 - Azure AD
 - OpenID Connect / OAuth 2.0
 - Identity Server
- Informationen über Token (JWT, JSON Web Token)
 - <https://www.iana.org/assignments/jwt/jwt.xhtml>
 - <https://tools.ietf.org/html/rfc7519>
 - Base64-kodierter String mit Signierung und Verschlüsselung der Informationen über eine Identität in Form von „Claims“ enthält
 - Es gibt ein Set von typischen Claims für Name, Geburtsdatum, Email
 - Es können beliebige eigene Claims definiert werden

Arten von Token| Identity Token

- Kann man sich als Personalausweis vorstellen
- Enthält Eigenschaften der Identität (Claims)
- Kann bei Bedarf verschlüsselt werden
- Enthält Informationen über die ausstellende Stelle (iss)
- Kann für „Sessions“ genutzt werden

sub	Subject (Identität des Users)
iss	Issuer (Aussteller des Token)
iat	Ausgestellt am
exp	Verfällt am
jti	JWT-Token Id
nbf	„Not Before Date“

```
{  
  "sub": "471587e9-a5e4-49d8-866e-5144565da3d6",  
  "name": "chuck",  
  "token_usage": "id_token",  
  "jti": "fcae25d9-1c04-4c1e-ae7d-4fac6bed8652",  
  "at_hash": "XhprWY5f_qBC50vi68WbcQ",  
  "nbf": 1496310113,  
  "exp": 1496311313,  
  "iat": 1496310113,  
  "iss": "http://localhost:28476/"  
}
```


Arten von Token| Access Token

- Enthält alle Identitätsinformationen und Claims
- Haben meist eine sehr kurze (Minuten bis Tage) Gültigkeit

Scope

Scopes definieren, welche Art und Menge von Claims man vom Identity-System erwartet, im Access Token wird nochmal zurückgeliefert welche Scopes Verwendung fanden

```
{
  "sub": "471587e9-a5e4-49d8-866e-5144565da3d6",
  "name": "chuck",
  "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/postalcode": "12345",
  "http://schemas.fhrwebservices.com/ws/2017/01/identity/claims/department":
"Customer Management",
  "CustomClaimType": "CustomClaimValue",
  "token_usage": "access_token",
  "jti": "3b0fed6-87c2-4d1a-846b-134057bdb369",
  "scope": [
    "openid",
    "email",
    "profile",
    "offline_access",
    "roles"
  ],
  "aud": "resource_server",
  "nbf": 1496310112,
  "exp": 1496313712,
  "iat": 1496310112,
  "iss": "http://localhost:28476/"
}
```

Arten von Token| Refresh Token

- Mit einem Refresh-Token kann ein neuer Access-Token generiert werden ohne nochmal die Credentials des Benutzers abfragen zu müssen
 - Sehr hilfreich bei Mobile Apps
- Haben meist eine lange Gültigkeit (Wochen, Monate)
- Häufig werden die Refresh-Tokens serverseitig gespeichert, so dass ein Benutzer diese für ungültig erklären kann. Danach ist keine neue Ausstellung von Access-Token möglich

Vorteile von Token-basierten Verfahren

- Erlauben Single-Sign-On
- Mit Refresh-Tokens „sicherer“, da nur einmalig die Credentials zum Auth-Server übertragen werden müssen
- Authentifizierung mit Access-Tokens ist stateless und self-contained
 - Alles Informationen sind bereits in den Token kodiert
 - Der Server muss nicht erst Informationen aus einer Datenbank oder einem anderen System laden
- Können über URL, Header oder Body übertragen werden das sehr kompakt

Wie sollten Token persistiert / vorgehalten werden?

- Access-Token / Refresh-Token sollten wie Kreditkartendaten behandelt werden
- Browser sind absolut vertrauensunwürdig, somit ist die Speicherung in Local Storage, Index-DB nicht empfehlenswert. 3rd-Party-Skripte können leicht alle Daten des Local Storage abgreifen. Was also tun für Browser?

Wie sollten Token persistiert / vorgehalten werden?

