1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

L'importanza della sicurezza nelle API: quali sono le procedure consigliate e come implementarle in ambiente .NET

DOCENTE
Prof. Ing. Antonio Della Selva

CORSISTA
Tommaso Petrelli

Corso di Reti di Calcolatori
Anno Accademico 2022-2023

Indice

Introduzione

Oggi, le API Web ricoprono vasti ruoli negli ambiti dell'informazione e della comunicazione, infatti, si considerano ormai come elementi indispensabili per lo sviluppo di sistemi Web, Cloud o IoT. Le API, incaricate della trasmissione dei dati, si pongono dunque come servizi tra questi sistemi.

È logico domandarsi che cosa succederebbe se un'API venisse danneggiata o esposta a potenziali attacchi informatici. Costruire API vulnerabili significa mettere a rischio enormi quantità di dati privati e sensibili.

Lo scopo del presente elaborato sarà quello di spiegare che cosa si intende per sicurezza delle API e quali sono i migliori approcci da utilizzare la loro protezione. Dopodiché si procederà nell'implementazione di un sistema basato su un'API REST. L'obbiettivo è quello di rendere sicure le comunicazioni che coinvolgono tale sistema e definire delle politiche di accesso alle risorse capaci di garantire il giusto grado di protezione. Le componenti che si considerano essenziali per costruire un sistema sicuro sono:

- API REST: servizio che offre le risorse. Implementa i meccanismi di autenticazione e validazione sui token JWT in ingresso forniti dai Client, così come le politiche di autorizzazione.
- Client: rappresenta un'applicazione che desidera accedere alle risorse dell'API. Questa utilizza il protocollo OpenID Connect (OIDC) per autenticare l'utente al sistema ed ottenere l'Access Token da un Idenity Server.
- Gateway API: servizio che si pone come interfaccia tra API privata e Client pubblici. Qui vengono definite le politiche di *throttling*, le *quota*, i vincoli per garantire la *Quality of Service* (QoS), e servizi di autenticazione per validare gli Access Token JWT in ingresso.
- Identity Server: è il servizio che si occupa di riconoscere i Client e gli utenti finali, e registrare le risorse che l'API mette a disposizione. A questo livello vengono implementati i protocolli OAuth 2.0 e OIDC per fornire il servizio di generazione dei token JWT. Affinché l'Identity Server possa generare dei token affidabili sarà

dotato di un certificato. Utilizzerà una chiave privata per firmarli, mentre la chiave pubblica potrà essere distribuita alle altre componenti perché esse possano verificare la firma del token ricevuto.

L'implementazione dell'intero sistema si basa sul framework Microsoft .NET Core 6. Il codice sorgente dell'intero progetto, può essere trovato alla repository GitHub [S1].

1 | Sicurezza delle API

Il primo capitolo è dedicato alla presentazione dei concetti essenziali che riguardano il mondo delle API e della loro sicurezza. L'obbiettivo non sarà quello di dare una rappresentazione esaustiva, bensì quello di fissare i concetti che verranno utilizzati e implementati nel capitolo successivo.

1.1 Le API: cosa sono e a cosa servono

Un'API (Application Programming Interface) definisce quali sono le regole da seguire per far comunicare sistemi software potenzialmente diversi tra loro. Il termine API si associa spesso a librerie software incluse in altre applicazioni; in realtà, è più corretto utilizzarlo facendo riferimento a una collezione di definizioni e protocolli che vengono poi usati dalle librerie. Le API possono essere considerate come un contratto tra un fornitore di informazioni e l'utente destinatario di tali dati fungendo quindi da elemento di intermediazione tra gli utenti o i clienti e le risorse o servizi che intendono ottenere. Le API si utilizzano maggiormente con lo scopo di integrarle all'interno di sistemi software per condividere risorse e informazioni assicurando al contempo sicurezza, controllo e autenticazione. L'utilizzo dell'API inoltre non impone all'utente di conoscere le specifiche con cui le risorse vengono recuperate o la loro provenienza [S2].

Le API dedicate alla comunicazione tra web server e web browser vengono dette API Web. Quest'ultime basano i meccanismi di comunicazione sul protocollo HTTP ed utilizzano principalmente formati di rappresentazione dati come JSON e XML.

1.1.1 Lo stile architetturale REST

Un'API REST (o RESTful) è un'API conforme ai principi di progettazione REST (*REpresentational State Transfer*). A differenza di SOAP [S3], REST è un insieme di vincoli architetturali, non un protocollo o uno standard. In questo modo è capace di fornire a chi sviluppa API un livello di flessibilità e di libertà relativamente elevato. Il fatto che REST sia un insieme di linee guida applicabili quando necessario, rende le API RESTful

più rapide, leggere, scalabili e flessibili, che di conseguenza risultano essere il metodo più diffuso per la connessione di componenti e applicazioni in un'architettura web [S2][S4].

Affinché un'API Web sia considerata RESTful deve rispettare i seguenti principi dello stile architetturale REST [S2][S4][S5]:

- **Interfaccia uniforme.** Le informazioni che vengono trasferite tra Client e Server sono caratterizzate da un formato di rappresentazione standard. Il formato JSON (Javascript Object Notaion) è quello attualmente più diffuso poiché, oltre ad essere indipendente dai linguaggi di programmazione, è anche facilmente leggibile.
- **Stateless.** Con questo termine ci si riferisce ad un metodo di comunicazione in cui il Server completa ogni richiesta del Client indipendentemente da tutte le richieste precedenti. Ciò significa che ogni richiesta deve includere tutte le informazioni necessarie per essere elaborata.
- **Sistema a livelli.** Spesso, le applicazioni Client e Server non si connettono direttamente tra loro poiché vengono aggiunti degli strati software e hardware intermedi. Lo stile architetturale REST prevede che né il Client né il Server dovrebbero essere capaci di capire se stanno comunicando con l'applicazione finale o con un intermediario.
- **Cacheability.** Quando possibile, le risorse devono essere archiviate nella cache dei Client o degli intermediari. L'obiettivo è quello di migliorare le prestazioni sui lati Client e Server, aumentandone al tempo stesso la scalabilità.

1.2 La sicurezza delle API Web REST

La sicurezza è un fattore che si deve sempre tenere in considerazione, ma diventa un punto cruciale quando si sviluppa un'API pubblica. Senza garantire pratiche di sicurezza adeguate, utenti malevoli potrebbero ottenere l'accesso a informazioni alle quali non dovrebbero accedere o addirittura ai privilegi per modificarle [S6]. Ad oggi, sono state consolidate delle *best practice* per rendere più sicura un'API REST, come ad esempio l'utilizzo di algoritmi di hashing per la protezione delle password e del protocollo HTTPS utilizzato per rendere la trasmissione dei dati sicura [S4]. Inoltre, è strettamente consigliabile implementare meccanismi di autenticazione e autorizzazione sfruttando tecnologie come chiavi o token di accesso.

