

A thick dark blue vertical bar is positioned on the left side of the page. From its base, several thin, curved lines in shades of blue and grey extend upwards and outwards, creating an abstract, organic shape.

Sviluppo di un'Applicazione MVC RESTful con OAuth 2.0 in ASP.NET

Progetto del corso di Sicurezza delle Architetture Orientate ai Servizi

Stefano Belloni (matricola 940842)
Riccardo Longoni (matricola 943531)
UNIVERSITÀ DEGLI STUDI DI MILANO

Indice

Introduzione	3
Tecnologie utilizzate	4
Asp.Net.....	4
Il Pattern Model View Controller (MVC)	4
Visual Studio.....	5
Git Hub	5
Bearer Token.....	5
OAuth2.0	6
Web Service RESTful	7
Applicazione.....	8
Come è stato impostato il progetto	8
Client dell'applicazione	8
OAuth Server.....	8
Resource Server	9
Come si presenta l'applicazione.....	10
Dall'applicazione al codice	15
Codice dell'applicazione	15
Folder Appstart.....	15
Classe Startup.Auth.....	15
Metodo ConfigureOAuthTokenGeneration	16
Classe IdentityConfig	17
Metodo SendAsync	17
Classe WebApiConfig	17
Metodo Register.....	17
Folder Controllers	18
Classe AccountController	18
Metodo Register.....	18
Metodo Login	19
Metodo Logoff.....	20
Classe HomeController	21
Metodo Catalogo.....	21
Classe ManageController	21
Metodo AddPhoneNumber	22
Metodo EnableTwoFactorAuthentication	23
Metodo DisableTwoFactorAuthentication	23

Metodo VerifyPhoneNumber	24
Metodo RemovePhoneNumber	25
Metodo ChangePassword	25
Metodo SetPassword	26
Classe ScialuppaController	27
Metodo AggiungiProdotto	27
Metodo LaMiaScialuppa	27
Metodo RimuoviProdotto	28
Folder OAuthServer.....	29
Classe JWTFormat.....	29
Metodo Protect.....	29
Metodo Encode	31
Classe OAuthServiceProvider	31
Folder ResourceServer.....	32
Classe Authorizer.....	32
Classe TokenManager	32
Metodo ControllToken.....	33
Metodo isValid	33
Metodo Decode.....	34
Metodo getRefreshToken	34
Folder Utils.....	35
Classe DBSpeaker	35
Metodo connectToDB	35
Metodo DMLOperation.....	35
Metodo SelectAll	36
Metodo GetQuantitaDisponibile.....	36
Test Generazione Token con Postman.....	36
Riferimenti	40

Introduzione

L'obiettivo del progetto è la creazione di un servizio REST con tecnologia OAuth 2.0 che consente l'accesso mediante il two-factor authentication e il Single sign-on (SSO) attraverso l'utilizzo di Google, Twitter e Facebook.

Al fine del raggiungimento dell'obiettivo, abbiamo sviluppato un'applicazione web in grado di gestire un marketplace online; per l'accesso, sarà richiesto all'utente (client) di registrarsi alla web-app attraverso la compilazione di una form dove dovrà inserire un proprio indirizzo mail e una password. Una volta registrato, l'utente potrà abilitare nella propria area personale l'autenticazione a due fattori, in cui dovrà inserire il numero di telefono sulla quale verrà inviato il codice da inserire ogni volta che esegue l'accesso all'applicazione.

Come alternativa ai tradizionali username e password, l'utente potrà scegliere come metodo di autenticazione l'utilizzo del Single Sign On sfruttando i servizi di Google, Twitter e Facebook. La gestione della login prevede anche la possibilità di cambiare la password accedendo alla propria area personale. Solo una volta eseguita l'autenticazione, sarà possibile visionare il catalogo dei prodotti ed effettuare acquisti.

Tecnologie utilizzate

Asp.Net

.NET [1] è una piattaforma per sviluppatori composta da strumenti, linguaggi di programmazione e librerie per la creazione di diverse tipologie di applicazione.

La piattaforma di base fornisce componenti comuni che vengono applicate alle diverse tipologie di app.

Framework aggiuntivi come ASP.NET, utilizzato per lo sviluppo del progetto, estendono .NET con componenti per la creazione di specifiche app. Nel nostro caso, ASP.NET è stato il framework adatto poiché consente lo sviluppo di Web app.

Alcune cose incluse nella piattaforma .NET sono:

- linguaggio di programmazione C# utilizzato da noi
- librerie di base per lavorare con stringhe, dati, file di I/O (input e output)

ASP.NET estende la piattaforma .NET con strumenti e librerie specifici per la creazione di app Web.

Di seguito un breve elenco di funzionalità che ASP.NET aggiunge alla piattaforma .NET:

- Framework di base per l'elaborazione delle richieste Web in C#
- Sintassi dei modelli di pagine Web, nota come Razor, per la creazione di pagine Web dinamiche utilizzando C#
- Librerie per pattern Web, come Model View Controller (MVC)
- Sistemi di autenticazione che includono librerie, database e modelli di pagine per la gestione degli accessi, inclusa l'autenticazione a più fattori (nel nostro caso a due fattori) e l'autenticazione esterna con Google, Twitter e altro.
- Estensioni dell'editor per fornire l'evidenziazione della sintassi, il completamento del codice e altre funzionalità specifiche per lo sviluppo di pagine web

Visto che ASP.NET estende .NET, è possibile utilizzare i numerosi pacchetti e librerie disponibili per tutti gli sviluppatori .NET; l'ambiente offre anche la possibilità di creare delle proprie librerie che possono essere condivise tra tutte le applicazioni scritte sulla piattaforma .NET.

Razor fornisce una sintassi per la creazione di pagine Web dinamiche utilizzando HTML e C#. Il codice C# viene valutato sul server e il contenuto HTML risultante viene inviato all'utente.

Il codice che viene eseguito lato client è scritto in Javascript.

Le applicazioni ASP.NET possono essere sviluppate ed eseguite su Windows, Linux, macOS e Docker.

Il Pattern Model View Controller (MVC)

Model-View-Controller (MVC) [2] [3] è un pattern utilizzato nella programmazione ASP.NET per dividere il codice in blocchi dalle funzionalità ben distinte.

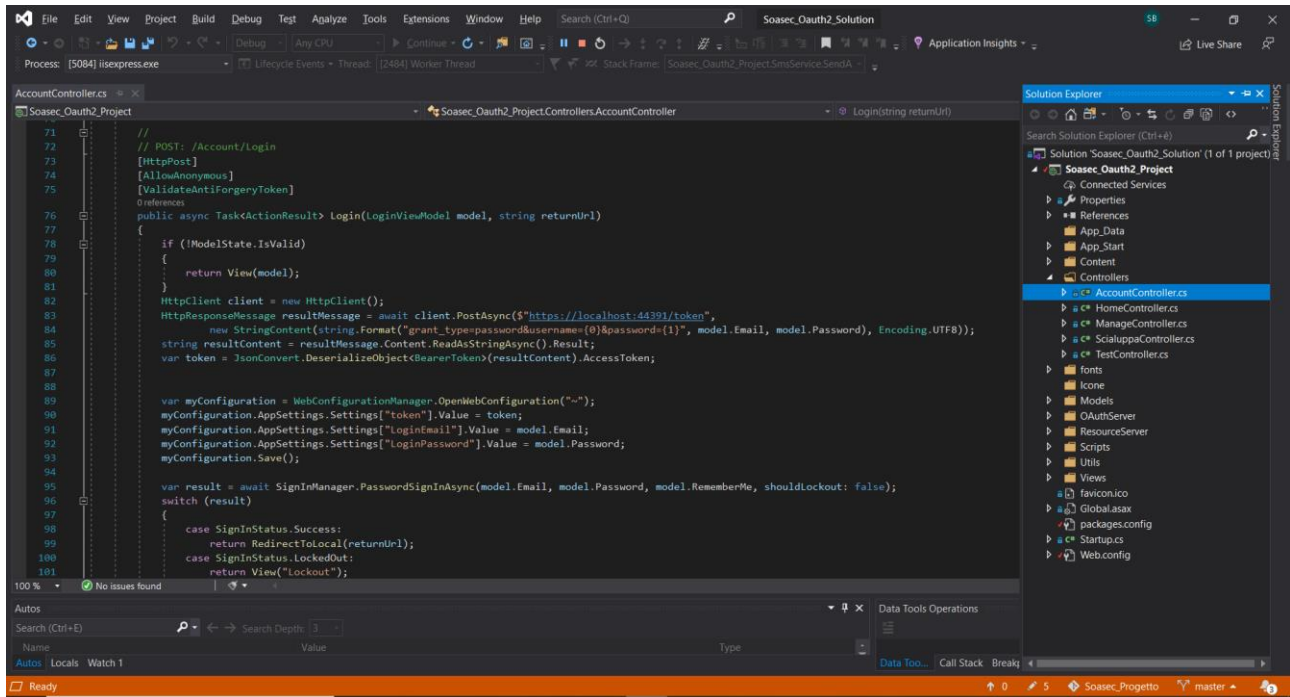
Infatti, quando si crea un progetto ASP.NET MVC, il lavoro viene diviso equamente tra:

1. **View.** Rappresenta l'interfaccia dell'applicazione (scritta in HTML e sintassi Razor) e rappresenta la parte front-end del progetto. Nello specifico, questo componente si occupa della visualizzazione dei dati che l'utente richiede e dell'interazione tra il client e l'infrastruttura sottostante della web app.
2. **Model.** Rappresenta i dati che il controller e view si scambiano. Nel progetto ASP.NET si tratta di una collezione di classi che andranno ad immagazzinare i dati che verranno estratti da un database.
3. **Controller.** Integra la logica sotto forma di codice C#, riceve i comandi del client attraverso il componente View andando ad eseguire delle azioni che in genere portano ad una alterazione dei dati presenti nel componente Model e un cambiamento di stato di View. Rappresenta la parte back-end. Al fine di raggiungere gli obiettivi posti dal progetto, l'attenzione principale è stata rivolta a questo componente poiché rappresenta il servizio che gestisce la parte di Login/Logout/Registrazione e SSO.

La particolarità di MVC è la netta separazione tra questi componenti. Infatti, ciascuno non conosce l'esistenza dell'altro, favorendo così una migliore testabilità del codice migliore rispetto ad approcci di sviluppo più tradizionali.

Visual Studio

Visual Studio [4] è un ambiente di sviluppo integrato (IDE) sviluppato da Microsoft che consente di modificare, eseguire il debug, compilare codice e pubblicare Web app. Rispetto ai più comuni editor e debugger disponibili nei diversi IDE, Visual Studio include compilatori, strumenti per il completamento del codice, finestre di progettazione con interfaccia grafica e altre funzionalità che semplificano il processo di sviluppo del software.



Aprendo un normale progetto di Visual Studio, ci troviamo di fronte una serie di finestre aperte come:

- *Esplora soluzioni* che consente di visualizzare, esplorare e gestire i file del codice; consente inoltre di organizzare il codice raggruppando i file in soluzioni e progetti
- *Finestra dell'editor* che visualizza il contenuto presente all'interno di un file; attraverso questa finestra è possibile modificare il codice o progettare un'interfaccia utente, come per esempio una finestra con pulsanti e caselle di testo
- *Team explorer* consente di tenere traccia degli elementi di lavoro e di condividere il codice con altri utenti usando tecnologie di controllo come Git

Git Hub

GitHub [5] è un servizio di hosting per progetti software, in cui gli sviluppatori caricano il codice sorgente dei loro programmi sulla rete e lo rendono scaricabile dagli utenti. Questi ultimi possono interagire con lo sviluppatore tramite un sistema di issue tracking che permette di migliorare il codice contenuto all'interno del repository risolvendo bug e aggiungendo anche nuove funzionalità.

Bearer Token

Il *Bearer Token* è un tipo di token utilizzato in fase di autenticazione dell'utente.

Il *bearer token* è una stringa criptata, generalmente generata dal server in risposta ad una richiesta di accesso. Il client deve inviare questo token nell'intestazione di autorizzazione quando effettua richieste a risorse protette:

Autorizzazione: Bearer <token>

Lo schema di autenticazione *Bearer* è stato originariamente creato come parte del protocollo di *OAuth 2.0* in RFC 6750, ma a volte viene utilizzato anche singolarmente. Analogamente all'autenticazione di base, la *Bearer authentication* deve essere utilizzata solo su HTTPS (SSL).

Il *Bearer Token* viene generato dall'authentication server. Quando un utente si autentica ad un'applicazione (client), il server di autenticazione genera un token; la creazione di questo token non avviene in maniera casuale, ma tiene in considerazione dell'utente e del client a cui viene fornito l'accesso.

Il token non ha una validità infinita, ma ha una scadenza breve (di solito circa un'ora, ma comunque è possibile sceglierne la durata); questo perché se avesse una durata molto lunga, comporterebbe un problema legato alla sicurezza poiché non vi sarebbe la possibilità di revocarlo in alcun modo. Prendiamo per esempio un utente con ruolo di amministratore che viene declassato come semplice utente, se mantiene il vecchio token con ruolo di amministratore, potrà accedere fino alla scadenza con tali diritti anche se risulta essere un semplice utente, ecco perché è necessario avere degli access token con scadenza breve.

Quando l'utente richiede un token al server, invia la propria login e password tramite SSL; come risposta il server restituisce un *Access token* e un *Refresh token*.

Il *Bearer token* viene utilizzato per generare un nuovo token di accesso; per ottenere quest'ultimo, viene inviato all'authentication server il *Refresh Token*, ne viene verificata la rispettiva validità e creato un nuovo token.

Un *Access token* è una stringa crittografata contenente alcune proprietà dell'utente, Claims e ruoli che si desiderano; ovviamente maggiore sarà la quantità di informazioni che contiene il token, maggiore sarà la sua dimensione. L'*Access Token* deve poi essere aggiunto a tutti gli header di richiesta per essere autenticato come un utente concreto.

Una volta che il *Resource Server* riceve un token di accesso, sarà in grado di decrittografarlo e leggere tutte le proprietà; in questo modo l'utente viene convalidato e gli viene fornito l'accesso all'applicazione.