- Für **Nicht-Single-Page** Applikationen sollten die Token serverseitig verwahrt werden.
- Für **Single-Page** Applikationen wird empfohlen die Token nur im temporären Speicher zu halten und bei jeder Anmeldung neu abzufragen. Oder man verwendet den Authorization-Token-Flow. Besser ist ein **BFF** (Backend For Frontend mit **HTTPS-only Cookies**)
- **Native Apps** können meist die Sicherheitsfunktionen des Betriebssystems verwenden. Für die meisten OAuth-Provider gibt es entsprechende SDKs für Android, Linux, Windows, MacOS

Erweiterte Lösungen für SPAs

- Den User einer SPA-Applikation ohne Server-Backend jedes Mal zum Login zu zwingen ist meist ein Hemmnis für Benutzer
- Das Speichern von Tokens im Browser Storage ist aber unsicher
- Mögliche Lösungen
 - Web Authentication API
 - Silent Renew mit Implicit oder Authentication Code Grant Flow

Web Authentication API

- https://developer.mozilla.org/en-US/docs/Web/API/Web_Authentication_API
- Ist ein Web Standard zur Authentifizierung ohne Passwort über moderne Sicherheitstechnologien wie Biometrie (Windows Hello, Apple ID) und SmartCards / Token
- Ist nicht als Ersatz für OpenIdConnect gedacht, sondern arbeitet damit zusammen

Silent Renew

- Beim **Silent Renew** wird die Tatsache ausgenutzt dass der User noch eine gültige Session (meist über Cookies) mit dem Authentication-Provider (Identity Server, Google, ...) hat.
- Bemerkt der Client das der Auth-Token ungültig geworden ist, rendert die SPA ein unsichtbares **IFrame** auf der Seite mit einem speziellen Aufruf an den OIDC-Provider (**prompt=none**).
- Das IFrame wird mit speziellen Headern vor Cookie-Diebstahl geschützt
 - **X-Frame-Options: same-origin**

Silent Renew

- Ist die Session noch gültig wird eine **Login-Response** zurückgegeben als hätte sich der User gerade neu angemeldet
- Ist die Session nicht mehr gültig wird ein **Error-Code** nach dem OIDC-Standard zurückgegeben
 - **login_required**: User nicht eingeloggt
 - **consent_required**: User eingeloggt, muss aber noch zustimmen
 - **interaction_required**: Der User muss erst noch einem Redirect auf eine andere Seite folgen

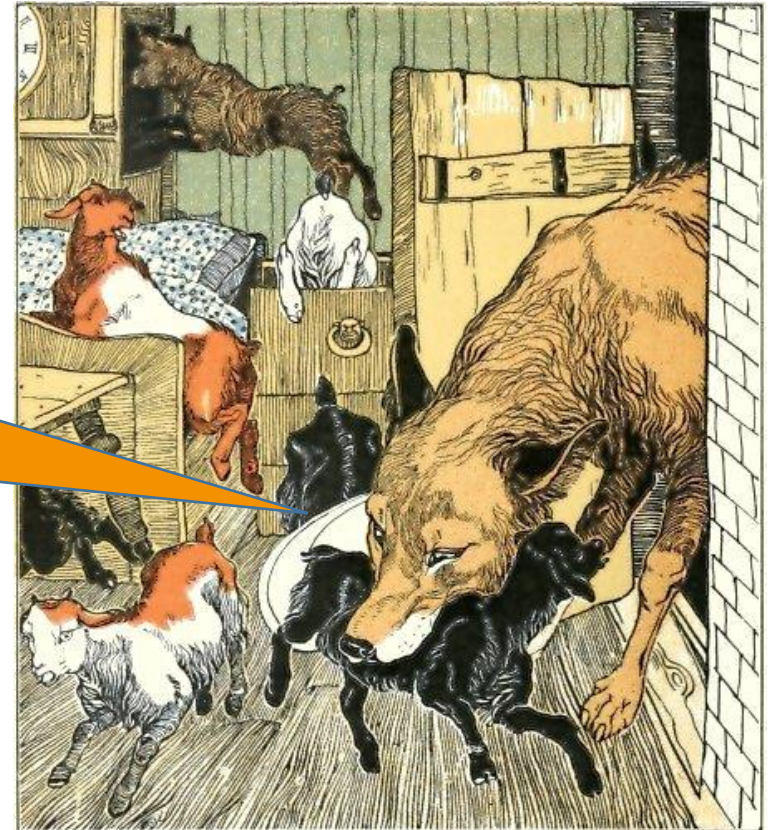
Silent Renew in SPAs

- Library to the rescue!
- Von den Machern von Identity Server gibt es eine JavaScript Library oidc-client.js. Diese implementiert den OpenIDConnect Standard. Mehr dazu in der Übung / Live Coding
- <https://github.com/IdentityModel/oidc-client-js>
- `npm install oidc-client --save`

Authentication

- Sicherstellung das der User wirklich der ist, für den er sich ausgibt (Authentifizierung)
- Verschiedene Möglichkeiten:
 - Username und Passwort
 - Client Zertifikate
 - Token

Mami ist da....