Le vulnerabilità più comuni che uno sviluppatore dovrebbe tenere a mente durante l'implementazione di una API Web sono le seguenti:

- **Cross-site scripting (XSS).** Vulnerabilità che consente a chi lancia l'attacco di iniettare pezzi di codice dannosi in pagine web e app [S7].
- **Distributed Denial of Service (DDoS).** Gli attacchi DDoS cercano di disattivare singoli siti web o intere reti sommergendoli di traffico causato da migliaia di computer infettati, noti collettivamente come *botnet* [S8]. Non sempre le richieste vengono gestite nel modo più efficiente dai servizi web, esponendo tali servizi a questa tipologia di attacchi.
- **SQL Injection.** Si sfruttano vulnerabilità software per iniettare codice SQL con l'obbiettivo di ottenere accesso ad informazioni sensibili e creare, leggere, alterare o eliminare i dati memorizzati in un database [S9].
- **Spoofing.** È uno degli attacchi più diffusi di tipo Man-in-the-Middle (MITM) e consiste nell'impersonificazione da parte di un hacker di un altro dispositivo o di un altro utente su una rete, al fine di impadronirsi di dati, diffondere malware o superare dei controlli di accesso [S10].

Fortunatamente esistono delle contromisure adottabili da un team di sviluppo di servizi web. Di seguito se ne vogliono elencare i più comuni ed efficaci:

- **Crittografare i dati con TLS.** Ogni API Web dovrebbe utilizzare connessioni HTTPS (HTTP Secure) così da garantire l'utilizzo di certificati TLS/SSL. Lo standard TLS, infatti, consente di mantenere privata e crittografata una connessione tra due sistemi [S2]. I certificati TLS/SSL vengono rilasciati e firmati dalle Certification Authority (CA), che garantiscono la legittimità del dispositivo o dell'utente che l'ha richiesto.
- **Tecniche di controllo d'accesso.** Nelle API Web RESTful, l'autenticazione viene tipicamente gestita dagli *endpoint* che questa mette a disposizione. I metodi più semplici per implementare i meccanismi di autenticazione sono quelli basati su username e password, anche se, in realtà, sono quelli meno sicuri. Si consiglia di adottare generalmente meccanismi più avanzati basati sugli standard OAuth 2.0 e OpenID Connect assieme a strumenti come i JSON Web Token e chiavi API [S11].
- **Processi basati su *throttling* e *quota*.** È importante definire delle regole di *throttling* per proteggere le API da errori di programmazione, picchi di richieste o attacchi di tipo DDoS. Vengono definite delle *quota*, ossia il numero massimo di chiamate che possono essere effettuate verso un *endpoint* entro un determinato intervallo di tempo [S2].

1.2.1 I Gateway API

Un Gateway API è uno strumento di gestione delle API che si situa tra un Client e una raccolta di servizi lato Server e rappresenta il principale punto di coordinamento del traffico API comportandosi come un proxy inverso che accetta le chiamate API, aggrega i servizi richiesti per gestirle e restituisce i risultati appropriati [S12].

Il ruolo dei Gateway API è quello di intercettare tutte le richieste in entrata ed inviarle attraverso il sistema di gestione delle API, dove avrà luogo l'elaborazione di quelle funzioni ritenute necessarie. In base al tipo di implementazione, il Gateway API eseguirà svariate funzioni distinte, tra le quali: autenticazione, routing, *throttling*, fatturazione, analisi del traffico, policy, avvisi e sicurezza [S12].

1.2.2 JSON Web Token

Il JSON Web Token (JWT) è un mezzo compatto e sicuro per rappresentare i Claim da trasferire tra due parti. I Claim in un JWT sono codificati come elementi JSON inseriti nel *payload* di una struttura JSON Web Signature (JWS) o come testo in chiaro di una struttura JSON Web Encryption (JWE), che consentano di firmare digitalmente i Claim o di proteggere la loro integrità con un Message Authentication Code (MAC) e/o crittografandoli [S13].

I JWT vengono largamente utilizzati all'interno dei meccanismi di autorizzazione con schema di autenticazione Bearer e per lo scambio di informazioni in modo sicuro. Un JWT si compone di tre parti separate da un punto (.), ovvero:

- **Header.** È tipicamente composto da due parti: il tipo di token, cioè JWT, e il tipo di algoritmo utilizzato per la firma, come HMAC SHA256 o RSA [S14]. Ad esempio

```
{
  "alg": "HMAC",
  "typ": "JWT"
}
```

- **Payload.** Questa sezione contiene i Claim, ovvero, dichiarazioni su un'entità (in genere, l'utente) oppure informazioni generiche. Esistono tre tipi di Claim:
  - **Registered Claim.** Si tratta di un insieme di Claim predefiniti, non obbligatori ma raccomandati, per fornire un insieme di Claim utili e interoperabili. Alcuni di essi sono: iss (issuer), exp (expiration time), sub (subject) ed aud (audience).

  - **Public Claim.** Questi possono essere definiti a piacere da chi utilizza i JWT. Tuttavia, per evitare collisioni, dovrebbero essere definiti nel registro IANA dei JSON Web Token [S15] o essere definiti come URI che contengono uno spazio dei nomi resistente alle collisioni.
  - **Private Claim.** Si tratta di Claim personalizzati creati per condividere informazioni tra le parti che concordano sul loro utilizzo.

  Un esempio di *payload* potrebbe essere il seguente:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

  Si fa presente che le informazioni contenute nei JWT, sebbene protette da manomissioni, sono leggibili da chiunque. È consigliabile non inserire informazioni segrete tra i Claim, a meno che non siano crittografate.

- **Signature.** Per creare la parte della firma è necessario prendere l'*header* ed il *payload* codificati, un *secret*, l'algoritmo del tipo specificato nell'*header* e firmare il tutto. Ad esempio, se si vuole utilizzare l'algoritmo HMAC SHA256, la parte di *signature* verrà creata nel modo seguente:

```
HMACSHA256(
    base64UrlEncode(header) + "." +
    base64UrlEncode(payload),
    secret
)
```

  La *signature* viene utilizzata per verificare che il JWT non sia stato modificato durante il percorso e, nel caso di token firmati con una chiave privata, può anche verificare che il mittente del JWT sia chi dice di essere.

Ognuna di queste parti viene codificata in Base64Url e aggiunta al JWT, costituito quindi da tre stringhe Base64Url separate da punti.

Di seguito viene mostrato un JWT con i precedenti *header* (in rosso) e *payload* (in blu) codificati, e firmato con un *secret* (in verde):

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOi
IxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5M
DIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

1.2.3 OAuth 2.0

Oauth 2.0 è un framework di autorizzazione che promuove la concessione di un accesso limitato all'utente sul proprio account tramite un servizio HTTP. Questo protocollo definisce quattro componenti che agiranno assieme per generare un Access Token [S16]:

- **Resource Owner.** È un'entità in grado di concedere l'accesso a una risorsa protetta.
- **Resource Server.** È il Server che ospita le risorse protette, capace di evadere le richieste di accesso a tali risorse utilizzando gli Access Token.
- **Client.** È un'applicazione che, in possesso di autorizzazione, richiede le risorse protette per conto del proprietario della risorsa.
- **Authorization Server.** È il Server che rilascia gli Access Token al Client dopo aver autenticato il proprietario della risorsa e ottenuto l'autorizzazione (può coincidere con il Resource Server).