Una domanda lecita che uno potrebbe porsi è: ma se un *Access Token* ha scadenza breve, dopo un tot periodo di tempo è necessario rinviare nuovamente utente e password ripetendo un'operazione di Login? Ovviamente quest'ultima non rappresenta una soluzione ottimale, sia per questioni legate alla sicurezza, ma anche perché ogni volta l'utente sarebbe costretto a ripetere in brevi periodi di tempo le operazioni di autenticazione; proprio per questo sono stati creati i *Refresh token* che evitano di effettuare queste operazioni.

OAuth2.0

OAuth 2.0 [6] è un framework autorizzativo standard per applicazioni Web

- framework: architettura di supporto sul quale è possibile realizzare software che necessitano di coprire la fase di autorizzazione
- autorizzativo: processo di autorizzazione attraverso il quale vengono assegnati dei privilegi ai diversi utenti per l'utilizzo delle risorse
- standard: esistenza di una proposta di standardizzazione ETF
- applicazioni web: un qualunque applicazione realizzata con tecnologie Web

In giro per la rete spesso si trovano i diversi "Accedi con Facebook o Google", questo significa che è possibile accedere ad un'area riservata di un qualunque sito Web, senza eseguire una nuova registrazione, ma sfruttando le credenziali già utilizzate su una di quelle piattaforme.

Il sito web a cui vogliamo accedere tramite Facebook o Google, riceve un token dall'*OAuth 2.0 Server*; quando il sito Web chiama l'API per avere le informazioni, consegna il token ricevuto e così gli viene garantito l'accesso. La tecnologia *OAuth 2* non specifica come questo token debba essere verificato.

La scelta solitamente implementata consiste nel rilasciare un token con validità temporale limitata e autoreferenziale, cioè firmato con crittografia asimmetrica: un *JWT* (JSON Web Tokens). Utilizzando questo metodo, il sito Web ha una finestra temporale per accedere alle informazioni richieste e non può modificare i propri diritti di accesso perché non possiede la chiave privata per modificare il token.

Un token JWT è composto da tre sezioni:

- Header: formato JSON codificato con Base64
- Claims: formato JSON codificato con Base64
- Signature: creata e firmata basandosi sugli header e Claims codificato con Base64

Web Service RESTful

REST (REpresentational State Transfer) definisce un insieme di principi architetturali per la progettazione di un sistema e rappresenta uno stile architetturale per sistemi distribuiti. REST non rappresenta un protocollo poiché non descrive in alcun modo come i messaggi devono essere fatti, ma specifica i requisiti (o vincoli architetturali) che essi devono soddisfare.

REST negli ultimi anni ha preso sempre più piede come metodo per la realizzazione di Web Service altamente efficienti e scalabili.

I principi che deve rispettare un'architettura REST sono:

- client-server: il server deve offrire una serie di funzionalità e ascoltare le richieste fatte dal client; un client invoca un servizio messo a disposizione dal server attraverso un messaggio di richiesta e il servizio lato server respinge la richiesta o esegue la richiesta prima di inviare un messaggio di risposta al client
- stateless: la comunicazione tra client e server deve essere senza stato tra le richieste, il che significa che ogni richiesta da parte di un client deve contenere tutte le informazioni necessarie per consentire al servizio di rispondere in maniera adeguata alla richiesta; in sostanza è come se ogni richiesta fosse la prima e non correlata alle precedenti
- struttura a strati
- identificazione delle risorse: una risorsa è un oggetto o la rappresentazione di qualcosa di significativo nel dominio applicativo; ogni risorsa deve essere identificata univocamente. In ambito WEB, il meccanismo più naturale per individuare una risorsa è dato dal concetto di URI.

Una volta indicato come individuare una risorsa, è necessario disporre di un meccanismo per indicare le operazioni che possono essere effettuate su di essa; per fare questo è possibile fare riferimento ai metodi offerti dal protocollo HTTP, e cioè GET, POST, PUT e DELETE [7].

Metodo HTTP	Operazione CRUD	Descrizione
POST	Create	Crea una nuova risorsa
GET	Read	Ottiene una risorsa esistente
PUT	Update	Aggiorna una risorsa o ne modifica lo stato
DELETE	Delete	Elimina una risorsa

Applicazione

Come è stato impostato il progetto

Seguendo la definizione delle varie componenti di OAuth, abbiamo implementato i differenti attori che caratterizzano tale protocollo, adattandoli ad una applicazione di ASP.NET MVC. Client, Resource Server e Authorization Server vengono avviati sullo stesso host, tuttavia, ciascuno agirà in maniera completamente indipendente; questo è possibile soprattutto grazie alla impostazione che le applicazioni MVC hanno.

Client dell'applicazione

Questo componente è stato implementato attraverso una serie di pagine scritte in Razor, un macro-linguaggio che unisce HTML e C#. All'interno della Solution sono presenti tutte le viste che verranno predisposte per il client e che l'utente navigherà sono state inserite nella cartella *Views*. Il Client, per definizione e predisposizione delle applicazioni MVC, comunicherà sia con il Resource Server che con l'OAuth server. A quest'ultimo verrà richiesto un token di accesso che verrà utilizzato per ottenere le autorizzazioni necessarie per poter consentire l'accesso all'utente al catalogo e al proprio carrello.

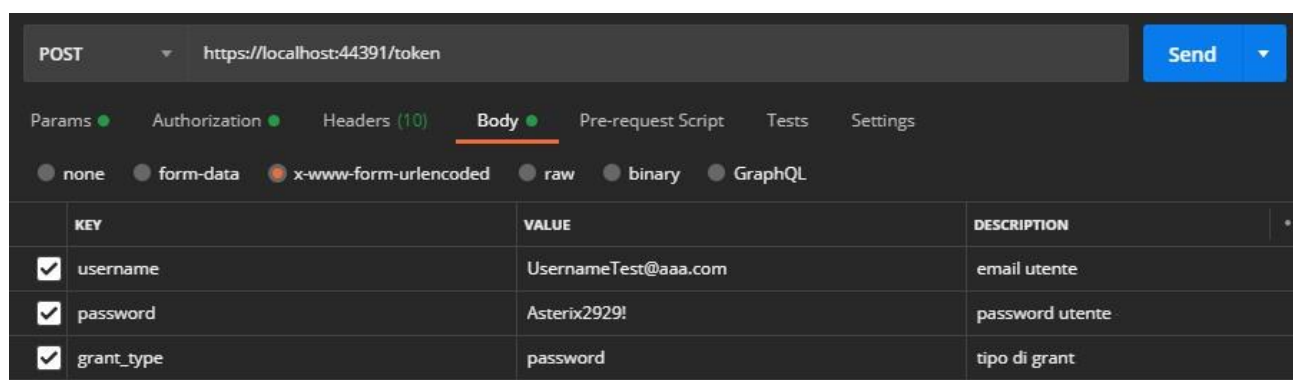
OAuth Server

Questo componente ha lo scopo di dover:

- Autenticare un utente che richiede l'accesso a determinate risorse. Nel nostro progetto, il carrello oppure il catalogo.
- Rilasciare un token JWT di autenticazione, contenente dei dati utilizzati successivamente dal resource server.

Per poter implementare questo attore presente in *OAuth*, abbiamo sviluppato due classi contenute nel folder nominato *OAuthServer*, che verranno di seguito spiegate nel dettaglio. Inoltre, per collegare tali classi all'applicazione, rendendo il tutto più sicuro, è stato utilizzato il middleware *Owin* di *Microsoft*. Tutte le volte che viene effettuata una chiamata *http Post* all'indirizzo "https://localhost:xxxx/token", includendo:

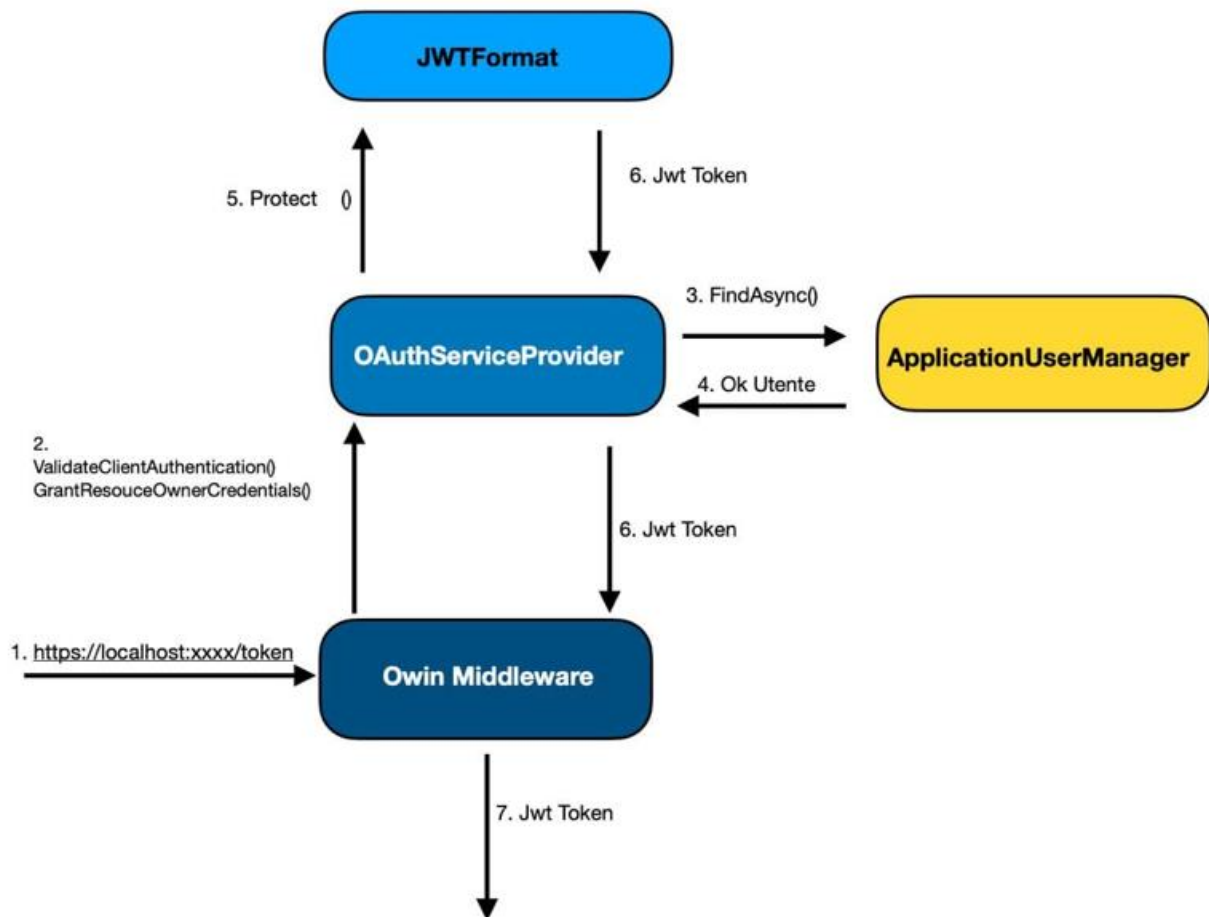
- grant_type impostato a "password";
- username contenente l'e-mail dell'utente che vuole effettuare il login;
- password contenente la password dell'utente che vuole effettuare il login.



The screenshot shows a REST client interface with a POST request to `https://localhost:44391/token`. The 'Body' tab is selected, and the 'x-www-form-urlencoded' format is chosen. The request body contains three parameters: 'username' with value 'UsernameTest@aaa.com', 'password' with value 'Asterix2929!', and 'grant_type' with value 'password'.

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> username	UsernameTest@aaa.com	email utente
<input checked="" type="checkbox"/> password	Asterix2929!	password utente
<input checked="" type="checkbox"/> grant_type	password	tipo di grant

e viene innescato il seguente meccanismo per il rilascio del token, mostrato nella figura seguente



Viene invocato il middleware *Owin* che, a sua volta, lancia la chiamata di due metodi (punto 2 della figura) presenti nella classe *OAuthServiceProviders* attraverso i quali vengono validati il client e l'utente. Per quanto riguarda il client, come si vedrà nel codice, esso viene sempre validato, mentre attraverso il metodo *GrantResourceOwnerCredentials* si invocherà il metodo della classe *ApplicationUserManager* per verificare la presenza dell'utente all'interno del database degli utenti tenuto dall'*OAuth*. Se la risposta dell'*ApplicationUserManager* risulta positiva, allora viene invocato il metodo *Protected* della classe *JWTFormat* che provvede a generare il token contenente una serie di informazioni utili per la verifica della validità del token. Il JWT token generato viene poi passato all'*Owin Middleware* che costruirà una risposta http alla post che inizialmente aveva innescato tutto il procedimento.

Resource Server

L'ultimo componente sviluppato si occupa della gestione lato back-end, nonché della gestione del rilascio dei dati richiesti dall'utente. Quello che lo compone sono una serie di classi che in un'applicazione MVC vengono etichettate come *Controller*. Una volta che il token viene rilasciato, successivamente ad una login dell'utente, vengono invocati una serie di controller che verificano la validità del token ricevuto dall'*OAuth Server*. Per verificarne l'effettiva validità si compiono due passaggi:

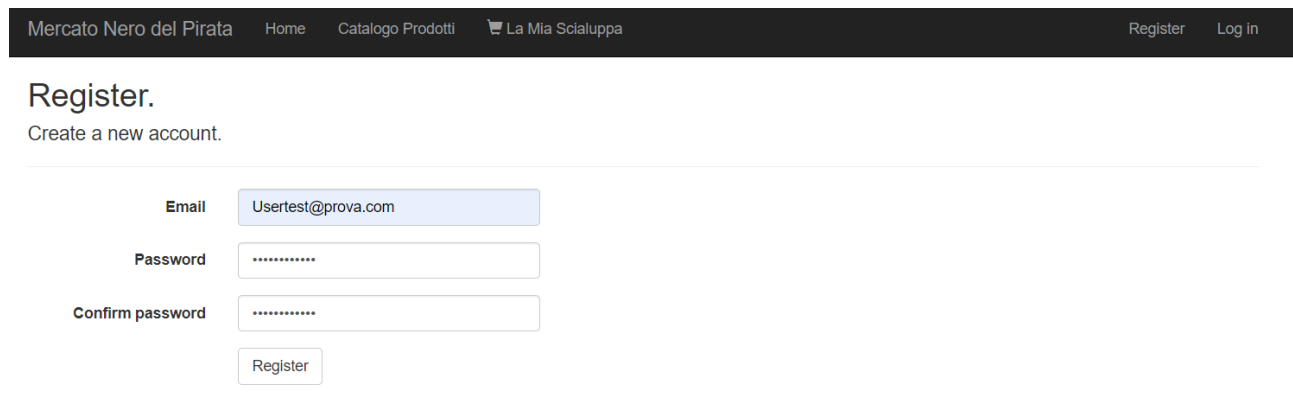
- Si verifica la firma del token;
- Si controlla che il token non sia scaduto.