Authentifizierungsoptionen

- **None**
 - Anonymer Zugriff
- **Basic**
 - http Standard
 - Username und Passwort werden Base64-encoded übertragen (unbedingt Transport Security aktivieren)
- **Digest**
 - Im Prinzip wie Basic, nur das Username und Passwort als Hash übermittelt werden
- **Windows**
 - Kerberos für Domänen
 - NTLM für Arbeitsgruppen
 - Arbeitet mit Active Directory zusammen
- **Zertifikate**
 - X.509 Zertifikate von einer vertrauenswürdigen Zertifizierungsstelle
 - Häufig in B2B-Szenarien
- **OAuth 2.0 / OpenID Connect**
 - Token-basierter Ansatz

Authorization

- Definiert Regeln die festlegen, welche Aktivitäten ein User im System durchführen darf (Autorisierung)
- Setzt die erfolgreiche Authentifizierung voraus
- Typische Implementierungen
 - Role-based
 - Claims-based

ASP.NET Identity

Identity als Middleware

- Basisimplementierung basierend auf Entity Framework Core, kann aber beliebig ersetzt werden (z.B. Mongo-DB)
 - Vorgänger waren an ein relationales Datenbankmodell gebunden
- Bietet Funktionalität für Login und Membership (Benutzer) Management
- Bietet Erweiterungspunkte für 3rd-Party Identitätsprovider wie Google, Twitter, Microsoft usw.
- Hohe Erweiterbarkeit durch modulare Strukturen
- Vereinheitlichung von vormals unterschiedliche Konzepte (simple membership, universal providers und membership)

ASP.NET Identity

- Kümmert sich um **User**, **Authentication** und **Authorization**
- Erweiterbarkeit war bei den Vorgängern nur begrenzt gegeben
- ASP.NET Identity ermöglicht User-Management und führt die Authentifikation und Autorisation durch
- **Claims-based**
- Unterstützt Two-Factor-Authentifizierung
 - Captcha
 - Email mit Code
 - SMS mit Code
 - Custom (z.B. Hardware Token Generator)

ASP.NET Identity

- OAuth / Open ID
- Organizational mit Single Sign On Support
 - Active Directory
 - Azure Active Directory
 - Office 365
- Custom
 - Database backed
 - Custom data stores
- Roles
- Claims
- Nicht für Windows Authentifizierung geeignet

ASP.NET Identity

- Ideal kombinierbar mit Identity Server
- ASP.NET Identity kümmert sich um die Benutzerverwaltung
- Identity Server kümmert sich um die Absicherung der Ressourcen und das Ausstellen von Tokens
- Die Verwendung in ASP.NET Core bleibt dadurch transparent

ASP.NET Identity| Bestandteile

ApplicationUser

Leitet von IdentityUser ab. Kann zusätzliche Properties enthalten die per Migration auf die DB angewendet werden müssen

IdentityDbContext

Ableitung von IdentityDbContext<ApplicationUser>. Andere Manager-Klassen des Identity Systems können damit direkt arbeiten

UserManager<ApplicationUser>

Speichert User, Claims, Roles und bietet ein API diese zu verwalten

AuthorizeAttribute

Ermöglicht es Controller und Actions durch Autorisierung und Authentifizierung zu schützen.

IAuthorizationService

API-Implementierung um direkt auf Policies zugreifen und diese z.B. innerhalb einer Controller-Action überprüfen zu können

SignInManager<ApplicationUser>

API um zu prüfen ob sich ein User am System anmelden darf, ihn ein- und auszuloggen. Er erzeugt außerdem das UserPrincipal aus einem ApplicationUser

ASP.NET Identity| Getting Started

Aktuelle Authentication Provider

Identity Server ist eine sehr mächtige und ausgereifte Lösung für das Identity-Management. Einfacher aufzusetzen ist aber OpenIddict.