Quando un Client necessita di ottenere l'accesso a delle risorse, OAuth definisce il seguente flusso di chiamate [S16]:

A. il Client effettua una richiesta di autorizzazione al Resource Owner;

B. il Resource Owner accetta fornendo un Authorization Grant (sono delle credenziali che rappresentano l'autorizzazione del Resource Owner);

C. il Client richiede un Access Token, autenticandosi attraverso l'Authorization Server e presentando l'Authorization Grant;

D. l'Authorization Server autentica il Client e convalida l'Authorization Grant. Se quest'ultima dovesse rivelarsi valida, l'Authorization Server potrà rilasciare un Access Token;

E. il Client richiede la risorsa protetta al Resource Server e si autentica presentando l'Access Token;

F. il Resource Server convalida l'Access Token e, se valido, serve la richiesta.

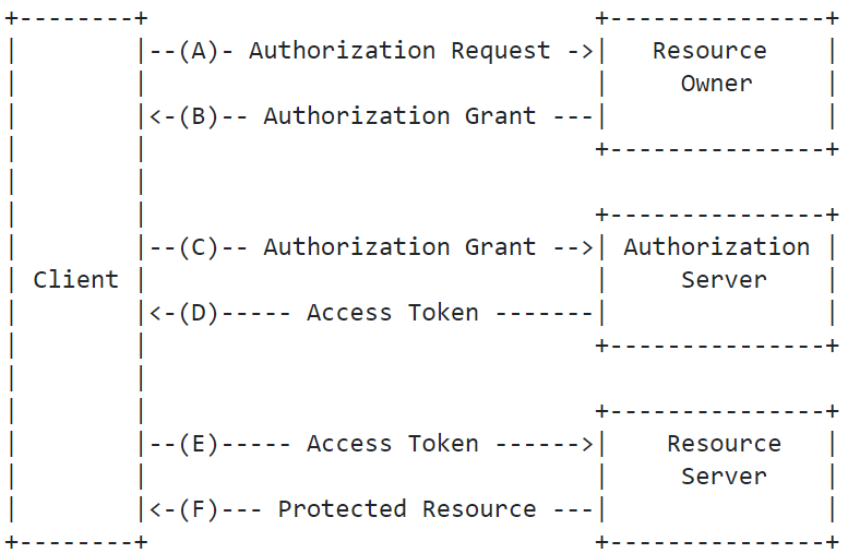

Figura 1.1: Flusso delle richieste per ottenere l'accesso alle risorse [S16].

OpenID Connect Un accorgimento che coinvolge il framework OAuth per aiutare a gestire l'autenticazione del Resource Owner è il livello di identità chiamato OpenID Connect [S17] che consente ai Client di verificare l'identità dell'utente finale in base all'autenticazione eseguita da un Authentication Server, nonché di ottenere informazioni di base sul profilo dell'utente finale in modo interoperabile e simile a REST.

1.2.4 MFA: autenticazione a più fattori

L'autenticazione a più fattori (MFA, o Multi-Factor Authentication) è un metodo di verifica dell'identità che richiede agli utenti di fornire almeno un fattore di autenticazione in aggiunta alla password oppure almeno due in sostituzione della stessa, per ottenere l'accesso a un sito web, un'applicazione o una rete [S18].

La solidità di uno schema MFA dipende dai tipi di fattori di autenticazione che vengono richiesti agli utenti:

- fattori di conoscenza: qualcosa di cui l'utente è a conoscenza;
- fattori di possesso: qualcosa che l'utente ha a disposizione;
- fattori intrinseci: qualcosa che rende unico l'utente come individuo;
- fattori comportamentali: qualcosa che l'utente fa.

Il tipo più comunemente usato di MFA è quello a due fattori (2FA, o Two-Factor Authentication). 2FA è un metodo di verifica dell'identità che richiede all'utente esattamente due fattori di autenticazione, di cui uno può essere una password, per accedere a un sito web, un'applicazione o una rete [S19].

2 | Sviluppare un'API RESTful Sicura in .NET Core

Nel seguente capitolo verrà sviluppato l'oggetto del progetto; si farà dunque vedere come è possibile implementare in modo sicuro un'API Web che segue i principi architetturali REST. L'ambiente che si prende come riferimento è quello Microsoft, infatti, l'API verrà implementata utilizzando il framework .NET Core 6 [S20]; mentre lo sviluppo e il test avranno luogo in una macchina Windows 11. Di conseguenza, i framework o pacchetti utilizzati sono stati scelti su tali considerazioni e per scrivere il codice si è utilizzato l'IDE Visual Studio 2022 [S21].

2.1 Implementare un'API REST

In primo luogo è necessario definire l'API Web che si vuole andare a proteggere. Per questo progetto si intende costruire una semplice API Web RESTful che implementi le operazioni CRUD (Create-Read-Update-Delete) e che vada a gestire degli itinerari di viaggio. Si inizia aggiungendo al progetto un Controller `ItineraryController.cs` in cui si definiranno i seguenti *endpoint*

```
[Route("api/itineraries")]
public class ItineraryController
{
    [HttpGet]
    public IActionResult GetAllItineraries()
    {
        // Some code ...
    }

    [HttpGet("{id}")]
    public IActionResult GetItineraryById(Guid Id)
    {
```

```
        // Some code ...
    }

    [HttpPut]
    public IActionResult EditItineraryById([FromBody] ItineraryTest itinerary)
    {
        // Some code ...
    }

    [HttpPost]
    public IActionResult CreateNewItinerary([FromBody] ItineraryTest itinerary)
    {
        // Some code ...
    }

    [HttpDelete("{id}")]
    public IActionResult DeleteItineraryById(Guid Id)
    {
        // Some code ...
    }
}
```

Dal momento che si vuole rendere sicura questa API, è necessario definire delle regole di autorizzazione per accedere agli *endpoint*. A tale scopo, .NET mette a disposizione il filtro `Authorize` che viene valutato prima di raggiungere l'*endpoint*. Questo filtro verifica se chi lo sta invocando sia un utente autenticato e, nel caso non lo fosse, restituisce una risposta HTTP con codice di stato 401 (Unauthorized). Per proteggere ognuno degli *endpoint* messi a disposizione dall'API è possibile applicare il filtro `Authorize` globalmente facendo ereditare ad `ItineraryController` una classe base che verrà chiamata `ApiController`:

```
// ApiController.cs
[ApiController]
[Authorize]
public class ApiController : ControllerBase
{
    // Some code ...
}
```

```
// ItineraryController.cs
[Route("api/itineraries")]
public class ItineraryController : ApiController
{
    // Endpoints definition ...
}
```

Una volta specificate le regole di accesso agli *endpoint* è necessario definire il modo con cui validare una richiesta. Per dimostrare di essere autenticato, l'utente deve fornire un token d'accesso JWT specificando lo schema di autenticazione Bearer. Il compito dell'API è quello di definire le regole da seguire durante la validazione del token di accesso. Il servizio di autenticazione basato su schema Bearer può essere definito all'interno della classe `Program.cs`. Di seguito viene mostrato il codice che aggiunge un servizio di autenticazione con schema di autenticazione JWT Bearer:

```
// Program.cs
builder.Services
    .AddAuthentication(defaultScheme: JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, options =>
        {
            options.SaveToken = true;
            options.TokenValidationParameters
                = new TokenValidationParameters()
            {
                ValidateIssuer = true,
                ValidateAudience = true,
                ValidateLifetime = true,
                ValidateIssuerSigningKey = true,
                ValidIssuer = jwtSettings.Issuer,
                ValidAudience = jwtSettings.Audience,
                IssuerSigningKey = new X509SecurityKey(
                    new X509Certificate2(certSettings.Location))
            };
        }
    );
```