Queste operazioni avvengono grazie alla classe *TokenManager* contenuta nel folder *ResourceServer*, che verrà spiegata in dettaglio durante la relazione.

Come si presenta l'applicazione

Registrazione utente

Prima di accedere all'applicazione, l'utente dovrà eseguire una registrazione nell'apposita sezione "Register".



The screenshot shows the 'Register' page of the application. At the top, there is a dark navigation bar with links: 'Mercato Nero del Pirata', 'Home', 'Catalogo Prodotti', and 'La Mia Scialuppa'. On the right side of the bar are links for 'Register' and 'Log in'. Below the navigation bar, the heading 'Register.' is followed by the text 'Create a new account.' The registration form consists of three input fields: 'Email' (containing 'Userstest@prova.com'), 'Password' (masked with dots), and 'Confirm password' (also masked with dots). A 'Register' button is located below the 'Confirm password' field.

© 2020 - Riccardo Longoni & Stefano Belloni ASP.NET Application

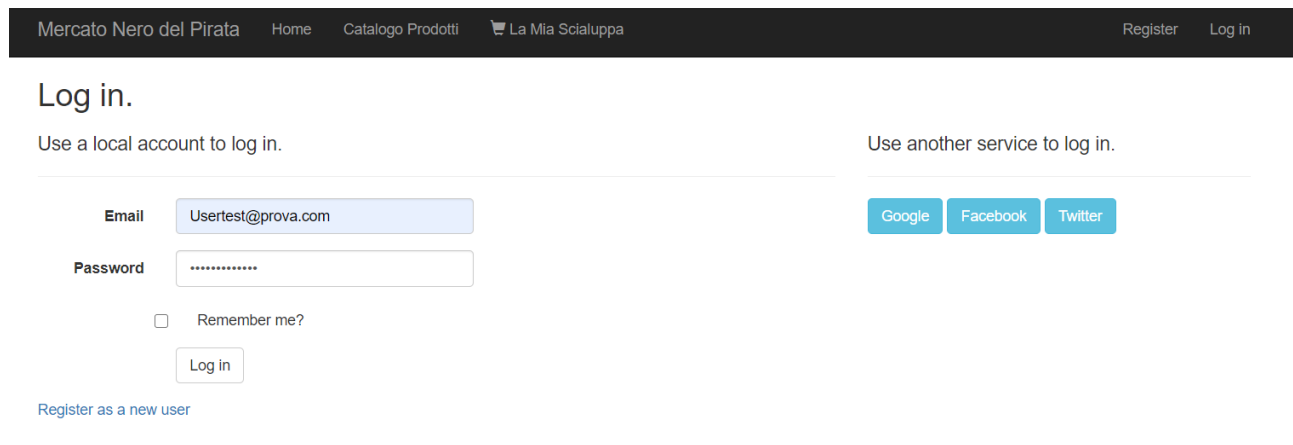
Per registrarsi, l'utente dovrà inserire una propria mail e una password; questi dati saranno poi memorizzati in una tabella presente in un database locale e, per una questione di sicurezza, la password non verrà salvata in chiaro, ma ne verrà memorizzato un suo hash. Una volta registrato, l'utente potrà andare nella sezione di login in cui potrà accedere all'applicazione.

Accesso dell'utente all'applicazione



The screenshot shows the home page of the application. The navigation bar is the same as in the previous screenshot, but the 'Log in' link is circled in red. Below the navigation bar, the heading 'Progetto Soasec' is displayed. Underneath, it says 'Questo è il progetto per il corso di Sicurezza delle Architetture Orientate ai Servizi' and 'Autori: Riccardo Longoni 943531 e Stefano Belloni 940842'. A blue button labeled 'Progetto su GitHub»' is located at the bottom of the main content area.

Nella fase di login, l'utente dovrà inserire la propria username e la password per potere accedere al catalogo ed eventualmente acquistare i prodotti.



The screenshot shows the 'Log in' page of the application. The navigation bar is the same as in the previous screenshots. Below the navigation bar, the heading 'Log in.' is followed by the text 'Use a local account to log in.' The login form consists of two input fields: 'Email' (containing 'Userstest@prova.com') and 'Password' (masked with dots). Below the 'Password' field is a checkbox labeled 'Remember me?'. A 'Log in' button is located below the 'Remember me?' checkbox. To the right of the local account login section, there is a section titled 'Use another service to log in.' with three buttons: 'Google', 'Facebook', and 'Twitter'. At the bottom left, there is a link 'Register as a new user'.

© 2020 - Riccardo Longoni & Stefano Belloni ASP.NET Application

L'utente, a sua scelta, potrà abilitare un sistema di doppia autenticazione aumentando così la sicurezza di accesso al proprio account; per farlo, oltre all'autenticazione con username e password, l'utente dovrà indicare il proprio numero di telefono sulla quale ad ogni accesso gli verrà inviato il codice da inserire. Questa opzione viene garantita dall'utilizzo del servizio messo a disposizione da *Twilio*.

Abilitazione doppia autenticazione

Mercato Nero del Pirata

Home

Catalogo Prodotti

La Mia Scialuppa

Salve Pirata Usertest@prova.com!

Log off

Manage.

Change your account settings

Password:

[[Change your password](#)]

External Logins:

0 [[Manage](#)]

Phone Number:

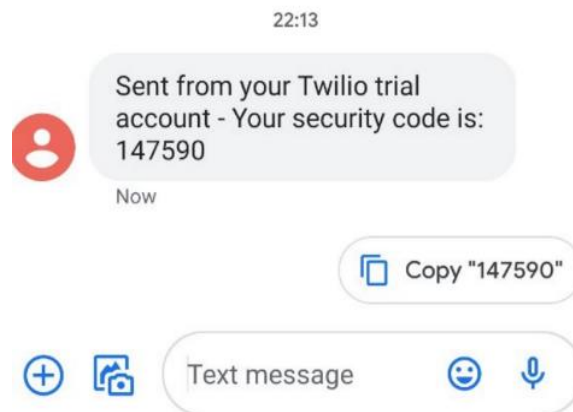
None [[Add](#)]

Two-Factor Authentication:

Disabled [Enable](#)

© 2020 - Riccardo Longoni & Stefano Belloni ASP.NET Application

Per abilitare la doppia autenticazione è necessario inserire il proprio numero di telefono; una volta inserito, verrà inviato sul proprio smartphone un messaggio contenente un codice da inserire per confermare il numero di telefono. Di seguito un esempio del messaggio:



L'utente poi procede con l'inserimento del codice

Mercato Nero del Pirata

Home

Catalogo Prodotti

La Mia Scialuppa

Salve Pirata Usertest@prova.com!

Log off

Verify Phone Number.

Enter verification code

Code

© 2020 - Riccardo Longoni & Stefano Belloni ASP.NET Application

Una volta inserito il codice, se corretto, appare il messaggio "Your phone number was added" che notifica che il numero di telefono è stato inserito correttamente nella propria area personale.

Manage.

Your phone number was added.

Change your account settings

Password: [[Change your password](#)]
External Logins: 0 [[Manage](#)]
Phone Number: +39 [REDACTED] [[Change](#) | [Remove](#)]
Two-Factor Authentication: Disabled [Enable](#)

© 2020 - Riccardo Longoni & Stefano Belloni ASP.NET Application

Visione e acquisto dei prodotti nel catalogo

L'utente, una volta autenticatosi, può sfogliare il catalogo dei prodotti ed eventualmente aggiungerli al carrello e procedere con il loro acquisto.

Nome del Prodotto: La Hechicera (Rum)



Categoria: Bevande

Descrizione: Un Rum che ti "Incanterà"

Quantità Disponibile: 2

Costo in Doblons: 10

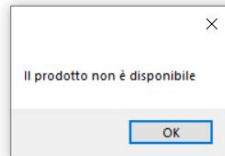
[Aggiungilo alla scialuppa!](#)

Di ogni prodotto presente all'interno del catalogo, vengono mostrate una serie di informazioni:

- nome del prodotto
- categoria di appartenenza
- una breve descrizione
- quantità disponibile
- costo in dobloni

Ovviamente se il prodotto è disponibile in 0 quantità, non è possibile aggiungerlo al carrello e viene fatto comparire un messaggio che notifica che il prodotto non è disponibile.

Nome del Prodotto: Capitan Flint



Categoria: Animali

Descrizione: Il Leggendaro pappagallo appartenuto al temibile pirata Long Jhon Silver!

Quantità Disponibile: 0

Costo in Doblioni: 100

 Aggiungilo alla scialuppa!

Se invece il prodotto risulta disponibile, viene aggiunto al carrello e la sua disponibilità viene decrementata in base alle quantità che sono state inserite nel carrello. In alto, nella barra, è possibile vedere il numero dei prodotti che sono presenti al momento nel carrello.

Sciabola
Arrugginita

Armi

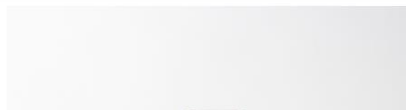


Arma arrugginita dall'acqua dei mari. 2
Danni 5-6

 Rimuovi

La Hechicera
(Rum)

Bevande



Un Rum che ti "Incanterà" 10

 Rimuovi

Se per errore viene inserito un prodotto non voluto nel carrello, è possibile rimuoverlo premendo sull'apposito pulsante "Rimuovi".

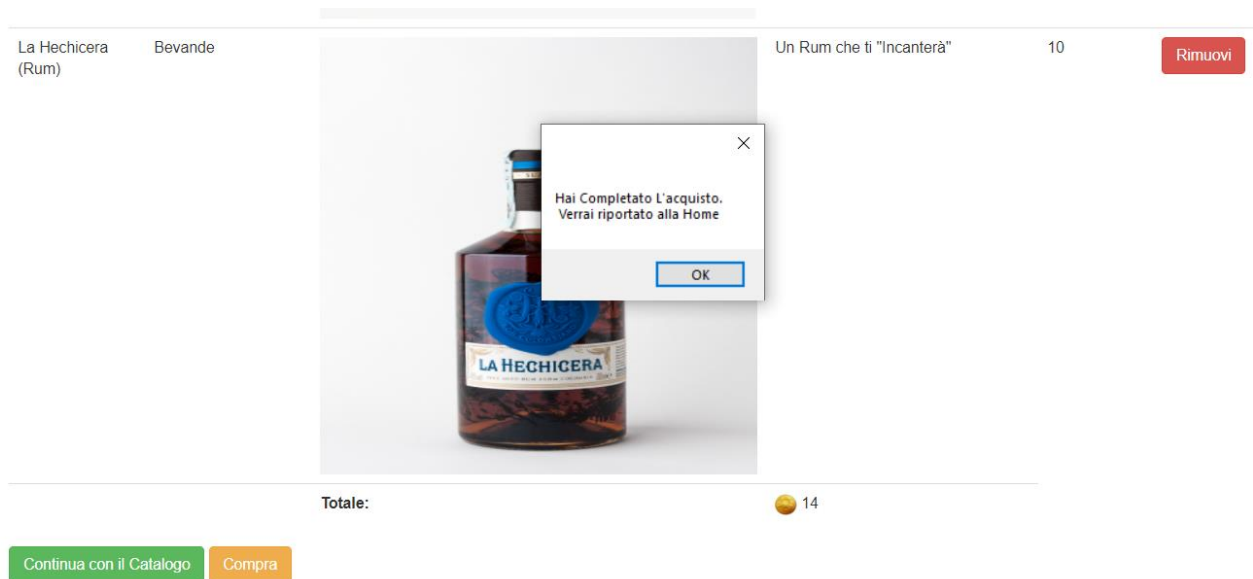
Quando l'utente è nella sezione "La Mia Scialuppa" può effettuare due scelte:

[Continua con il Catalogo](#)

[Compra](#)

- premere su "Continua con il Catalogo" e andare nel catalogo per continuare a visionare i prodotti
- premere su "Compra" e procedere con l'acquisto dei prodotti presenti nel carrello

Se l'opzione cliccata è "Compra", viene visualizzato il messaggio che indica che è stato fatto l'acquisto e si verrà reindirizzati alla Homepage.



Cambio password dell'utenza

L'utente dalla propria area personale può cambiare la password della propria utenza.

[Mercato Nero del Pirata](#) [Home](#) [Catalogo Prodotti](#) [La Mia Scialuppa](#) [Salve Pirata Usertest@prova.com!](#) [Log off](#)

Change Password.

Change Password Form

Current password

New password

Confirm new password

Change password

L'opzione del cambio prevede l'inserimento di quella attuale e quella nuova; una volta modificata, viene visualizzato a schermo il messaggio "Your password has been changed" che notifica che la password è stata cambiata.

[Mercato Nero del Pirata](#) [Home](#) [Catalogo Prodotti](#) [La Mia Scialuppa](#) [Salve Pirata Usertest@prova.com!](#) [Log off](#)

Manage.

Your password has been changed.

Change your account settings

Password:

[\[Change your password \]](#)

External Logins:

0 [\[Manage \]](#)

Phone Number:

+39 [REDACTED] [\[Change \]](#) [\[Remove \]](#)

Two-Factor Authentication:

Enabled [Disable](#)

Abilitazione/disabilitazione doppia autenticazione

L'utente dalla propria area personale può in qualsiasi momento abilitare/disabilitare la doppia autenticazione con il numero di telefono

Manage.

Change your account settings

Password: [[Change your password](#)]
External Logins: 0 [[Manage](#)]
Phone Number: +39 [REDACTED] [[Change](#) | [Remove](#)]
Two-Factor Authentication: Enabled [Disable](#)

Doppia autenticazione abilitata

Manage.