Name

[AspNet.Security.OpenIdConnect.Server \(ASOS\)](#)

[IdentityServer4](#)

[OpenIddict](#)

Description

Low-level/protocol-first OpenID Connect server framework for ASP.NET Core and OWIN/Katana

OpenID Connect and OAuth 2.0 framework for ASP.NET Core - officially certified by the OpenID Foundation and under governance of the .NET Foundation

Easy-to-use OpenID Connect server for ASP.NET Core

<https://docs.microsoft.com/en-us/aspnet/core/security/authentication/community>

Policies

Policies

In ASP.NET Core wurde das Identity Model grundlegend überarbeitet. Policies erlauben eine deutlich granularere Autorisierung. Policies können sich aus Claims, Roles oder CustomRequirements zusammensetzen

Beispiel-Claim

Department: Customer
Management
Email: user1@company.com
AccessLevel: E2
Age: 34
ManagerId: 123

AddAuthorization

In der Startup.cs Configure-Methode können die Policies dynamisch konfiguriert werden. Sie könnten z.B. aus einer Konfigurationsdatei oder Datenbank kommen

```
services.AddAuthorization(options =>
{
    options.AddPolicy("CanAccessSaleHistory", policy => policy.RequireClaim("ManagerId"));

    options.AddPolicy("CanReadCustomerAge", policy => policy.RequireClaim("Department", "Customer Management"));

    options.AddPolicy("CanCreateCustomer", policy => policy.RequireClaim("Department", "Customer Management", "Customer Support"));
});
```

AddPolicy

Mittels AddPolicy kann eine neue Policy-Definition erstellt werden. Es kann gefordert werden das ein oder mehrere Claims vorhanden sind. Zusätzlich kann noch vorausgesetzt werden, dass die Claims bestimmte Werte beinhalten

Policies

Komplexe Policies

Policies können aus mehreren Anforderungen zusammengesetzt werden. Auch Rollen und Add-Hoc Regeln (RequireAssertion) werden unterstützt. Für komplexere Anforderungen sollte aber eine Klasse welche das Interface `IAuthorizationRequirement` implementiert genutzt werden.

Beispiel-Claim

Department: Customer

Management

Email: user1@company.com

AccessLevel: E2

Age: 34

ManagerId: 123

```
options.AddPolicy("TopSecret", policy =>
{
    policy.RequireClaim("AccessLevel", "A1")
        .RequireRole("Executive Manager")
        .RequireAssertion(authContext =>
            authContext.User.HasClaim(claim =>
                claim.Type == "Age" && int.TryParse(claim.Value, out int age) && age > 18));
});
```

Policies

Komplexe Policies

Requirements erlauben es, besonders komplexe Policies abzubilden. Pro Requirement kann es ein oder mehrere Handler geben. Falls einer der Handler "Succeed" aufruft, ist das Requirement damit erfüllt.

```
options.AddPolicy(AppPolicies.CanReadStudentsEnrolledInCourse, policy =>
    {
        policy.AddRequirements(new ProfessorOrPrincipalRequirement());
    });
```

Policies

Komplexe Policies

Requirements erlauben es, besonders komplexe Policies abzubilden. Pro Requirement kann es ein oder mehrere Handler geben. Falls einer der Handler "Succeed" aufruft, ist das Requirement damit erfüllt.

```
public class ProfessorOrPrincipalRequirement : IAuthorizationRequirement
{
}
```


Policies

Komplexe Policies

Requirements erlauben es, besonders komplexe Policies abzubilden. Pro Requirement kann es ein oder mehrere Handler geben. Falls einer der Handler "Succeed" aufruft, ist das Requirement damit erfüllt.

```
public class IsProfessorHandler : AuthorizationHandler<ProfessorOrPrincipalRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
ProfessorOrPrincipalRequirement requirement)
    {
        if (context.User.IsProfessor())
        {
            // Wenn ein Authorization Handler pro Requirement Succeeded, ist das Requirement erfüllt.
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

Policies

Komplexe Policies

Requirements erlauben es, besonders komplexe Policies abzubilden. Pro Requirement kann es ein oder mehrere Handler geben. Falls einer der Handler "Succeed" aufruft, ist das Requirement damit erfüllt.

```
public class IsPrincipalHandler : AuthorizationHandler<ProfessorOrPrincipalRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
        ProfessorOrPrincipalRequirement requirement)
    {
        if (context.User.IsPrincipal())
        {
            // Wenn ein Authorization Handler pro Requirement Succeeded, ist das Requirement erfüllt.
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

Policies

Komplexe Policies

Requirements erlauben es, besonders komplexe Policies abzubilden. Pro Requirement kann es ein oder mehrere Handler geben. Falls einer der Handler "Succeed" aufruft, ist das Requirement damit erfüllt.

```
public class UniversityMemberRequirement : IAuthorizationRequirement
{
}

public class IsUniversityMemberHandler : AuthorizationHandler<UniversityMemberRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
        UniversityMemberRequirement requirement)
    {
        var user = context.User;
        if (user.IsPrincipal() || user.IsProfessor() || user.IsStudent())
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

Policies

Resource Policies

Policies können auch auf Ressourcen angewendet werden