Si può notare come il servizio di autenticazione JWT Bearer consenta di specificare alcune opzioni. Ad esempio, è molto importante definire i parametri per la validazione del token istanziando l'oggetto `TokenValidationParameters`. Di seguito si descrivono i parametri

utilizzati:

- Con `ValidateIssuer = true` si richiede al servizio di autenticazione di verificare l'Issuer del token, ovvero chi lo ha generato. Nel parametro `ValidIssuer` si specifica l'URI dell'Issuer.
- Con `ValidateAudience = true` si richiede di verificare l'Audiance, ossia a chi è rivolto il token JWT. In `ValidAudience` si specifica un riferimento al destinatario, ad esempio, `bdcrApi` per indicare che quel token è destinato all'intera API.
- Impostando a `true` il parametro `ValidateLifetime` si richiede di verificare la validità del token JWT da un punto di vista temporale.
- Il parametro `ValidateIssuerSigningKey` impostato a `true` indica la volontà di validare la chiave che l'Issuer ha utilizzato per firmare il token. A `IssuerSigningKey` si assegna la chiave asimmetrica (`new X509SecurityKey()`) basata su quella pubblica, condivisa in un primo momento dall'Issuer stesso. X.509 è uno standard utilizzato per definire il formato dei certificati TLS/SSL a chiave pubblica.

Successivamente si possono specificare delle policy di autorizzazione per ognuno degli *endpoint*. All'interno del *payload* di un token JWT possono essere aggiunti Claim che riguardano le proprietà di quell'utente che sta cercando di accedere alle risorse. Ad esempio, è possibile specificare un Claim `position` che esprima uno tra i seguenti ruoli che l'utente può assumere: `User` o `Administrator`. Immaginando di voler garantire l'accesso ad alcuni *endpoint* solo agli utenti con ruolo di `Administrator`, è possibile definire una policy come `AdminCRUDPolicy`:

```
// Program.cs
builder.Services
    .AddAuthorization(authOpt =>
    {
        authOpt.AddPolicy("AdminCRUDPolicy", policyBuilder =>
        {
            policyBuilder.RequireAuthenticatedUser();
            policyBuilder.RequireClaim("position", "Administrator");
            policyBuilder.RequireClaim("country", "ITA");
        });
    });
```

La policy `AdminCRUDPolicy` richiede che l'utente sia autenticato (`RequireAuthenticatedUser`), quindi in possesso di un token di accesso che abbia superato l'autenticazione Bearer speci-

ficata in precedenza. In secondo luogo, si prendono dal token i Claim `position` e `country`, i quali devono essere valorizzati rispettivamente con `Administrator` e `ITA`. Per applicare questa policy basterà modificare il filtro `Authorization` per l'*endpoint* che si vuole proteggere. Ad esempio, se si vuole dare il permesso di creare nuovi itinerari di viaggio solamente agli utenti che rispettano la policy, si può scrivere:

```
// ItineraryController.cs
[HttpPost("create")]
[Authorize(Policy = "AdminCRUDPolicy")]
public IActionResult CreateNewItinerary([FromBody] ItineraryTest itinerary)
{
    // Some code ...
}
```

Infine, è necessario definire la *pipeline* che implementa autenticazione e autorizzazione. Nel file `Program.cs` si aggiungono quindi i seguenti middleware:

```
var app = builder.Build();
{
    app.UseHttpsRedirection();

    app.UseAuthentication();
    app.UseAuthorization();
}
```

Si fa notare inoltre l'importanza del middleware `UseHttpsRedirection`. Questo infatti forza l'utilizzo del protocollo HTTPS per scambiare messaggi. Tutte le richieste fatte mediante HTTP ora verranno reindirizzate su HTTPS rendendo privata la connessione.

2.2 Implementare un Gateway API con Ocelot

Ocelot API Gateway [S22] consiste in una collezione di middleware altamente configurabili, scritti specificatamente per l'ambiente .NET Core. La configurazione di Ocelot viene definita in un file tipicamente chiamato `ocelot.json`. Ocelot, mette a disposizione un middleware che crea un oggetto `HttpRequestMessage` che viene utilizzato per fare la richiesta all'API che si vuole nascondere, a cui si fa riferimento come *downstream service*. Questo middleware è posizionato in fondo alla *pipeline* definita da Ocelot. Una volta ricevuta la risposta dall'API, Ocelot impiega un diverso middleware per mappare l'oggetto `HttpResponseMessage` all'interno dell'`HttpResponse`; questo verrà inoltrato al Client da

cui Ocelot ha ricevuto la richiesta. Ocelot permette di risolvere perfettamente il problema del disaccoppiare Client e Server poiché capace di funzionare come *reverse proxy*.

All'interno del file `ocelot.json` è possibile configurare tutto il necessario per far funzionare il Gateway API. Come prima cosa Ocelot ha bisogno di sapere su che macchina viene eseguito. Quindi si definisce il parametro `BaseUrl`, il quale rappresenta l'URI del Gateway API che verrà contattato dai Client:

```
{
    "GlobalConfiguration": {
                "BaseUrl": "https://localhost:5003"
    }
}
```

Per quanto riguarda la configurazione dei singoli *endpoint*, è possibile specificare un oggetto JSON per ognuno di essi. Questi oggetti dovranno essere contenuti all'interno della lista JSON `Routes`. Ad esempio di seguito si aggiunge la configurazione per l'*endpoint* dell'API che permette di creare un nuovo itinerario di viaggio:

```
"Routes": [
   {
       "UpstreamPathTemplate": "/gateway/itineraries",
       "UpstreamHttpMethod": [ "Post" ],
       "DownstreamPathTemplate": "/api/itineraries",
       "DownstreamScheme": "https",
       "DownstreamHostAndPorts": [
           {
               "Host": "localhost",
               "Port": 2024
           }
        ]
    },
]
```

In questo caso, si sta specificando al Gateway API come trattare una richiesta proveniente da un Client, detta *upstream request*. Di seguito si descrivono i parametri utilizzati:

- Con `UpstreamPathTemplate` Ocelot espone verso i Client l'*endpoint* specificato. Il Gateway gestirà questa *upstream request* andando ad invocare il metodo `POST` che l'API espone con l'*endpoint* specificato in `DownstreamPathTemplate`.

- Se impostato a `https`, il parametro `DownstreamScheme` forza l'utilizzo del protocollo HTTPS da parte del Gateway API; quindi, si garantisce che la comunicazione verso l'API sia sicura e privata.
- In `DownstreamHostAndPorts` si definisce un oggetto JSON specificando `Host` e `Port` così che il Gateway sappia che server e che porta contattare.