Change your account settings

Password: [[Change your password](#)]
External Logins: 0 [[Manage](#)]
Phone Number: +39 [REDACTED] [[Change](#) | [Remove](#)]
Two-Factor Authentication: Disabled [Enable](#)

Doppia autenticazione disabilitata

Dall'applicazione al codice

In questa sezione viene illustrato brevemente quali sono le parti di codice che gestiscono le varie funzioni

- La parte riguardante la gestione della registrazione dell'utente (sezione *Register* dell'applicazione) viene gestita attraverso il metodo *Register()* della classe *AccountController*
- La parte riguardante la gestione della Login dell'utente (sezione *Login* dell'applicazione) viene gestita attraverso il metodo *Login()* della classe *AccountController*
- La parte della gestione del numero di telefono e della password viene gestita nella classe *ManageController*
- La verifica della validità del token e la parte della visione del catalogo viene gestita nel metodo *Catalogo()* della classe *HomeController*
- La gestione dell'aggiunta e della rimozione dei prodotti viene gestita rispettivamente dai metodi *AggiungiProdotto()* e *RimuoviProdotto()* della classe *ScialuppaController*
- La conclusione dell'acquisto viene gestita dal metodo *PiazzaOrdine()* della classe *ScialuppaController*

Codice dell'applicazione

Folder Appstart

Il folder Appstart contiene all'interno le seguenti classi:

- BundleConfig
- FilterConfig
- IdentityConfig
- RouteConfig
- Startup.Auth
- WebApiConfig

Classe Startup.Auth

All'interno della classe *Startup.Auth* sono presenti dei metodi che consentono di impostare OAuth come framework di autorizzazione.

In questa classe vengono definiti i seguenti metodi

- *ConfigureAuth*
- *ConfigureOAuthTokenGeneration*

Metodo ConfigureAuth

Il metodo `ConfigureAuth()` viene utilizzato nella nostra classe invocare il metodo `ConfigureOAuthTokenGeneration()` usato per la gestione del rilascio del token e spiegato appena dopo.

Metodo ConfigureOAuthTokenGeneration

Nel metodo `ConfigureOAuthTokenGeneration()` vengono specificate le opzioni che OAuth dovrà avere per la gestione del rilascio del token.

Si passa come parametro un oggetto di tipo `IApplicationBuilder`, tramite il quale verranno richiamati dei metodi per generare dei *contesti* utilizzati dal middleware *Owin*.

Successivamente all'invocazione di tali metodi, si prosegue con la dichiarazione di una variabile di tipo `OAuthAuthorizationServerOptions` che conterrà una serie parametri per il funzionamento dell'OAuth server con il rilascio di token di tipo *JWT*.

```
private void ConfigureOAuthTokenGeneration(IApplicationBuilder app)
{
    // Configure the db context and user manager to use a single instance per request
    app.CreatePerOwinContext(ApplicationDbContext.Create);
    app.CreatePerOwinContext<ApplicationUserManager>(ApplicationUserManager.Create);

    OAuthAuthorizationServerOptions OAuthServerOptions = new OAuthAuthorizationServerOptions()
    {
        AllowInsecureHttp = true, // da usare solo in fase di sviluppo
        TokenEndpointPath = new PathString("/token"),
        AccessTokenExpireTimeSpan = TimeSpan.FromMinutes(3),
        Provider = new OAuthServiceProvider(),
        AccessTokenFormat = new JWTFormat("OAuthServer"),
    };

    app.UseOAuthAuthorizationServer(OAuthServerOptions);
}
```

Tali parametri sono commentati di seguito:

- **AllowInsecureHttp**: consente l'utilizzo di Http, oltre ad Https che viene utilizzato di default. Questa opzione, per questione di sicurezza, è da considerare solamente durante la fase di sviluppo, non durante il rilascio effettivo dell'intera applicazione.
- **TokenEndpointPath**: definisce l'URL che deve essere invocato per far avviare la procedura di rilascio del token JWT da OAuthServer.
- **Provider**: nel nostro progetto è stato utilizzato un `OAuthServiceProvider` ossia una classe presente della quale è stata personalizzata la logica del rilascio dei token JWT e dell'autenticazione del client, attraverso la quale si effettua la richiesta di rilascio del token e l'autenticazione dell'utente richiedente le risorse.
- **AccessTokenFormat**: Attraverso questo parametro, come specificato dallo stesso nome, si indica il tipo di token che si vuole rilasciare. Anche in questo caso, il tipo di formato e la sua logica è stata implementata a parte, mantenendo il tipo di formato JWT.

Successivamente, con il comando `app.UseOAuthAuthorizationServer` si va a indicare a tutta l'applicazione che tipo di Server di Autorizzazione verrà utilizzato.

Il metodo `ConfigureOAuthTokenGeneration` è invocato dal metodo `ConfigureAuth`, che a sua volta verrà invocato in fase di lancio dell'applicazione.

Classe IdentityConfig

Nel momento in cui è stata creata l'intera soluzione di ASP.NET sono state generate delle classi con dei metodi già funzionanti, ma non funzionali al nostro obiettivo di sviluppo.

SendAsync(IdentityMessage) è stato uno di questi e, su di esso, abbiamo lavorato al fine di rendere possibile l'autenticazione a due fattori attraverso il numero di telefono.

Metodo SendAsync

Il metodo SendAsync() viene utilizzato per la gestione del messaggio della fase di doppia autenticazione.

```
public Task SendAsync(IdentityMessage message)
{
    var accountSid = ConfigurationManager.AppSettings["SID_SMS_Account"];
    var authToken = ConfigurationManager.AppSettings["Token_SMS_Account"];
    var fromNumber = ConfigurationManager.AppSettings["From_SMS_Account"];
    TwilioClient.Init(accountSid, authToken);

    MessageResource result = MessageResource.Create(
        new PhoneNumber(message.Destination),
        from: new PhoneNumber(fromNumber),
        body: message.Body
    );

    Trace.TraceInformation(result.Status.ToString());
    return Task.FromResult(0);
}
```

Nella prima parte vengono impostati i parametri rilasciati da *Twilio* ovvero un identificativo, uno username e il numero di telefono dal quale vengono poi inviati i messaggi contenenti il codice utilizzato per la doppia autenticazione. Tali valori vengono presi da un file esterno *Web.config* attraverso la classe *ConfigurationManager*.

Il metodo *Twilio.Init()* viene utilizzato per inizializzare un client *Twilio* prendendo come parametri uno username, che corrisponde all'id rilasciato da *Twilio* e, una password, in questo caso un token rilasciato da *Twilio*.

Successivamente si istanzia una variabile di tipo *MessageResource* il cui contenuto è il risultato della chiamata del metodo *MessageResource.Create()* attraverso la quale vengono settati i parametri per l'autenticazione a due fattori; viene impostato il numero di telefono su cui verrà ricevuto il messaggio, il numero di telefono dal quale verrà inviato il messaggio e il corpo del messaggio contenente il codice di verifica.

Il metodo *Trace.TraceInformation()* scrive un messaggio informativo nei listener usando il messaggio specificato.

L'ultimo *return* indica che il tutto è avvenuto correttamente.

Classe WebApiConfig

Metodo Register

Il metodo *Register()* viene utilizzato per impostare il tipo di percorso attraverso il quale verranno invocate le API dell'Authorization Server e del Resource Server.

```

public static void Register(HttpConfiguration config)
{
    // Aggiunta per il server OAuth2
    config.SuppressDefaultHostAuthentication(); //aggiunta
    config.Filters.Add(new HostAuthenticationFilter(DefaultAuthenticationTypes.ExternalBearer)); // aggiunta
    var cors = new EnableCorsAttribute("*", "*", "*");
    config.EnableCors(cors);
    // Web API configuration and services

    // Web API routes
    config.MapHttpAttributeRoutes();

    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );
}

```

Il metodo `config.MapHttpAttributeRoutes()` abilita il routing degli attributi che verranno indicati nelle righe di codice dopo. Il metodo `MapHttpRoute()` crea una nuova istanza di `IHttpRoute`; di seguito i parametri utilizzati:

- **name:** indica il nome della route
- **routeTemplate:** indica il pattern URL della route
- **defaults:** un parametro oggetto che include valore di route predefiniti

Folder Controllers

Classe AccountController

I metodi principali della classe `AccountController` sono:

- Login
- Logout
- Register
- Dispose

Metodo Register

Il metodo `Register()` viene utilizzato per registrare un nuovo utente all'applicazione.

```

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
0 references
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            HttpClient client = new HttpClient();
            HttpResponseMessage resultMessage = await client.PostAsync($"https://localhost:44391/token",
                new StringContent(string.Format("grant_type=password&username={0}&password={1}", model.Email, model.Password), Encoding.UTF8));
            string resultContent = resultMessage.Content.ReadAsStringAsync().Result;
            var token = JsonConvert.DeserializeObject<BearerToken>(resultContent).AccessToken;

            var myConfiguration = WebConfigurationManager.OpenWebConfiguration("~");
            myConfiguration.AppSettings.Settings["token"].Value = token;
            myConfiguration.AppSettings.Settings["LoginEmail"].Value = model.Email;
            myConfiguration.AppSettings.Settings["LoginPassword"].Value = model.Password;
            myConfiguration.Save();

            await SignInManager.SignInAsync(user, isPersistent:false, rememberBrowser:false);

            return RedirectToAction("Index", "Home");
        }
        AddErrors(result);
    }
}

```

Per prima cosa viene fatto un controllo per verificare se il modello della View, denominata *Login*, passato è valido; in caso contrario viene rilasciato un errore con la relativa specifica.

Qualora risulti valido, vengono recuperati i parametri passati nella form (vedi `ApplicationUser`) e con il metodo `await UserManager.CreateAsync()` viene creato l'utente. Questo comporta la creazione di un nuovo record all'interno della tabella *AspNetUsers* del database SQL utilizzato, salvando come elementi la

mail indicata e l'hash della password inserita nella forma. Se la creazione del nuovo utente ha avuto successo (invocando *result.Succeeded*), quest'ultimo viene fatto accedere all'applicazione (con il metodo *SignInAsync*).

Trattandosi di una login, è necessario rilasciare un *token JWT* per l'utente appena registrato, pertanto viene eseguita una chiamata *POST* all'OAuth Server. Nella chiamata vengono passati i dati che sono stati inseriti nella form di registrazione, come la Email e Password. L'OAuth Server, una volta autenticato l'utente appena registrato, risponderà con un *HttpResponseMessage* dentro al quale sarà presente il *token JWT*. Il token e i dati inseriti dall'utente in fase di registrazione, e di login, vengono archiviati nel file *Web.config* e verranno utilizzati dal Resource server per consentire l'accesso alle risorse.

Il metodo *SignInManager.SignInAsync()* genera un *ClaimsIdentity*. Poiché l'autenticazione dei cookie *ASP.NET Identity* e *OWIN* è un sistema basato su attestazioni, il Framework richiede che l'app generi un *ClaimsIdentity* per l'utente. *ClaimsIdentity* contiene informazioni riguardanti le attestazioni per l'utente, ad esempio i ruoli a cui appartiene l'utente. Il metodo *SignInAsync()* utilizzato prende in input tre parametri: il primo parametro rappresenta l'utente che si deve loggare, il secondo che indica se il cookie di accesso deve persistere dopo la chiusura del browser e il terzo indica che non è flaggato il *Rememberme* nella pagina di Login.

Metodo Login

Il metodo *Login()* non è altro che un metodo invocabile attraverso una chiamata *Https POST* utilizzato per eseguire l'accesso all'applicazione.

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
0 references
public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }

    HttpClient client = new HttpClient();
    HttpResponseMessage resultMessage = await client.PostAsync($"https://localhost:44391/token",
        new StringContent(string.Format("grant_type=password&username={0}&password={1}", model.Email, model.Password), Encoding.UTF8));
    string resultContent = resultMessage.Content.ReadAsStringAsync().Result;
    var token = JsonConvert.DeserializeObject<BearerToken>(resultContent).AccessToken;

    var myConfiguration = WebConfigurationManager.OpenWebConfiguration("~");
    myConfiguration.AppSettings.Settings["token"].Value = token;
    myConfiguration.AppSettings.Settings["LoginEmail"].Value = model.Email;
    myConfiguration.AppSettings.Settings["LoginPassword"].Value = model.Password;
    myConfiguration.Save();

    var result = await SignInManager.PasswordSignInAsync(model.Email, model.Password, model.RememberMe, shouldLockout: false);
    switch (result)
    {
        case SignInStatus.Success:
            return RedirectToLocal(returnUrl);
        case SignInStatus.LockedOut:
            return View("Lockout");
        case SignInStatus.RequiresVerification:
            return RedirectToAction("SendCode", new { ReturnUrl = returnUrl, RememberMe = model.RememberMe });
        case SignInStatus.Failure:
        default:
            ModelState.AddModelError("", "Invalid login attempt.");
            return View(model);
    }
}
```

Viene utilizzata la classe *HttpClient* per consentire di eseguire la *POST* all'indirizzo <https://localhost:44391/token> (da notare l'utilizzo di *https* che garantisce una maggiore sicurezza).

Il metodo *await client.PostAsync()* prende come parametri l'uri a cui viene inviata la richiesta e il contenuto della richiesta *http* inviata al server; trattasi di un metodo asincrono (come indicato dalla parola chiave *async*), mentre la parola chiave *await* sospende l'esecuzione di una funzione in attesa che una data attività venga eseguita. Utilizzato poi il metodo *resultMessage.Content.ReadAsStringAsync().Result* per recuperare informazioni da *expiredatas* ecc..

Con il metodo *JsonConvert.DeserializeObject<BearerToken>(resultContent).AccessToken* andiamo a estrarre l'*AccessToken* dell'oggetto di tipo *BearerToken* creato da noi (vedi nella classe *Models*).

Proseguendo con il codice, ci sono una serie di comandi che vengono utilizzati per cambiare/aggiornare i parametri presenti nella classe *WebConfig* con quelli attuali; i dati aggiornati saranno utilizzati per la gestione del token.

Infine, come ultima parte, vi è la gestione dei controlli che consentono, a seconda dei parametri inseriti nella form, la gestione della *login* e del *Remember me*.

Metodo Logoff

Il metodo `LogOff()` consente la gestione del Log out (uscita) dall'applicazione.