```
public class CanReadCourseGradesRequirement : IAuthorizationRequirement
{
}

public class CanReadCourseGradesHandler : AuthorizationHandler<CanReadCourseGradesRequirement,
Course>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
CanReadCourseGradesRequirement requirement, Course resource)
    {
        if (context.User.IsPrincipal())
        {
            context.Succeed(requirement);
        }

        var userid = context.User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
        if (!string.IsNullOrEmpty(userid) && userid == resource.ProfessorId)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

Anwendung in Minimal API

Authorize

RequireAuthorization

Durch die Fluent-API können ein oder mehrere Policies angegeben werden.

```
group.MapGet("/chuck/books", () =>
{
    return Results.Ok(new Book[] { new Book() { Title = "Chucks Wisdom" } });
})
.Produces<List<Book>>(StatusCodes.Status200OK)
.Produces(StatusCodes.Status401Unauthorized)
.Produces(StatusCodes.Status403Forbidden)
.WithOpenApi()
.RequireAuthorization(AuthConstants.Policies.AllowedToReadChuckNorrisBooks);
```

Anwendung im API-Controller

Authorize

```
[Route("api/[controller]")]
[Authorize]
public class CustomersController : Controller
{

    [HttpGet, AllowAnonymous]
    public Task<IEnumerable<Customer>> Get()
    {
        return GetSecuredData();
    }

    [HttpPost, Authorize(Policy = AppPolicies.CanCreateCustomer)]
    public IActionResult InsertCustomer(Customer p)
    {
    }

    [HttpPut("{id}"), Authorize(Policy = AppPolicies.CanUpdateCustomer)]
    public IActionResult UpdateCustomer(string id, Customer p)
    {
    }
}
```

Authorize

Durch das Authorize-Attribut wird festgelegt, dass alle Controller-Actions des Controllers einen authentifizierten Benutzer voraussetzen. Es kann auf eine Controller-Action, einen Controller oder global für alle Controller angewendet werden. Das Attribut bietet auch die Möglichkeit Rollen und Policies zu definieren die erfüllt sein müssen um den API-Endpoint aufrufen zu dürfen

AllowAnonymous

Mit diesem Attribut kann die Wirkung des Authorize-Attributs wieder aufgehoben werden

Authorize

```
public class CustomersController : Controller
{
    [HttpGet, AllowAnonymous]
    public Task<IEnumerable<Customer>> Get() => GetSecuredData();

    private async Task<IEnumerable<Customer>> GetSecuredData()
    {
        var includeAge = await FullfilesPolicy(AppPolicies.CanReadCustomerAge);
        return people.Values
            .Select(p =>
                new Customer
                {
                    Name = p.Name,
                    Id = p.Id,
                    Age = includeAge ? p.Age : "TOP SECRET"
                }).ToList();
    }

    private Task<bool> FulfillsPolicy(string policy)
    {
        return _authorizationService.AuthorizeAsync(User, policy);
    }
}
```

AuthorizationService

Mit dem AuthorizationService kann im Controller geprüft werden, ob der aktuelle User eine Policy erfüllt oder nicht. Er erlaubt außerdem die Prüfung gegen Implementierungen von `IAuthorizationRequirement`-Implementierungen. Dadurch kann bis auf Ressourcenebene hinab eine Autorisierung durchgesetzt werden. Z.B. das Dokumente nur vom Ersteller gelöscht werden dürfen

Policies

```
[Authorize(Policy = AppPolicies.CanReadCourses)]
[Route("api/[controller]")]
[ApiController]
public class CoursesController : ControllerBase
{
    private readonly UniversityDbContext _dbContext;
    private readonly IAuthorizationService _authorizationService;

    public CoursesController(UniversityDbContext dbContext, IAuthorizationService authorizationService)
    {
        _dbContext = dbContext;
        _authorizationService = authorizationService;
    }

    [HttpGet("{courseId}/grades")]
    public async Task<ActionResult<IEnumerable<CourseGrade>>> GetCourseGrades(string courseId)
    {
        var course = await _dbContext.Courses
            .Include(c => c.Grades)
            .ThenInclude(sc => sc.Student)
            .Where(c => c.Id == courseId).FirstOrDefaultAsync();

        if (course == null)
        {
            return NotFound();
        }

        var authorizationResult = await _authorizationService.AuthorizeAsync(User, course, new CanReadCourseGradesRequirement());
        if (!authorizationResult.Succeeded)
        {
            return Forbid();
        }

        return Ok(course.Grades);
    }
}
```

Resource Policies

Policies können auch auf Ressourcen angewendet werden