Controllo degli accessi

Per regolare gli accessi verso gli *endpoint* è possibile aggiungere ad una configurazione le regole

```
"AuthenticationOptions": {
    "AuthenticationProviderKey": "Bearer"
},
"RouteClaimsRequirement": {
    "position": "Administrator",
    "country": "ITA"
}
```

dove:

- Con l'oggetto JSON `AuthenticationOptions` è possibile configurare Ocelot con diverse opzioni di autenticazione. Ad esempio si può richiedere ai Client che contattano questo *endpoint* di fornire un token JWT così da poterlo validare secondo lo schema Bearer. In questo caso, l'autenticazione con schema Bearer a cui si fa riferimento è quella definita sul Gateway stesso e non nell'API. Per questo motivo è necessario aggiungere il servizio di autenticazione che si vuole utilizzare e validare il token JWT anche a questo livello.
- L'oggetto JSON `RouteClaimsRequirement` permette di specificare quali sono i Claim richiesti al token JWT fornito dal Client affinché il Gateway possa accettare l'*upstream request*. Ad esempio, in questo caso si vuole verificare che il token del Client abbia i Claim `position` e `country` valorizzati rispettivamente come `Administrator` e `ITA`. Infatti, come si è visto nella sezione precedente, l'API che si sta proteggendo definisce proprio una policy basata su questi due Claim così valorizzati. Si fa notare che spostare la fase di validazione dei token JWT a questo livello rende molto più performante, oltre che sicuro, il servizio offerto dall'API.

Quality Of Service e Rate Limit

Ocelot integra la libreria Polly per .NET così da permettere di definire una configurazione anche per la qualità del servizio. Per ogni *endpoint*, infatti, basterà aggiungere l'oggetto JSON `QoSOptions`, ad esempio:

```
"QoSOptions": {
    "ExceptionsAllowedBeforeBreaking": 3,
    "DurationOfBreak": 1000,
    "TimeoutValue": 5000
}
```

In questo caso si sta specificando che, per un determinato *endpoint*, il Gateway API non accetta più di tre eccezioni di servizio superate le quali interrompe l'*upstream request*. Il parametro `TimeoutValue`, invece, definisce la durata massima di una *upstream request*. Nell'esempio sopra riportato, se la richiesta non viene soddisfatta entro cinque secondi, il Gateway la interrompe restituendo al Client un messaggio HTTP con codice di stato 503 (Service Unavailable).

Ocelot permette anche di definire per ogni *endpoint* delle *quota* assieme a meccanismi di *throttling*. A tale scopo, è possibile definire un oggetto JSON `RateLimitOptions` specificando i seguenti parametri:

```
"RateLimitOptions": {
    "EnableRateLimiting": true,
    "Period": "10s",
    "PeriodTimespan": 10,
    "Limit": 3
},
```

In questo modo si sta dicendo al Gateway API di non accettare più di tre *upstream request* fatte da un Client all'interno di un intervallo di tempo lungo dieci secondi.

Per ogni endpoint dell'API, è possibile specificare quali schemi di autenticazione vengano usati e quali autorizzazioni siano necessarie. Ad esempio, consideriamo l'endpoint che permette di creare gli itinerari. Nel file `ocelot.json` si aggiunge il seguente oggetto (sotto `Routes`):

```
{
    "UpstreamPathTemplate": "/gateway/itineraries/create",
    "UpstreamHttpMethod": [ "Post" ],
    "DownstreamPathTemplate": "/api/itineraries/create",
```

```
    "AuthenticationOptions": {
        "AuthenticationProviderKey": "Bearer"
    },
    "RouteClaimsRequirement": {
        "position": "Administrator",
        "country": "ITA"
    }
}
```

2.3 Implementare un Identity Server OAuth 2.0 con Duende

Duende IdentityServer [S23] è un framework sviluppato per ASP.NET Core che implementa la famiglia di protocolli OAuth 2.0 e OpenID Connect. Questo framework serve, infatti, per implementare un Identity Server, vale a dire un software che decide se rilasciare token di accesso ai Client che li chiedono. Tra le funzioni più importanti che si possono implementare con Duende IdentityServer vi è la gestione dell'accesso alle risorse, l'autenticazione degli utenti, l'autenticazione dei Client ed il rilascio di Access Token.

2.3.1 Definire le risorse

Duende, permette di configurare le risorse dell'Identity Server all'interno di una classe comunemente chiamata `Config`.

Gli Scope

Uno degli strumenti principali su cui si basa lo standard OAuth è il concetto di Scope. Gli Scope permettono di specificare, all'interno di un token JWT, il motivo per il quale si vuole accedere ad una risorsa. Questi Scope vengono decisi dal Client che contatta l'Identity Server per richiedere un Access Token. L'Identity Server che riconosce il Client, dovrà poi decidere quali, tra gli Scope richiesti, inserire nell'Access Token. Per registrare gli Scope nell'Identity Server si aggiunge alla classe `Config` il codice seguente:

```
// Config.cs
public static IEnumerable<ApiScope> ApiScopes =>
    new ApiScope[]
    {
        new ApiScope(name: "bdrcApi", displayName: "Progetto BDRC API")
```

```
    };
```

La scelta di progetto che si può compiere è quella di utilizzare lo Scope `bdrcApi`, ad esempio, per garantire l'autorizzazione ad ogni *endpoint* a meno della presenza di policy particolari. L'API sviluppata nelle sezioni precedenti potrà rispecchiare tale scelta di progetto aggiungendo il filtro `RequiredScope` nel Controller base `ApiController`:

```
// ApiController.cs
[ApiController]
[Authorize]
[RequiredScope("bdrcApi")]
public class ApiController : ControllerBase
{
    // Some code ...
}
```

I Client

La definizione dei Client che si vogliono registrare nell'Identity Server è essenziale poiché solo questi verranno riconosciuti e quindi avranno la possibilità di accedere alle risorse dell'API. Questi rappresentano applicazioni che possono richiede gli Access Token all'Identity Server. Ogni Client può essere configurato attraverso diverse opzioni, ad esempio:

```
// Config.cs
public static IEnumerable<Client> Clients =>
    new Client[]
    {
        new Client
        {
            ClientId = "your-client-id",
            ClientName = "BDRC Client App",
            RequireClientSecret = true,
            ClientSecrets =
            {
                new Secret("your-client-secret".Sha256())
            },
            AllowedScopes =
            {
                IdentityServerConstants.StandardScopes.OpenId,
```

```
                IdentityServerConstants.StandardScopes.Profile,
                "bdrcApi",
                "position",
                "country"
            },
        },
    };
```

Di seguito si passa a descrivere i parametri più degni di nota:

- `ClientId` e `ClientSecrets` sono i principali strumenti utilizzati dall'Identity Server per riconoscere un Client. I Secret rappresentano delle sorte di password condivise solo tra Identity Server e Client.
- La lista `AllowedScopes` definisce una collezione di Scope che il Client è autorizzato a richiedere.
  - `StandardScopes.OpenId`. Questo Scope è richiesto qualora il Client decida di autenticarsi all'Identity Server utilizzando il protocollo OpenId Connect. Implicitamente, segnala che al token JWT restituito al Client va aggiunto il Claim `sub` che identifica in modo univoco l'utente finale.
  - `StandardScopes.Profile`. Richiede di aggiungere al token JWT da restituire al Client i Claim rappresentanti informazioni sull'utente finale, ad esempio: `name`, `family_name`, `given_name`, `nickname`.
  - Gli Scope `bdrcApi`, `position` e `country` non sono standard e quindi il loro significato dipende dalle scelte di progetto. Ad esempio, data la definizione della policy `AdminCRUDPolicy` descritta nelle sezioni precedenti, gli Scope `position` e `country` richiedono che nel token JWT siano aggiunti i Claim `position` e `country`.