```
public ActionResult LogOff()
{
    AuthenticationManager.SignOut(DefaultAuthenticationTypes.ApplicationCookie);
    var myConfiguration = WebConfigurationManager.OpenWebConfiguration("~");
    myConfiguration.AppSettings.Settings["token"].Value = "NoToken";
    myConfiguration.AppSettings.Settings["LoginEmail"].Value = "NoUsername";
    myConfiguration.AppSettings.Settings["LoginPassword"].Value = "NoPassword";
    myConfiguration.AppSettings.Settings.Remove("SID_SMS_Account");
    myConfiguration.AppSettings.Settings.Remove("Token_SMS_Account");
    myConfiguration.AppSettings.Settings.Remove("From_SMS_Account");
    myConfiguration.AppSettings.Settings.Remove("API_Twitter_Key");
    myConfiguration.AppSettings.Settings.Remove("API_Twitter_Secret_Key");
    myConfiguration.AppSettings.Settings.Remove("Bearer_Twitter-Token");
    myConfiguration.AppSettings.Settings.Remove("Google_ClientID");
    myConfiguration.AppSettings.Settings.Remove("Google_ClientSecret");
    myConfiguration.AppSettings.Settings.Remove("Facebook_ClientID");
    myConfiguration.AppSettings.Settings.Remove("Facebook_ClientSecret");
    myConfiguration.AppSettings.Settings.Remove("Microsoft_ClientID");
    myConfiguration.AppSettings.Settings.Remove("Microsoft_SecretID");
    myConfiguration.AppSettings.Settings.Remove("SecretKey");
    myConfiguration.AppSettings.Settings.Remove("RefreshSecretKey");
    myConfiguration.AppSettings.Settings.Remove("AudienceID");

    myConfiguration.Save();
    return RedirectToAction("Index", "Home");
}
```

Il metodo `Logoff()` è un metodo invocabile attraverso una chiamata *Https Post*.

Il comando `AuthenticationManager.SignOut(DefaultAuthenticationTypes.ApplicationCookie)` viene utilizzato per la rimozione dei cookie.

Il resto del codice viene utilizzato per l'aggiornamento dei parametri nel file *Web.config*, dove per questioni legate alla sicurezza dei parametri *Token*, *LoginEmail* e *LoginPassword* vengono assegnati rispettivamente i valori *NoToken*, *NoUsername* e *NoPassword*.

L'ultima cosa che viene impostata è il re-indirizzamento verso la Homepage in cui viene riproposta la pagina di Login.

Classe HomeController

Metodo Catalogo

Il metodo Catalogo() consente la gestione del catalogo dei prodotti da visualizzare.

```
public async Task<ActionResult> Catalogo()
{
    bool emailFlag = ConfigurationManager.AppSettings["LoginEmail"].Equals("NoUsername");
    if (emailFlag || OAuthSoasec.IsAuthorized().Equals("Nuova Login"))
    {
        return RedirectToAction("Login", "Account");
    }
    else {
        if (OAuthSoasec.IsAuthorized().Equals("Nuova Token"))
        {
            using (HttpClient client = new HttpClient())
            {
                HttpResponseMessage resultMessage = await client.PostAsync($"https://localhost:44391/token", new StringContent(string.Format("grant_type=password&username={0}&password={1}",
                    ConfigurationManager.AppSettings["LoginEmail"], ConfigurationManager.AppSettings["LoginPassword"]), Encoding.UTF8));
                string newToken = resultMessage.Content.ReadAsStringAsync().Result;
                var myConfiguration = WebConfigurationManager.OpenWebConfiguration("~");
                myConfiguration.AppSettings.Settings["token"].Value = newToken;
                myConfiguration.Save();
            }
        }
    }
}
```

```
string selectCatalogo = "select * from [dbo].[Catalogo]";
TabellaCatalogo = new DataTable();
TabellaCatalogo = speaker.SelectAll(selectCatalogo);
List<ProdottoModel> listaProdotti = new List<ProdottoModel>();
for (int i = 0; i < TabellaCatalogo.Rows.Count; i++)
{
    ProdottoModel prodotto = new ProdottoModel();
    prodotto.Id = Convert.ToInt32(TabellaCatalogo.Rows[i]["Id"]);
    prodotto.NomeProdotto = TabellaCatalogo.Rows[i]["Nome Prodotto"].ToString();
    prodotto.CategoriaProdotto = TabellaCatalogo.Rows[i]["Categoria Prodotto"].ToString();
    prodotto.Descrizione = TabellaCatalogo.Rows[i]["Descrizione"].ToString();
    prodotto.QuantitaDisponibile = Convert.ToInt32(TabellaCatalogo.Rows[i]["Quantità Disponibile"]);
    prodotto.CostoInDoblioni = Convert.ToInt32(TabellaCatalogo.Rows[i]["Costo in Doblioni"]);
    prodotto.Immagine = TabellaCatalogo.Rows[i]["Immagine URL"].ToString();
    listaProdotti.Add(prodotto);
}
return View(listaProdotti);
}
```

Essendo un metodo utilizzato dal Resource Server, è importante andare a verificare la validità del token, prima di rilasciare e visualizzare le informazioni del catalogo. Il controllo viene fatto attraverso il costrutto if-then-else in cui la verifica avviene su due condizioni poste in OR; nella prima viene verificato se l'utente è loggato, mentre nella seconda viene verificato attraverso il metodo `isAuthorized` della classe `Authorized` (spiegata in seguito) che il *Access token* e il *Refresh token* non siano scaduti. Se l'utente non risulta loggato oppure i token sono scaduti, lo user viene indirizzato nella schermata di Login. Qualora entrambe le condizioni risultino false, si entra nell'else e viene verificato se l'*Access Token* sia ancora valido (il *Refresh Token* è ancora valido); se non lo fosse, viene inviato tramite una chiamata POST all'OAuth server assieme al *RefreshToken* per la generazione di un nuovo Access Token. Il nuovo Access Token poi verrà aggiornato nel file *Web.config*.

La seconda parte del metodo riguarda la gestione del catalogo in cui vengono mostrati i prodotti, i relativi prezzi, una breve descrizione di ognuno e la quantità disponibile.

Classe ManageController

La classe `ManageController` è un'estensione della classe `Controller` che fornisce metodi che rispondono alle richieste HTTP effettuate a un sito *Web ASP.NET MVC*.

```
[RequireHttps]
[Authorize]
2 references
public class ManageController : Controller
{
    private ApplicationSignInManager _signInManager;
    private ApplicationUserManager _userManager;
```

In questa classe vengono utilizzati l'attributo *RequireHttps* che forza il rinvio di una richiesta HTTP non protetta tramite HTTPS e l'attributo *Authorize* che viene utilizzato per limitare l'accesso agli utenti autenticati correttamente.

Metodo Index

Il metodo `Index()` viene utilizzato per la gestione dei messaggi nel momento in cui vengono svolte operazioni come cambio delle password, inserimento/rimozione numeri di telefono e autenticazione a due fattori.

```
public async Task<ActionResult> Index(ManageMessageId? message)
{
    ViewBag.StatusMessage =
        message == ManageMessageId.ChangePasswordSuccess ? "Your password has been changed."
        : message == ManageMessageId.SetPasswordSuccess ? "Your password has been set."
        : message == ManageMessageId.SetTwoFactorSuccess ? "Your two-factor authentication provider has been set."
        : message == ManageMessageId.Error ? "An error has occurred."
        : message == ManageMessageId.AddPhoneSuccess ? "Your phone number was added."
        : message == ManageMessageId.RemovePhoneSuccess ? "Your phone number was removed."
        : "";

    var userId = User.Identity.GetUserId();
    var model = new IndexViewModel
    {
        HasPassword = HasPassword(),
        PhoneNumber = await UserManager.GetPhoneNumberAsync(userId),
        TwoFactor = await UserManager.GetTwoFactorEnabledAsync(userId),
        Logins = await UserManager.GetLoginsAsync(userId),
        BrowserRemembered = await AuthenticationManager.TwoFactorBrowserRememberedAsync(userId)
    };
    return View(model);
}
```

ViewBag è un contenitore/dizionario che contiene e consente di condividere dinamicamente i valori con le *View* dal controller; vi sono una serie di variabili che non sono note prima dell'esecuzione e i valori vengono immessi solo durante il runtime.

Il metodo `User.Identity.GetUserId()` consente di ritornare l'utente che risulta in quel momento loggato. L'oggetto `IndexViewModel` permette la gestione dei dati dell'utente.

Metodo AddPhoneNumber

Il metodo `AddPhoneNumber()` viene utilizzato per l'aggiunta del numero di telefono nell'area personale dell'utente.

```
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<ActionResult> AddPhoneNumber(AddPhoneNumberViewModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }
    // Generate the token and send it
    var code = await UserManager.GenerateChangePhoneNumberTokenAsync(User.Identity.GetUserId(), model.Number);
    if (UserManager.SmsService != null)
    {
        var message = new IdentityMessage
        {
            Destination = model.Number,
            Body = "Your security code is: " + code
        };
        await UserManager.SmsService.SendAsync(message);
    }
    return RedirectToAction("VerifyPhoneNumber", new { PhoneNumber = model.Number });
}
```

Il metodo `UserManager.GenerateChangePhoneNumberTokenAsync()` viene utilizzato per generare un codice che l'utente deve inserire per confermare il numero di telefono da lui indicato; il metodo prende due parametri che corrispondono allo user ID e al numero di telefono.

Viene poi eseguito un controllo su `userManager.SmsService` e verificato che sia diverso da `null`; tale metodo viene utilizzato per inviare messaggi SMS. Se la condizione viene verificata, viene settato come destinatario il numero di telefono passato come parametro e il corpo del messaggio inviato sarà "Your security code is" + il codice creato attraverso il metodo

`userManager.GenerateChangePhoneNumberTokenAsync()` invocato in precedenza.

Il metodo `await userManager.SmsService.SendAsync()` viene utilizzato per l'invio del messaggio con il codice.

Metodo EnableTwoFactorAuthentication

Il metodo `EnableTwoFactorAuthentication()` consente di abilitare l'accesso all'applicazione attraverso la doppia autenticazione, ovvero l'autenticazione con la verifica del numero di telefono, oltre che alla normale verifica di username e password.

```
[HttpPost, ValidateAntiForgeryToken]
0 references
public async Task<ActionResult> EnableTwoFactorAuthentication()
{
    await userManager.SetTwoFactorEnabledAsync(user.Identity.GetUserId(), true);
    var user = await userManager.FindByIdAsync(user.Identity.GetUserId());
    if (user != null)
    {
        await signInManager.SignInAsync(user, isPersistent: false, rememberBrowser: false);
    }
    return RedirectToAction("Index", "Manage");
}
```

Il metodo `userManager.SetTwoFactorEnabledAsync()` viene utilizzato per impostare la doppia autenticazione; come parametri prende l'user ID e un parametro booleano che se `true` abilita la doppia autenticazione, mentre se `false` non la rende attiva. Nel nostro caso è `true` perché vogliamo che sia presente la doppia autenticazione.

Il metodo `userManager.FindByIdAsync()` trova e restituisce l'utente, se presente, che corrisponde allo user ID specificato come parametro. trattasi di un metodo asincrono (come indicato dalla parola chiave `async`), mentre la parola chiave `await` sospende l'esecuzione di una funzione in attesa che una data attività venga eseguita. Se l'utente è stato trovato, quindi il metodo invocato non restituisce `null`, viene invocato il metodo `signInManager.SignInAsync()` utilizzato per far accedere l'utente e prende come parametri l'utente da loggare, il flag che indica se il cookie di accesso deve persistere dopo la chiusura del browser e il nome del metodo utilizzato per far autenticare l'utente.

Metodo DisableTwoFactorAuthentication

Il metodo `DisableTwoFactorAuthentication()` consente di disabilitare la doppia autenticazione nell'applicazione.

```
public async Task<ActionResult> DisableTwoFactorAuthentication()
{
    await userManager.SetTwoFactorEnabledAsync(user.Identity.GetUserId(), false);
    var user = await userManager.FindByIdAsync(user.Identity.GetUserId());
    if (user != null)
    {
        await signInManager.SignInAsync(user, isPersistent: false, rememberBrowser: false);
    }
    return RedirectToAction("Index", "Manage");
}
```

Il metodo `userManager.SetTwoFactorEnabledAsync()` viene utilizzato per impostare la doppia autenticazione; come parametri prende l'user ID e un parametro booleano che se `true` abilita la doppia autenticazione, mentre se `false` la disabilita. Nel nostro caso è `false` perché non vogliamo considerarla.

Il metodo `UserManager.FindByIdAsync()` trova e restituisce l'utente, se presente, che corrisponde allo user ID specificato come parametro. trattasi di un metodo asincrono (come indicato dalla parola chiave *async*), mentre la parola chiave *await* sospende l'esecuzione di una funzione in attesa che una data attività venga eseguita. Se l'utente è stato trovato, quindi il metodo invocato non restituisce *null*, viene invocato il metodo `SignInManager.SignInAsync()` utilizzato per far accedere l'utente e prende come parametri l'utente da loggare, il flag che indica se il cookie di accesso deve persistere dopo la chiusura del browser e il nome del metodo utilizzato per far autenticare l'utente.

Metodo VerifyPhoneNumber

Il metodo `VerifyPhoneNumber()` viene utilizzato per verificare il numero di telefono e ne troviamo presenti in due versioni con firme diverse.

```
public async Task<ActionResult> VerifyPhoneNumber(string phoneNumber)
{
    var code = await UserManager.GenerateChangePhoneNumberTokenAsync(User.Identity.GetUserId(), phoneNumber);
    // Send an SMS through the SMS provider to verify the phone number
    return phoneNumber == null ? View("Error") : View(new VerifyPhoneNumberViewModel { PhoneNumber = phoneNumber });
}
```

In questa versione viene passato come parametro una stringa contenente il numero di telefono e viene utilizzato il metodo `UserManager.GenerateChangePhoneNumberTokenAsync()` che prende come parametri l'user ID e il numero di telefono per generare il codice da inviare via SMS.

```
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<ActionResult> VerifyPhoneNumber(VerifyPhoneNumberViewModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }
    var result = await UserManager.ChangePhoneNumberAsync(User.Identity.GetUserId(), model.PhoneNumber, model.Code);
    if (result.Succeeded)
    {
        var user = await UserManager.FindByIdAsync(User.Identity.GetUserId());
        if (user != null)
        {
            await SignInManager.SignInAsync(user, isPersistent: false, rememberBrowser: false);
        }
        return RedirectToAction("Index", new { Message = ManageMessageId.AddPhoneSuccess });
    }
    // If we got this far, something failed, redisplay form
    ModelState.AddModelError("", "Failed to verify phone");
    return View(model);
}
```

In questa versione del metodo, l'obiettivo è la verifica del numero di telefono nel caso in cui ne avessi già uno impostato, ma si desidera cambiarlo. Al metodo generale viene passato un oggetto di tipo `VerifyPhoneNumberViewModel`.

Viene poi invocato il metodo `UserManager.ChangePhoneNumberAsync()` utilizzato per impostare il numero di telefono per uno specifico utente solo nel caso in cui il token di modifica specificato è valido; questo metodo prende in input tre parametri: il primo rappresenta l'utente per il quale deve essere impostato il numero di telefono, il secondo il numero di telefono da settare e il terzo il token di conferma del numero di telefono da validare.

Il metodo `UserManager.FindByIdAsync()` trova e restituisce l'utente, se presente, che corrisponde allo user ID specificato come parametro. trattasi di un metodo asincrono (come indicato dalla parola chiave *async*), mentre la parola chiave *await* sospende l'esecuzione di una funzione in attesa che una data attività venga eseguita. Se l'utente è stato trovato, quindi il metodo invocato non restituisce *null*, viene invocato il metodo `SignInManager.SignInAsync()` utilizzato per far accedere l'utente e prende come parametri l'utente da loggare, il flag che indica se il cookie di accesso deve persistere dopo la chiusura del browser e il nome del metodo utilizzato per far autenticare l'utente.

Attraverso il metodo `RedirectToAction()` viene fatto ritornare un messaggio che il numero di telefono è stato aggiunto correttamente; come primo parametro viene passato "Index" che corrisponde al metodo presente in questa classe con la quale vengono gestiti i vari messaggi di avvenuta modifica/inserimento, mentre come secondo parametro l'elemento `ManageMessageId.AddPhoneSuccess` che corrisponde al messaggio "Your phone number was added".

Metodo RemovePhoneNumber

Il metodo `RemovePhoneNumber()` consente di rimuovere il numero di telefono dal profilo di un utente.

```
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<ActionResult> RemovePhoneNumber()
{
    var result = await UserManager.SetPhoneNumberAsync(User.Identity.GetUserId(), null);
    if (!result.Succeeded)
    {
        return RedirectToAction("Index", new { Message = ManageMessageId.Error });
    }
    var user = await UserManager.FindByIdAsync(User.Identity.GetUserId());
    if (user != null)
    {
        await SignInManager.SignInAsync(user, isPersistent: false, rememberBrowser: false);
    }
    return RedirectToAction("Index", new { Message = ManageMessageId.RemovePhoneSuccess });
}
```

Questo metodo consente di rimuovere il numero di telefono dal profilo di un utente.

Il metodo `UserManager.SetPhoneNumberAsync()` viene utilizzato per la rimozione del numero di telefono dell'utente. Come parametri prendi in input l'utente per la quale si vuole rimuovere il numero di telefono e il numero di telefono da settare; considerato che non si vuole impostare nessun nuovo numero di telefono, ma se vuole semplicemente rimuoverlo, come valore gli passiamo *null*. Facciamo successivamente una verifica per capire se la modifica ha avuto successo; qualora il risultato fosse negativo, facciamo ritornare un messaggio che indica il tipo di errore che è avvenuto.

Il metodo `UserManager.FindByIdAsync()` trova e restituisce l'utente, se presente, che corrisponde allo user ID specificato come parametro. trattasi di un metodo asincrono (come indicato dalla parola chiave *async*), mentre la parola chiave *await* sospende l'esecuzione di una funzione in attesa che una data attività venga eseguita. Se l'utente è stato trovato, quindi il metodo invocato non restituisce *null*, viene invocato il metodo `SignInManager.SignInAsync()` utilizzato per far accedere l'utente e prende come parametri l'utente da loggare, il flag che indica se il cookie di accesso deve persistere dopo la chiusura del browser e il nome del metodo utilizzato per far autenticare l'utente.

Attraverso il metodo `RedirectToAction` viene fatto ritornare un messaggio che il numero di telefono è stato rimosso correttamente; come primo parametro viene passato "Index" che corrisponde al metodo presente in questa classe con la quale vengono gestiti i vari messaggi di avvenuta modifica/inserimento, mentre come secondo parametro l'elemento `ManageMessageId.RemovePhoneSuccess` che corrisponde al messaggio "Your phone number was removed".

Metodo ChangePassword

Il metodo `ChangePassword()` consente di cambiare la password all'utente all'interno dell'applicazione.

```

[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<ActionResult> ChangePassword(ChangePasswordViewModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }
    var result = await UserManager.ChangePasswordAsync(User.Identity.GetUserId(), model.OldPassword, model.NewPassword);
    if (result.Succeeded)
    {
        var user = await UserManager.FindByIdAsync(User.Identity.GetUserId());
        if (user != null)
        {
            await SignInManager.SignInAsync(user, isPersistent: false, rememberBrowser: false);
        }
        return RedirectToAction("Index", new { Message = ManageMessageId.ChangePasswordSuccess });
    }
    AddErrors(result);
    return View(model);
}

```

Questo metodo consente di cambiare la password all'utente all'interno dell'applicazione.

Il metodo `UserManager.ChangePasswordAsync()` consente all'utente di effettuare un cambio password solo dopo aver confermato la password corrente, il tutto come un'operazione asincrona. Il metodo prende in ingresso come parametri lo user dell'utente, la password corrente che deve essere validata prima del cambio e la nuova password specificata dall'utente.

Il metodo `UserManager.FindByIdAsync()` trova e restituisce l'utente, se presente, che corrisponde allo user ID specificato come parametro. trattasi di un metodo asincrono (come indicato dalla parola chiave *async*), mentre la parola chiave *await* sospende l'esecuzione di una funzione in attesa che una data attività venga eseguita. Se l'utente è stato trovato, quindi il metodo invocato non restituisce *null*, viene invocato il metodo `SignInManager.SignInAsync()` utilizzato per far accedere l'utente e prende come parametri l'utente da loggare, il flag che indica se il cookie di accesso deve persistere dopo la chiusura del browser e il nome del metodo utilizzato per far autenticare l'utente.

Attraverso il metodo `RedirectToAction()` viene fatto ritornare un messaggio che la password è stata modificata correttamente; come primo parametro viene passato "Index" che corrisponde al metodo presente in questa classe con la quale vengono gestiti i vari messaggi di avvenuta modifica/inserimento, mentre come secondo parametro l'elemento `ManageMessageId.ChangePasswordSuccess` che corrisponde al messaggio "Your password has been changed".

Metodo SetPassword

Il metodo `SetPassword()` viene utilizzato per impostare la password di un utente.

```

[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<ActionResult> SetPassword(SetPasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        var result = await UserManager.AddPasswordAsync(User.Identity.GetUserId(), model.NewPassword);
        if (result.Succeeded)
        {
            var user = await UserManager.FindByIdAsync(User.Identity.GetUserId());
            if (user != null)
            {
                await SignInManager.SignInAsync(user, isPersistent: false, rememberBrowser: false);
            }
            return RedirectToAction("Index", new { Message = ManageMessageId.SetPasswordSuccess });
        }
        AddErrors(result);
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}

```

Il metodo `userManager.AddPasswordAsync()` consente di aggiungere la password all'utente solo se l'utente non dispone già password; come parametri gli passiamo il nome utente e la password.

Il metodo `userManager.FindByIdAsync()` trova e restituisce l'utente, se presente, che corrisponde allo user ID specificato come parametro. trattasi di un metodo asincrono (come indicato dalla parola chiave *async*), mentre la parola chiave *await* sospende l'esecuzione di una funzione in attesa che una data attività venga eseguita. Se l'utente è stato trovato, quindi il metodo invocato non restituisce *null*, viene invocato il metodo `signInManager.SignInAsync()` utilizzato per far accedere l'utente e prende come parametri l'utente da loggare, il flag che indica se il cookie di accesso deve persistere dopo la chiusura del browser e il nome del metodo utilizzato per far autenticare l'utente.

Attraverso il metodo `RedirectToAction()` viene fatto ritornare un messaggio che la password è stata impostata correttamente; come primo parametro viene passato "Index" che corrisponde al metodo presente in questa classe con la quale vengono gestiti i vari messaggi di avvenuta modifica/inserimento, mentre come secondo parametro l'elemento `ManageMessageId.SetPasswordSuccess` che corrisponde al messaggio "Your password has been set".

Classe ScialuppaController

Metodo AggiungiProdotto

Il metodo `AggiungiProdotto()` consente la gestione dell'aggiunta di un prodotto al carrello.

```
[Authorize]
0 references
public ActionResult AggiungiProdotto(ProdottoModel prodotto)
{
    if (prodotto.QuantitaDisponibile > 0)
    {
        string updateQuantita = $"UPDATE [dbo].[Catalogo] SET [Quantità Disponibile]='{prodotto.QuantitaDisponibile - 1}' WHERE [Nome Prodotto] = '{prodotto.NomeProdotto}'";
        if (Session["cart"] == null)
        {
            List<ProdottoModel> listaProdotti = new List<ProdottoModel>();

            listaProdotti.Add(prodotto);
            Session["cart"] = listaProdotti;
            ViewBag.cart = listaProdotti.Count();
            Session["count"] = 1;
        }
        else
        {
            List<ProdottoModel> listaProdotti = (List<ProdottoModel>)Session["cart"];
            listaProdotti.Add(prodotto);
            Session["cart"] = listaProdotti;
            ViewBag.cart = listaProdotti.Count();
            Session["count"] = Convert.ToInt32(Session["count"]) + 1;
        }
        _speaker.DMLOperation(updateQuantita);
        return RedirectToAction("Catalogo", "Home");
    }
    MessageBox.Show("Il prodotto non è disponibile");
    return RedirectToAction("Catalogo", "Home");
}
```

Un prodotto per essere aggiunto al carrello deve essere disponibile in almeno una quantità (controllo che viene fatto nell'if iniziale); nel caso fosse disponibile, una volta aggiunto al carrello, la quantità del prodotto disponibile viene aggiornata e decrementata di 1 ad ogni aggiunta, questo update viene effettuato aggiornando direttamente la tabella presente nel DB SQL. In parallelo viene anche aggiornata la quantità dei prodotti presenti nel carrello. L'aggiornamento della quantità del prodotto è un'operazione di tipo DML (Data Manipulation Language) in quanto viene anche aggiornata la quantità del prodotto presente nel database.

Nel caso la quantità del prodotto sia 0, viene fatta comparire tramite una *MessageBox* un messaggio di prodotto non disponibile.

Metodo LaMiaScialuppa

Il metodo `LaMiaScialuppa()` ha il compito di mostrare il carrello degli acquisti che l'utente possiede. Questo metodo, invocabile attraverso una chiamata *http Get*, essendo utilizzato dal Resource server controlla l'effettiva autenticità del token che l'utente attuale possiede. Come nel caso del *HomeController*, laddove il token è scaduto viene forzato un nuovo accesso, altrimenti viene effettuato il refresh del token di autorizzazione e successivamente viene mostrato il carrello.

```

public async Task<ActionResult> LaMiaScialuppa()
{
    bool emailFlag = ConfigurationManager.AppSettings["LoginEmail"].Equals("NoUsername");
    if (emailFlag || OAuthSoasec.IsAuthorized().Equals("Nuova Login"))
    {
        return RedirectToAction("Login", "Account");
    }
    if (OAuthSoasec.IsAuthorized().Equals("Nuova Token"))
    {
        using (HttpClient client = new HttpClient())
        {
            HttpResponseMessage resultMessage = await client.PostAsync($"https://localhost:44391/token",
                new StringContent(string.Format("grant_type=password&username={0}&password={1}",
                    ConfigurationManager.AppSettings["LoginEmail"],
                    ConfigurationManager.AppSettings["LoginPassword"]),
                    Encoding.UTF8));

            string newToken = resultMessage.Content.ReadAsStringAsync().Result;
            var myConfiguration = WebConfigurationManager.OpenWebConfiguration("~");
            myConfiguration.AppSettings.Settings["token"].Value = newToken;
            myConfiguration.Save();
        }
    }
    return View((List<ProdottoModel>)Session["cart"]);
}

```

Nell'if sono poste in or due condizioni: la prima corrisponde al risultato della riga precedente in cui viene verificato se l'utente è loggato, mentre nella seconda se il metodo `IsAuthorized()` della classe `Authorized` (spiegata in seguito) ritorna come valore *Nuova Login* significa che il token e il refresh token sono scaduti e l'utente sarà indirizzato nell'apposita schermata in cui dovrà eseguire nuovamente l'accesso. Una volta eseguito il nuovo accesso e generato il nuovo token, l'utente potrà visualizzare i prodotti nel proprio carrello.

Metodo RimuoviProdotto

Il metodo `RimuoviProdotto()` consente la rimozione del prodotto dal carrello.

```

public ActionResult RimuoviProdotto(ProdottoModel prodotto)
{
    DataTable disp = _speaker.GetQuantitaDisponibile(prodotto.NomeProdotto);
    int toUpdate = Convert.ToInt32(disp.Rows[0]["Quantità Disponibile"]);
    string updateQuantita = $"UPDATE [dbo].[Catalogo] SET [Quantità Disponibile]=' {toUpdate + 1}' WHERE [Nome Prodotto] = '{prodotto.NomeProdotto}'";
    List<ProdottoModel> lista = (List<ProdottoModel>)Session["cart"];
    for(int i = 0; i < lista.Count; i++)
    {
        if (lista[i].NomeProdotto.Equals(prodotto.NomeProdotto))
        {
            lista.Remove(lista[i]);
            break;
        }
    }
    _speaker.DMLOperation(updateQuantita);
    Session["cart"] = lista;
    Session["count"] = Convert.ToInt32(Session["count"]) - 1;
    return RedirectToAction("LaMiaScialuppa", "Scialuppa");
}

```

Una volta rimosso il prodotto dal carrello, viene aggiornata la quantità del prodotto disponibile nella sezione catalogo prodotti e viene aggiornata anche la quantità disponibile nel DB attraverso un'operazione DML.

Metodo PiazzaOrdine

Il metodo `PiazzaOrdine()` consente di terminare e concludere l'ordine dei prodotti presenti nel carrello.