I Claim

I Claim sono i pezzi di informazione che andranno a costituire il *payload* del JWT emesso dall'Identity Server. La configurazione dei Claim è importante perché si dichiara nel token JWT quello su cui l'API si può basare per definire i meccanismi di autenticazione e autorizzazione. Ad esempio, nel Client precedentemente configurato si è definita l'IdentityResource `position`, per questo motivo l'Identity Server sa che può emettere nel token JWT il Claim `position` dell'utente. Per gestire l'emissione dei Claim, Duende permette di definire una classe `ProfileService` che implementi il metodo `GetProfileDataAsync`.

Ad esempio, nell'API degli itinerari si è aggiunta la policy `AdminCRUDPolicy` che richiedeva che nel token fossero presenti i Claim `position` e `country`. Il codice che si aggiunge è:

```
// ProfileService.cs
public class ProfileService : IProfileService
{
    public Task GetProfileDataAsync(ProfileDataRequestContext context){
        var user = context.Subject;

        var claims = new List<Claim>
        {
            new Claim("position", user.Claims
                .Where(x => x.Type.Equals("position")).First().Value),
            new Claim("country", user.Claims
                .Where(x => x.Type.Equals("country")).First().Value)
        };

        context.IssuedClaims.AddRange(claims);
    }
}
```

L'oggetto `ProfileDataRequestContext` rappresenta l'insieme delle informazioni che riguardano la richiesta corrente che l'Identity Server si trova a gestire. Si può ottenere ad esempio il Claim `sub` che identifica l'utente. In questo caso, si sta dicendo che se l'utente è in possesso dei Claim `postition` e `country` allora li aggiungiamo al token JWT da emettere.

2.3.2 Autenticazione con OpenID Connect

OpenID Connect (OIDC) estende il framework di autenticazione OAuth 2.0 rendendo più semplice la gestione dei token JWT e l'interazione tra Client e Identity Server. OpenID Connect definisce degli *endpoint* che possono essere contattati per richiedere l'autorizzazione, un Access Token oppure informazioni sull'utente in modo semplice e con modalità familiari ai principi architetturali REST.

ID Token

L'estensione principale che OpenID Connect apporta a OAuth 2.0 è la struttura dati detta ID Token. L'ID Token è un token di sicurezza che contiene Claim riguardanti l'autenticazione di un utente finale forniti da un Authentication Server al Client che li ha richiesti. L'ID Token è rappresentato come un token JWT. I Claims utilizzati all'interno di un ID Token per il flusso di autenticazione OAuth 2.0 e OpenID Connect sono specificati nella documentazione OIDC [S24], ma tra i più importanti si citano: `iss` (Issuer Identifier), `sub` (Subject Identifier), `aud` (Audience), `exp` (Expiration time), tutti obbligatori.

Hybrid Flow Authentication

Per questo progetto si sceglie di implementare l'autenticazione OIDC utilizzando il modello detto Hybrid Flow [S25]. Questo modello consente di avere accesso immediato all'ID Token che potrà essere impiegato quando un'applicazione Client ha bisogno di ottenere immediatamente l'accesso alle informazioni su un utente. Inizialmente, si deve dichiarare che un Client vuole utilizzare questo metodo di autenticazione. Nel file `Config.cs` dell'Indentity Server si definisce il parametro `AllowedGrantTypes` all'interno della configurazione del Client dichiarato precedentemente:

```
new Client
{
    ClientName = "BDRC Client App",
    AllowedGrantTypes = GrantTypes.Hybrid,
    RequirePkce = false,

    // Others configurations ...
}
```

Nel metodo Hybrid Flow gli endpoint che emettono i token sono due: Authorization Endpoint e Token Endpoint. Le fasi per l'autenticazione definite dal metodo Hybrid Flow sono:

1. il Client genera una Authentication Request e la manda all'Authorization Server;
2. l'Authorization Server autentica e ottiene l'autorizzazione per l'utente finale;
3. l'Authorization Server restituisce al Client un Authorization Code;
4. il Client utilizza l'Authorization Code per fare una richiesta al Token Endpoint;

5. il Token Endpoint risponde al Client aggiungendo nel corpo della risposta un ID Token e l'Access Token;
6. il Client valida l'ID Token e recupera il parametro `SubjectIdentifier` dell'utente finale.

Quando si contattano tali *endpoint*, il Client deve specificare il tipo di richiesta OAuth 2.0 andando a valorizzare il parametro `response_type` definito da OAuth. Per specificare che il metodo è Hybrid Flow e il flusso di autorizzazione voluto è quello appena descritto, si deve valorizzare `response_type` con `"code id_token"`. L'Authrization Endpoint risponderà alla richiesta di autorizzazione del Client aggiungendo i seguenti parametri come *query* nell'URI, detto Redirect Uri, specificato dal Client. Se in `Config.cs` dell'Identity Server si aggiunge alla configurazione del Client la seguente opzione:

```
new Client
{
    ClientName = "BDRC Client App",
    AllowedGrantTypes = GrantTypes.Hybrid,
    RequirePkce = false,
    RedirectUris =
    {
        "https://localhost:7235/signin-oidc"
    },

    // Others configurations ...
}
```

allora l'Authorization Endpoint restituirà una risposta di successo come quella seguente:

```
https://localhost:7235/signin-oidc
?code=SplxlOBeZQQYbYS6WxSbIA
&id_token=eyJ0 ... NiJ9.eyJ1c ... I6IjIifX0.DeWt4Qu ... ZXso
```

Il Client deve ora ottenere l'Access Token contattando il Token Endpoint e fornendo nella richiesta l'Authorization Code e l'ID Token. Il `response_type` sarà lo stesso: `"code id_token"`. Il Token Endpoint valida l'ID del Client, poi si assicura che l'Authorization Code sia stato emesso dall'Authorization Endpoint e che l'ID Token fornito non sia stato alterato. A questo punto, il Token Endpoint restituisce una Token Response contenente, tra le altre cose, l'Access Token, il tipo di token impostato come Bearer e l'ID

Token. Quando il Client valida l'Access Token, potrà utilizzarlo per autenticare l'utente e accedere alle risorse protette dell'API.

Implementazione di un Client ASP.NET MVC

Per autenticare l'utente si ha bisogno di un'applicazione Client che comunichi con l'Identity Server. A questo punto, nel file `Program.cs` del Client è necessario configurare i servizi di autenticazione così da poter correttamente ricevere l'Access Token con cui contattare l'API.

```
// Program.cs
builder.Services.AddAuthentication(opt =>
{
    opt.DefaultScheme = "Cookies";
    opt.DefaultChallengeScheme = "oidc";
})
.AddCookie("Cookies", opt =>
{
    opt.AccessDeniedPath = "/Account/AccessDenied";
})
.AddOpenIdConnect("oidc", opt =>
{
   opt.SignInScheme = "Cookies";
   opt.Authority = "https://localhost:7009";
   opt.ClientId = builder.Configuration["OidcConfig:ClientId"];
   opt.ResponseType = "code id_token";
   opt.SaveTokens = true;
   opt.ClientSecret = builder.Configuration["OidcConfig:ClientSecret"];
   opt.GetClaimsFromUserInfoEndpoint = true;

   opt.Scope.Add("bdrcApi");
   opt.Scope.Add("position");
   opt.Scope.Add("country");
   opt.ClaimActions.MapUniqueJsonKey("position", "position");
   opt.ClaimActions.MapUniqueJsonKey("country", "country");

   opt.TokenValidationParameters = new TokenValidationParameters
   {
```

```
        RoleClaimType = "position",
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new X509SecurityKey(
                new X509Certificate2(Path.Combine(".", "keys", "AuthSample.cer")))
    };
});
```