```

public ActionResult PiazzaOrdine()
{
    Session["cart"] = null;
    Session["count"] = 0;
    MessageBox.Show("Hai Completato L'acquisto. \n Verrai riportato alla Home");
    return RedirectToAction("Index", "Home");
}

```

Una volta completato l'acquisto, viene svuotato il carrello e si viene reindirizzati alla homepage simulando l'acquisto di un prodotto. Tale metodo ha la funzione di simulare un acquisto.

Folder OAuthServer

Il Folder OAuthServer è formato dai seguenti file

- JWTFormat
- OAuthServiceProviders

Classe JWTFormat

La classe JWTToken definisce le proprietà dei token e come quest'ultimo viene generato. La classe è composta dai seguenti metodi:

- Protect
- GenerateToken
- Encode

```
public class JWTFormat : ISecureDataFormat<AuthenticationTicket>
{
    private readonly string _issuer = string.Empty;
    private static Dictionary<JwtHashAlgorithm, Func<byte[], byte[], byte[]>> HashAlgorithms;
    9 references
    public enum JwtHashAlgorithm
    {
        HS256,
        HS384,
        HS512
    }
}
```

L'attributo *Issuer* è una stringa di sola lettura che contiene l'identificativo dell'entità che ha generato il token; inizialmente assumerà come valore una stringa vuota.

Successivamente mediante un enum che chiamiamo *JwtHashAlgorithm* vengono elencate le tipologie di algoritmo di cifratura che utilizzeremo; in questo caso HS256, HS384 e HS512 che indica che viene utilizzato un HMAC con rispettivamente SHA256, SHA384 e SHA512.

```
public JWTFormat(string issuer)
{
    _issuer = issuer;
    HashAlgorithms = new Dictionary<JwtHashAlgorithm, Func<byte[], byte[], byte[]>>
    {
        { JwtHashAlgorithm.HS256, (key, value) => { using (var sha = new HMACSHA256(key)) { return sha.ComputeHash(value); } } },
        { JwtHashAlgorithm.HS384, (key, value) => { using (var sha = new HMACSHA384(key)) { return sha.ComputeHash(value); } } },
        { JwtHashAlgorithm.HS512, (key, value) => { using (var sha = new HMACSHA512(key)) { return sha.ComputeHash(value); } } }
    };
}
```

Metodo Protect

Il metodo `Protect()` viene utilizzato per la generazione del token. Esso viene invocato tutte le volte che l'OAuth server riceve una chiamata post del tipo <https://localhost:xxxx/token> con accorpate le informazioni dell'utente, tra cui e-mail e password, e con il parametro *grant_type* settato a "password".

```

public string Protect(AuthenticationTicket data)
{
    if (data == null)
    {
        throw new ArgumentNullException("data");
    }
    var refreshTokenPayload = new
    {
        unique_name = data.Identity.Name,
        Issuer = _issuer,
        IssuedAt = DateTime.Now,
        ExpiresDate = DateTime.Now.AddMinutes(20),
        Type = "Refresh Token"
    };
    var refreshToken = Encode(refreshTokenPayload, ConfigurationManager.AppSettings["RefreshSecretKey"], JwtHashAlgorithm.HS256);

    var payload = new
    {
        unique_name = data.Identity.Name,
        Audience = "SoasecUser",
        Issuer = _issuer,
        IssuedAt = DateTime.Now,
        ExpiresDate = DateTime.Now.AddMinutes(3),
        RefreshToken = refreshToken,
        Type = "Access Token"
    };
    string result = Encode(payload, ConfigurationManager.AppSettings["SecretKey"], JwtHashAlgorithm.HS256);
    return result;
}

```

Nel progetto, la creazione di un token corrisponde alla generazione di due diversi token: un *Access Token* e un *Refresh Token*. Quest'ultimo viene incorporato nell'*Access Token*.

Nel metodo viene passato un oggetto di tipo *AuthenticationTicket*, classe che viene utilizzata per creare un oggetto che rappresenta il ticket di autenticazione basata su una form per identificare un utente.

Se l'oggetto passato è *null* viene generata un'eccezione, in caso contrario si procede con la creazione dei due token; per prima cosa viene creato il *refresh token* aventi le seguenti proprietà:

- *unique_name*: indica il nome dell'utente per il quale viene generato il token
- *Issuer*: identificativo di colui che ha generato il token
- *IssuedAt*: indica la data e l'ora in cui è stato creato il token
- *ExpiresDate*: indica la data e l'ora dopo il quale il token non è più valido
- *Type*: indica il tipo, in questo caso il *refreshToken*

Con il metodo *Encode()* viene firmato il *refreshToken* utilizzando HMAC con l'algoritmo con SHA256.

Successivamente creiamo anche l'*access token* aventi le seguenti proprietà:

- *unique_name*: indica il nome dell'utente per il quale viene generato il token
- *Audience*: rappresenta il tipo di utente su cui viene applicato il token
- *Issuer*: identificativo di colui che ha generato il token
- *IssuedAt*: indica la data e l'ora in cui è stato creato il token
- *ExpiresDate*: indica la data e l'ora dopo il quale il token non è più valido
- *RefreshToken*: perché all'interno contiene anche il *RefreshToken*
- *Type*: indica il tipo di token, in questo caso il *AccessToken*

Attraverso il metodo *Encode()* viene firmato l'*accessToken* utilizzando l'algoritmo con SHA256. Il risultato viene archiviato in una variabile e rilasciato attraverso la *return*.

Metodo Encode

Il metodo Encode() viene utilizzato per la creazione del token JWT.

```
public static string Encode(object payload, string key, JwtHashAlgorithm algorithm)
{
    return Encode(payload, Encoding.UTF8.GetBytes(key), algorithm);
}

1 reference
public static string Encode(object payload, byte[] keyBytes, JwtHashAlgorithm algorithm)
{
    var segments = new List<string>();
    var header = new { alg = algorithm.ToString(), typ = "JWT" };

    byte[] headerBytes = Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(header, Formatting.None));
    byte[] payloadBytes = Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(payload, Formatting.None));

    segments.Add(Base64UrlEncode(headerBytes));
    segments.Add(Base64UrlEncode(payloadBytes));

    var stringToSign = string.Join(".", segments.ToArray());

    var bytesToSign = Encoding.UTF8.GetBytes(stringToSign);

    byte[] signature = HashAlgorithms[algorithm](keyBytes, bytesToSign);
    segments.Add(Base64UrlEncode(signature));

    return string.Join(".", segments.ToArray());
}
```

Abbiamo due metodi chiamati Encode() che differiscono fra di loro per la firma, ma uno non fa altro che richiamare l'altro.

Utilizziamo la variabile *segments*, una lista di stringhe, per costruire il nostro *JWTToken* formato, per sua definizione, da tre parti: *header*, *payload* e *signature*.

- L'header è composto da due parti: il tipo di token e l'algoritmo di crittografia utilizzato per la firma
- Payload: contiene le informazioni che devono essere inviate all'applicazione
- Signature: firma creata utilizzando la codifica Base64 dell'header e del payload insieme all'algoritmo di firma specificato come parametro

Le generazione del token si ottiene codificando l'header e il payload in Base64, unendo poi i risultati separandoli da un punto (.) attraverso l'utilizzo del metodo *string.join*; successivamente viene applicato l'algoritmo indicato nell'header al risultato dell'unione del payload e dell'header e si ottiene così una chiave segreta.

Classe OAuthServiceProvider

La classe OAuthServiceProvider si occupa di verificare e rilasciare token. I metodi di questa classe sono i primi ad essere invocati tutte le volte che viene raccolta una richiesta http *Post* del tipo:

<https://localhost:xxxx/token>.

```
public override async Task ValidateClientAuthentication(OAuthValidateClientAuthenticationContext context)
{
    // Il client è sempre fidato
    /*
    * Si potrebbe controllare la validità del client attraverso una coppia di valori (string clientID, string clientSecret)
    * Pseudo codice
    * if(clientID == IDClientRegistrato AND clientSecret == SecretRegistrato)
    * return OK; --> Client approvato
    * else
    * return Rejected; --> Client rigiutato
    * PS: IDClientRegistrato e SecretRegistrato sono registrati in una tabella del DB utilizzato da OAuth Server
    */
    context.Validated();
}
```


Nel progetto si è voluto validare sempre il client, essendo questo sul medesimo host su cui era online anche l'OAuth Server. Tuttavia, si potrebbe generare una coppia di valori (ClientID, SecretClient), con cui andare a verificare, alla pagina <https://localhost:xxxx/token>, che questi valori siano corretti per validare il client richiedente.

```
public override async Task GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)
{
    context.OwinContext.Response.Headers.Add("Access-Control-Allow-Origin", new[] { "*" });
    var userManager = context.OwinContext.GetUserManager<ApplicationUserManager>();

    ApplicationUser user = await userManager.FindAsync(context.UserName, context.Password);
    if (user == null)
    {
        context.SetError("invalid_grant", "The user name or password is incorrect.");
        return;
    }
    else
    {
        ClaimsIdentity oAuthIdentity = await user.GenerateUserIdentityAsync(userManager, "JWT");
        var ticket = new AuthenticationTicket(oAuthIdentity, null);
        context.Validated(ticket);
    }
}
```

Il metodo `GrantResourceOwnerCredentials()` viene invocato successivamente a `ValidateClientAuthentication()` e ha il compito di validare le credenziali dell'utente passate attraverso la chiamata `http Post` <https://localhost:xxxx/token>.

Nella parte dell'`else` viene generato un ticket contenente le informazioni quali username e il tipo di token da generare (in questo caso `jwt`); infine viene poi validato il ticket. Tale ticket, viene poi inviato al middleware `Owin` che invocherà il metodo `Protected` della classe `JWTFormat` per costruire e rilasciare il token.

Folder ResourceServer

Classe Authorizer

La classe `Authorizer` viene utilizzata per controllare se il token salvato nel file `Web.config`, e ottenuto dal client, sia valido oppure scaduto. Per controllare il token, si avvale del metodo `ControllToken()` della classe `TokenManager` spiegato di seguito.

```
public class Authorizer
{
    TokenManager tokenController = new TokenManager();
    6 references
    public string isAuthorized()
    {
        return tokenController.ControllToken(ConfigurationManager.AppSettings["token"]);
    }
}
```

Classe TokenManager

La classe `TokenManager` viene utilizzata per verificare il token e per la gestione dell'accesso a determinate risorse.

I metodi fondamentali di questa classe sono:

- `ControllToken`
- `getRefreshToken`
- `isValid`
- `Decode`

Metodo ControllToken

Il metodo ControllToken() viene utilizzato per stabilire la validità di un token.

```
public string ControllToken(string token)
{
    string validity = isValid(token);
    if (validity.Equals("Token Scaduto"))
    {
        string refValidity = isValid(getRefreshToken(token));
        if (refValidity.Equals("Token Valido"))
            return "Nuovo Token";
    }
    else if (validity.Equals("Token Valido"))
        return "Ok";
    return "Nuova Login";
}
```

Il metodo prende in ingresso una stringa che rappresenta il token su cui vengono eseguiti una serie di controlli utilizzando il metodo *isValid()* (spiegato appena dopo); come risultato ritorna

- “Ok” se il token è valido;
- “Nuovo Token” se il token deve essere refreshato con uno nuovo;
- “Nuova Login” se sia l’access Token che il refresh Token sono scaduti.

Questi risultati verranno poi gestiti dai controller utilizzati dal resource server per consentire, oppure negare, l’accesso alle risorse richieste dall’utente.

Metodo isValid

Il metodo isValid() prende come parametro un token e ne verifica la validità.

```
private string isValid(string token)
{
    string toBeControlled = Decode(token, ConfigurationManager.AppSettings["SecretKey"], true);
    string payload = string.Empty;
    if (toBeControlled.Equals("Firma Non Valida!"))
    {
        payload = Decode(token, ConfigurationManager.AppSettings["RefreshSecretKey"], true);
    }
    else
    {
        payload = toBeControlled;
    }

    if (!payload.IsNullOrEmpty())
    {
        JObject payloadData = JObject.Parse(payload);
        DateTime ExpiresDate = (DateTime)payloadData["ExpiresDate"];
        int result = DateTime.Compare(DateTime.Now, ExpiresDate);
        if (result <= 0)
        {
            return "Token Valido";
        }
        else
        {
            return "Token Scaduto";
        }
    }
    return "Token Non Valido";
}
```

Nella prima riga viene chiamato il metodo Decode (spiegato appena dopo) utilizzato per “decodificare” il token e viene restituita una stringa che rappresenta il payload del token e, a seconda del valore che assume il terzo parametro (*true* o *false*), viene specificato se fare la verifica o meno del medesimo token. Come secondo parametro viene passato il *secret key* che rappresenta la chiave simmetrica utilizzata fra client e server per effettuare la firma del JWT Token. Se il risultato dell’applicazione del metodo Decode è “Firma Non Valida” significa che la firma non è valida e può trattarsi di un refresh token; in caso contrario la firma risulta valida e si prosegue.

Un successivo controllo consiste nel verificare se il *payload* non è *null* oppure se corrisponde ad una stringa vuota; se non è *null*, si recupera l'*ExpiresDate* che rappresenta la data e l'orario dopo il quale il token non è più valido. Per eseguire questo controllo, utilizzo il metodo *Compare* della classe *DateTime* passando come parametro *DateTime.now* che rappresenta la data e l'orario attuale ed *ExpiresDate*; il metodo ritornerà

- -1 se *DateTime.Now* è minore di *ExpiresDate*
- 0 se *DateTime.Now* è uguale a *ExpiresDate*
- +1 se *DateTime.Now* è maggiore di *ExpiresDate*

Se il risultato è -1 oppure 0 il token è ancora valido, mentre se è +1 il token è scaduto.

In tutti gli altri casi il metodo ritorna che il Token non è valido.

Metodo Decode

Il metodo *Decode()* viene utilizzato per recuperare i valori in chiaro dei token JWT.

```
public string Decode(string token, string key, bool verify)
{
    var parts = token.Split('.');
    var header = parts[0];
    var payload = parts[1];
    byte[] crypto = Base64UrlDecode(parts[2]);

    var headerJson = Encoding.UTF8.GetString(Base64UrlDecode(header));
    var headerData = JObject.Parse(headerJson);
    var payloadJson = Encoding.UTF8.GetString(Base64UrlDecode(payload));
    var payloadData = JObject.Parse(payloadJson);

    if (verify)
    {
        var bytesToSign = Encoding.UTF8.GetBytes(string.Concat(header, ".", payload));
        var keyBytes = Encoding.UTF8.GetBytes(key);
        var algorithm = (string)headerData["alg"] == "HS256" ? "RS256" : (string)headerData["alg"];
        var signature = HashAlgorithms[GetHashAlgorithm(algorithm)](keyBytes, bytesToSign);
        var decodedCrypto = Convert.ToBase64String(crypto);
        var decodedSignature = Convert.ToBase64String(signature);

        if (decodedCrypto != decodedSignature)
        {
            return $"Firma Non Valida!";
            // throw new ApplicationException(string.Format("Invalid signature. Expected {0} got {1}", decodedCrypto, decodedSignature));
        }
    }

    return payloadData.ToString();
}
```

Il metodo *Decode()* prende in ingresso come parametro il token, la chiave segreta che rappresenta la chiave simmetrica e un parametro booleano che dice se deve essere eseguita la verifica o meno.

La prima cosa che viene fatta è eseguire un *split* del token utilizzando come separatore il punto (.) dalla quale si ottengono le tre parti del token jwt ovvero l'header, il payload e la signature.

La fase successiva consiste nel convertire la firma in byte e ottenere il formato JSON dell'header e del payload.

Se il booleano passato come parametro è *true*, viene eseguita la verifica della firma; per eseguire questa fase viene generata una nuova firma a partire dall'header e dal payload del token passato come parametro e confrontata con la signature estrapolata dal token (terza parte del token JWT). La generazione della firma avviene concatenando l'header e il payload, viene poi recuperata la chiave segreta e l'algoritmo con cui codificare la signature, ed infine viene costruita la firma. Se la firma non è valida, il metodo ritorna che la firma non è valida, altrimenti viene ritornato il payload del token in formato stringa.

Metodo getRefreshToken

Il metodo *getRefreshToken()* non fa altro che estrapolare da un token il refresh token e verificarne la validità, utilizzando il metodo *Decode()* definito precedentemente.

```
private string getRefreshToken(string token)
{
    string payload = Decode(token, ConfigurationManager.AppSettings["SecretKey"], true);
    JObject payloadData = JObject.Parse(payload);
    return (string)payloadData["RefreshToken"];
}
```

Folder Utils

Classe DBSpeaker

La classe DBSpeaker viene utilizzata per collegare l'applicazione al database e alla tabelle utilizzate e gestite dal ResourceOwner.

Metodo connectToDB

Il metodo connectToDB() viene utilizzato per ritornare una connessione ad un database di SQL Server.

```
public static SqlConnection connectToDB()
{
    string connectionString = ConfigurationManager.ConnectionStrings["DefaultConnection"].ConnectionString;
    SqlConnection connection = new SqlConnection(connectionString);
    if (connection.State == ConnectionState.Open)
    {
        connection.Close();
    }
    else
    {
        connection.Open();
    }
    return connection;
}
```

Il metodo ConfigurationManager.ConnectionStrings ottiene i dati di ConnectionStringsSection per la configurazione predefinita dell'applicazione corrente; quest'ultimo fornisce l'accesso a livello di codice alla sezione del file di configurazione relativa alle stringhe di connessione.

Attraverso la classe SqlConnection viene creata una connessione ad un SQL Server.

Nel metodo if viene verificata se la connessione è aperta; se la connessione è aperta, viene chiusa, altrimenti viene aperta.

Alla fine viene poi ritornata la connessione.

Metodo DMLOperation

Il metodo DMLOperation() viene utilizzato per consentire l'esecuzione di operazioni di tipo DML sul DB; come parametro viene passato una stringa che corrisponde alla query da eseguire sul DB.

```
public bool DMLOperation(string SQLquery)
{
    SqlCommand = new SqlCommand(SQLquery, DBSpeaker.connectToDB());
    int x = SqlCommand.ExecuteNonQuery();
    if (x == 1)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

La classe SqlCommand viene utilizzata per eseguire delle query SQL all'interno di un db; questa classe prende in input due parametri: il primo che corrisponde alla stringa (che corrisponde ad una query SQL) passata come parametro al metodo spiegato e il secondo indica invece il db a cui si vuole connettersi.

Successivamente viene invocato il metodo `SqlCommand.ExecuteNonQuery` che consente di eseguire un'istruzione SQL e ritorna un tipo di dato intero che corrisponde al numero di righe modificate; se ritorna 1 significa che è stata modificata una sola riga e viene ritornato *true*, mentre in tutti gli altri casi viene ritornato *false*. Ad 1 ritorniamo *true* perché viene modificata solo una riga della tabella per volta.

Metodo SelectAll

Il metodo `SelectAll()` ritorna una tabella che contiene i risultati della query.

```
public DataTable SelectAll(string query)
{
    SoasecAdapter = new SqlDataAdapter(query, DBSpeaker.connectToDB());
    DataTable Soasec_Oauth2_DB = new DataTable();
    SoasecAdapter.Fill(Soasec_Oauth2_DB);
    return Soasec_Oauth2_DB;
}
```

Al metodo viene passato come parametro la stringa contenente la query SQL da considerare. La classe `SqlDataAdapter` rappresenta un set di comandi dati (nel nostro caso corrisponde alla query passata) e una connessione al database, che non sono altro che i parametri che vengono passati al costruttore della classe e utilizzati per riempire l'oggetto `DataTable` che viene definito e che non rappresenta altro che una tabella dati salvata in memoria.

Il metodo `fill` della classe `SqlDataAdapter` viene utilizzato per aggiungere righe all'oggetto di tipo `DataTable`.

Il metodo poi ritorna un oggetto di tipo `DataTable` con i dati.

Metodo GetQuantitaDisponibile

Il metodo `GetQuantitaDisponibile()` viene utilizzato per ritornare la quantità disponibile di un singolo prodotto.

```
public DataTable GetQuantitaDisponibile(string nomeProdotto)
{
    string query = $"SELECT [Quantità Disponibile] FROM [dbo].[Catalogo] WHERE [Nome Prodotto] = '{nomeProdotto}'";
    SoasecAdapter = new SqlDataAdapter(query, DBSpeaker.connectToDB());
    DataTable risultato = new DataTable();
    SoasecAdapter.Fill(risultato);
    return risultato;
}
```

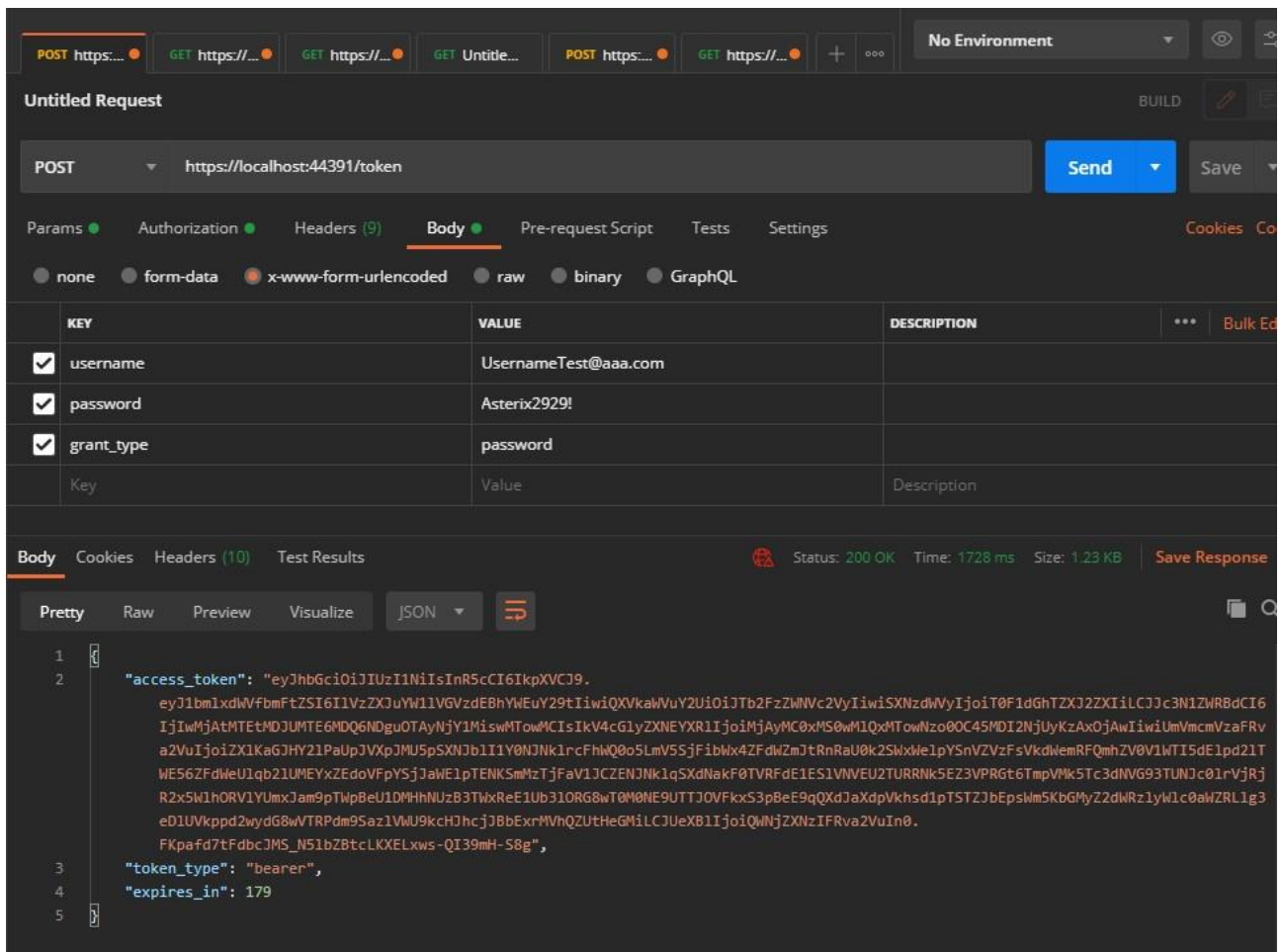
Il metodo prende in ingresso come parametro un dato di tipo `string` che corrisponde al nome del prodotto della quale si vogliono avere informazioni in merito alla quantità. Viene poi creata una stringa attraverso cui viene formalizzata la query SQL per estrapolare il dato inerente alla quantità del prodotto.

La classe `SqlDataAdapter` rappresenta un set di comandi dati (nel nostro caso la query passata) e una connessione al database, che non sono altro che i parametri che vengono passati al metodo, utilizzati per riempire l'oggetto `DataTable` che abbiamo definito e che non rappresenta altro che una tabella dati salvata in memoria.

Il metodo `fill` della classe `SqlDataAdapter` viene utilizzato per aggiungere righe all'oggetto di tipo `DataTable`; nel nostro caso ritorniamo un oggetto di tipo `DataTable` che contiene la quantità disponibile del prodotto per la quale si vuole avere l'informazione.

Test Generazione Token con Postman

Come primo approccio abbiamo utilizzato il tool *Postman* che viene impiegato per effettuare dei test alle API. Come primo passo è stata inviata una richiesta `http Post` per la generazione del token di accesso, ottenendo come risultato un token simile a quello riportato nella seguente figura.



Per controllare l'effettiva validità del token è stato utilizzato il sito [JWT.io](https://jwt.io) [8], nello specifico la sezione *Debugger*, in cui viene copiato il token rilasciato nel campo *Encoded* del sito e passata la chiave utilizzata per firmare il token nel campo *Decoded*. Una volta passati questi parametri viene notificata la validità effettiva del token e, come nell'esempio riportato sotto, si ha un esito positivo. Inoltre, tramite [JWT.io](https://jwt.io), è possibile vedere i campi e le informazioni che vengono salvate all'interno del token come, per esempio,

- il nome dell'utente abilitato all'utilizzo di quel token
- il tipo di Audience a cui viene rilasciato il token, in questo caso a tutti gli utenti "SoasecUser"
- L'Issuer, nonché colui che rilascia il token: "OAuthServer".

Vengono inoltre riportate due date: una è inerente alla data di rilascio del token, mentre la seconda riguarda la data di scadenza. Viene incorporato un refresh token che viene utilizzato per rigenerare un nuovo token di accesso ogni volta che esso scade.

Il refresh token viene firmato con una chiave simmetrica differente rispetto a quella utilizzata per firmare il token di accesso. Tali chiavi vengono condivise dal Resource Server e dall'OAuth Server.

PASTE A TOKEN HERE

Signature Verified

EDIT THE PAYLOAD AND SECRET

[SHARE JWT](#)

Encoded

Signature Verified

Decoded EDIT THE PAYLOAD AND SECRET

[SHARE JWT](#)

È stato inserito un campo contenente il nome del possessore del token e un campo che indica chi lo ha rilasciato. Anche per questo token vengono rilasciate la data di rilascio e la data di scadenza del token, ed infine il campo che specifica il tipo di token.

Riferimenti

- [1] Microsoft, «What Is ASP.NET?: .NET.» Microsoft,» [Online]. Available: dotnet.microsoft.com/learn/aspnet/what-is-aspnet.
- [2] F. Camarlinghi, «Pattern MCV: Cos'è, Vantaggi e Come Utilizzarlo.,» [Online]. Available: www.html.it/pag/18299/il-pattern-mvc/.
- [3] C. C. M. D. S. A. L. M. L. S. M. Daniele Bochicchio, C# 6 e Visual Studio 2015, Guida completa per lo sviluppatore, Editore Ulrico Hoepli Milano, 2015.
- [4] Microsoft, «Panoramica di Visual Studio,» [Online]. Available: <https://docs.microsoft.com/it-it/visualstudio/get-started/visual-studio-ide?view=vs-2019>.
- [5] GitHub, «GitHub,» [Online]. Available: <https://it.wikipedia.org/wiki/GitHub>.
- [6] «Cos'è OAuth 2?,» [Online]. Available: <https://www.cyber-security-libro.it/articoli/cos-e-oauth-2/>.
- [7] «Mappare i metodi CRUD sulle azioni HTTP,» [Online]. Available: <https://www.html.it/pag/19597/mappare-le-azioni-crud-sui-metodi-http/>.
- [8] «JWT Site,» [Online]. Available: <https://jwt.io/>.