Si fa notare che quando si aggiungono i servizi di autenticazione con `AddAuthentication`, il parametro `DefaultScheme` viene impostato come `cookies` mentre `DefaultChallengeScheme` come `oidc`. Questo perché il `DefaultChallengeScheme` viene utilizzato quando gli utenti non autenticati richiedono di fare il login. Quindi, avendolo impostato a `oidc` viene inizializzato il protocollo OIDC instradando l'utente verso l'Identity Server. Quando l'Identity Server autentica l'utente e lo rimanda al Client, quest'utlimo salva dei cookie localmente. Per gli utenti autenticati, infatti, il Client utilizza il servizio `AddCookie` per gestire i cookie locali. `AddOpenIdConnect` gestisce l'autenticazione dell'utente secondo il protocollo OIDC. Di seguito si va a descrivere la configurazione del Client:

- `Authority`: indica al protocollo dove trovare l'Identity Server che genera il token.
- `ClientId` e `ClientSecret`: identificano il Client e devono essere registrati nell'Identity Server.
- `Scope`: si aggiungono gli Scope che il Client intende richiedere. L'Identity Server restituirà un errore nel caso in cui il Client richieda Scope a cui non ha accesso. Gli scope `oidc` e `profile` vengono automaticamente aggiunti.
- `SaveTokens`: serve per indicare al Client di salvare localmente i token ricevuti così da poterli usare in un secondo momento.
- `ResponseType`: si chiede di utilizzare l'Hybrid Flow con i tipi di ritorno `code` e `id_token`.
- `GetClaimsFromUserInfoEndpoint`: OIDC mette a disposizione uno UserInfo Endpoint che restituisce i Claim dell'utente autenticato. Impostando questo parametro a `true` si richiedono sempre i Claim dell'utente. L'utente deve essere già autenticato, poiché si ha bisogno dell'Access Token per contattare lo UserInput Endpoint e Claim `sub` per identificare l'utente.
- `TokenValidationParameters`: si specificano i parametri per effettuare la validazione dell'Access Token. In questo caso si vuole validare la `SignInKey` utilizzata dall'Identity Server per la *signature* del token.

2.3.3 Usare le chiavi per firmare gli Access Token

L'Identity Server si pone all'interno del progetto sviluppato in questa relazione come un servizio che genera Access Token e regola l'autenticazione degli utenti così come dei Client. Per un servizio di questo tipo è molto importante che l'Identity Server abbia a disposizione un certificato con cui potrà firmare i token emessi. Per aggiungere un certificato da utilizzare per firmare i token emessi è possibile utilizzare il metodo `AddDeveloperSigningCredential`, tuttavia, in un normale scenario si preferiscono utilizzare certificati reali rilasciati da una Certificate Authority. Si può indicare all'Identity Server di utilizzare un certificato X.509 aggiungendo il seguente codice:

```
builder.Services
    .AddIdentityServer()
    // Other configurations
    .AddSigningCredential(
        new X509Certificate2(certLocation, certPassword));
```

È possibile generare dei certificati auto-firmati attraverso i due strumenti a linea di comando: `makecert` [S26] e `pvk2pfx` [S27]. Come prima cosa si genera un certificato di test auto-firmato:

```
makecert -n "CN=AuthSample" -a sha256 -sv AuthSample.pvk -r AuthSample.cer
```

i parametri possono essere interpretati come segue:

- `-n` per assegnare il nome al certificato;
- `-a` per definire l'algoritmo di hashing;
- `-sv` per salvare la chiave privata all'interno del file `AuthSample.pvk`
- `-r` per salvare la chiave pubblica nel file `AuthSample.cer`.

A questo punto si combinano chiave pubblica e privata:

```
pvk2pfx -pvk AuthSample.pvk -spc AuthSample.cer -pfx  AuthSample.pfx
-pi [password]
```

Il file `AuthSample.pfx` è protetto da password poiché contiene la chiave privata che l'Identity Server utilizzerà per firmare i token emessi. È quindi importantissimo proteggere questo file e non renderlo pubblico. Il file `AuthSample.cer` contiene la chiave pubblica, e può essere condiviso con le altre componenti allo scopo di fargli validare i token in arrivo. Ad esempio, si è già potuto vedere come il Client aggiunga la validazione della `SignInKey`

nell'autenticazione, utilizzando il file `AuthSample.cer`, quindi la chiave pubblica. La stessa cosa viene fatta nell'API e nel Gateway API.

2.3.4 Aggiungere l'autenticazione a due fattori

Nel primo capitolo si è parlato dei vantaggi di utilizzare un'autenticazione a due fattori. In questa sezione verrà implementata un'autenticazione a due fattori in cui il primo fattore è la password, mentre il secondo è un codice OTP (*One-Time-Password*) inviato per Email. L'OTP è un qualcosa che si conosce, come la password. Per semplicità però, si considera OTP come qualcosa che si ha, quindi un fattore di possesso, poiché, dal momento sarà inviato per Email, se si è in grado di conoscere l'OTP allora si presume che l'utente possegga l'account Email a cui è stato inviato.

Per abilitare la 2FA ogni utente viene registrato all'Identity Server con il parametro `TwoFactorEnabled = true` e, in seguito alla registrazione, gli sarà chiesto di verificare la sua Email.

Nell'Identity Server si definisce un controller `AccountController`. Qui aggiungiamo due *endpoint* che implementano il *login* a due fattori:

```
// AccountController.cs
[HttpGet]
public async Task<IActionResult> LoginTwoStep(
    string username, bool rememberMe, string returnUrl = null)
{
    // Code to
    // - get ValidTwoFactorProviders
    // - generate TwoFactorToken
    // - send email
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> LoginTwoStep(
    TwoStepModel twoStepModel, string returnUrl = null)
{
    // Code to authenticate the user with two factors
}
```

3 | Conclusioni

Il sistema sviluppato è stato pensato con l'obbiettivo di proteggere le risorse messe a disposizione da un'API da accessi non autorizzati. Per fare ciò si è ritenuto necessario l'impiego di quattro componenti: API REST, Gateway API, Client, Identity Server.

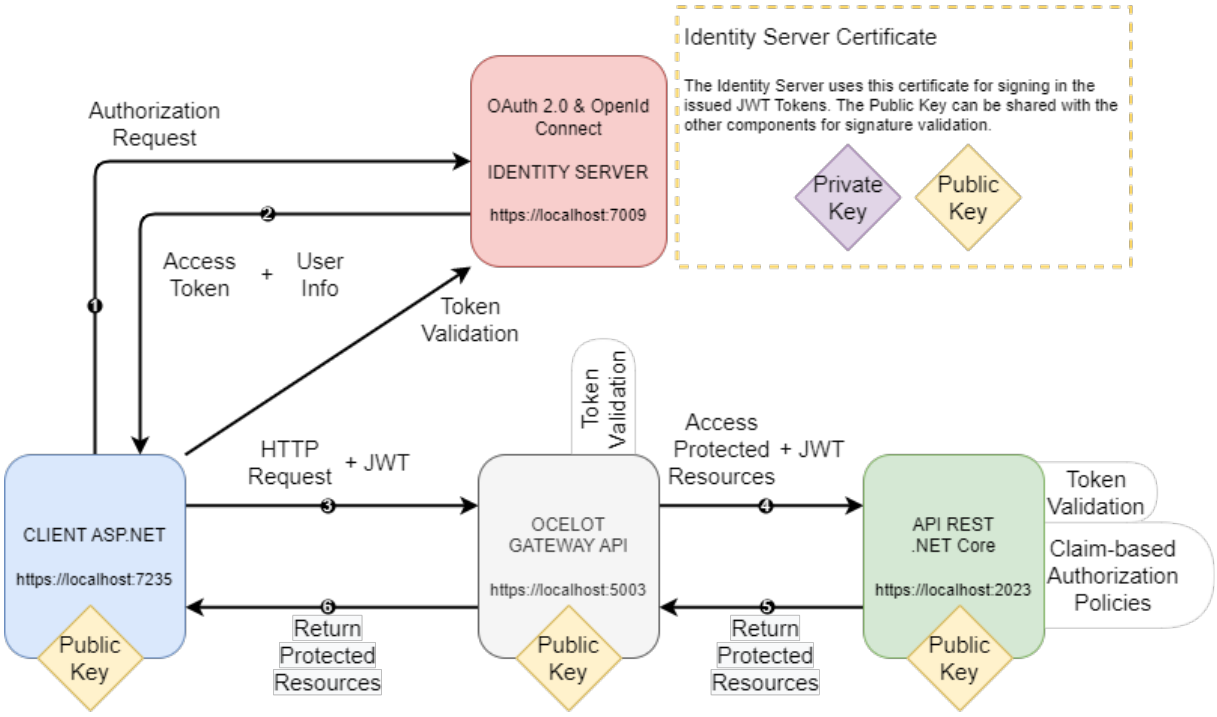

Figura 3.1: Architettura del progetto sviluppato. La figura mostra il flusso di autenticazione e di richiesta delle risorse dell'API da parte del Client.

Con questa architettura (vedi figura 3.1) si è riusciti a raggiungere un ottimo livello di protezione delle risorse messe a disposizione da un'API. Si fa notare come sia importante che un buon livello di sicurezza sia in primo luogo implementato nel Client. In secondo luogo, è importante configurare l'Identity Server con i giusti criteri, infatti, questo fornisce le autorizzazioni agli utenti ed è fondamentale che queste rispecchino i principi di protezione definiti nell'API che si sta proteggendo. Si è visto come queste configurazioni possano basarsi su Claim e Scope. Infine, i vantaggi che si sono riscontrati aggiungendo un

Gateway API sono molteplici ed aumenterebbero se si suddividesse l'API in microservizi. A prescindere dall'architettura costruita intorno all'API, questa rimane l'ultimo livello di protezione prima di raggiungere le risorse. Di conseguenza, vale la pena impiegare sforzi maggiori nella definizione dei metodi di autenticazione e autorizzazione a questo livello.

D'altra parte la complessità di un'architettura costruita in questo modo è sicuramente maggiore rispetto ad una che prevede che tutto sia gestito dall'API, in cui il Client riceverebbe l'Access Token dall'API stessa e la comunicazione tra le due parti sarebbe diretta.

Vista l'enorme quantità di benefici che essa porta è bene suddividere i ruoli in componenti distinti ed indipendenti. Le tecnologie adottate permettono di mantenere l'architettura facilmente scalabile e mantenibile.

È inoltre importante aggiungere che per ognuno di questi componenti non è stato sfruttato a pieno il potenziale dato che il progetto sviluppato è relativamente semplice rispetto ad uno scenario reale. Questi strumenti sono infatti anche capaci di adattarsi a scenari molto più complicati poiché altamente configurabili.

Sitografia

[S1] T. Petrelli. *GitHub Repository: Progetto_BDRC_SessioneEstiva_2022_2023*. https://github.com/petrello/Progetto_BDRC_SessioneEstiva_2022_2023, 2023.

[S2] Red Hat. *What is a REST API?* https://www.redhat.com/it/topics/api/what-is-a-rest-api, 2020.

[S3] Wikipedia. *SOAP*. https://en.wikipedia.org/wiki/SOAP, 2023.

[S4] IBM. *What is a REST API?* https://www.ibm.com/it-it/topics/rest-apis, 2023.

[S5] Amazon AWS. *What is a RESTful API?* https://aws.amazon.com/it/what-is/restful-api, 2023.

[S6] Wikipedia. *API*. https://en.wikipedia.org/wiki/API, 2023.

[S7] Avast. *Cross-site scripting (XSS)*. https://www.avast.com/it-it/c-xss, 2023.

[S8] Avast. *DDoS - Distributed Denial of Service*. https://www.avast.com/it-it/c-ddos, 2023.

[S9] Avast. *SQL injection*. https://www.avast.com/it-it/c-sql-injection, 2023.

[S10] Avast. *Spoofing*. https://www.avast.com/it-it/c-spoofing, 2023.

[S11] T. Siddiqui. *Securing Web APIs*. https://www.developer.com/web-services/securing-web-apis, 2021.

[S12] Red Hat. *Cos'è un gateway API?* https://www.redhat.com/it/topics/api/what-does-an-api-gateway-do, 2019.

[S13] M. Jones J. Bradley N. Sakimura. *JSON Web Token (JWT)*. https://datatracker.ietf.org/doc/html/rfc7519, 2015.

[S14] JWT.io. *Introduction to JSON Web Tokens*. https://jwt.io/introduction, 2023.

[S15] IANA. *JSON Web Token (JWT)*. https://www.iana.org/assignments/jwt/jwt.xhtml, 2023.

[S16] D. Hardt. *The OAuth 2.0 Authorization Framework.* https://datatracker.ietf.org/doc/html/rfc6749, 2012.

[S17] N. Sakimura J. Bradley M. Jones B. de Medeiros C. Mortimore. *OpenID Connect Core 1.0.* https://openid.net/specs/openid-connect-core-1_0.*html*, 2014.

[S18] IBM. *Autenticazione a più fattori.* https://www.ibm.com/it-it/topics/multi-factor-authentication, 2023.

[S19] IBM. *Autenticazione a due fattori.* https://www.ibm.com/it-it/topics/2fa, 2023.

[S20] Microsoft. *.NET.* https://dotnet.microsoft.com, 2023.

[S21] Microsoft. *The Visual Studio 2022 IDE.* https://visualstudio.microsoft.com, 2023.

[S22] ThreeMammals. *Ocelot.* https://ocelot.readthedocs.io/en/latest/index.html, 2023.

[S23] Duende Software. *Duende IdentityServer.* https://docs.duendesoftware.com/identityserver/v6, 2023.

[S24] N. Sakimura J. Bradley M. Jones B. de Medeiros C. Mortimore. *ID Token.* https://openid.net/specs/openid-connect-core-1_0.htmlIDToken, 2014.

[S25] N. Sakimura J. Bradley M. Jones B. de Medeiros C. Mortimore. *Authentication using the Hybrid Flow.* https://openid.net/specs/openid-connect-core-1_0.*html*$HybridFlowAuth$, 2014.

[S26] Microsoft. *Makecert.* https://learn.microsoft.com/it-it/windows/win32/seccrypto/makecert, 2023.

[S27] Microsoft. *Pvk2Pfx.* https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/pvk2pfx, 2023.