Progetto Prova Finale

Nome: Stefano Di Nucci   
Codice Persona: 10763612   
Repository: <https://github.com/Stepness/polimi-project>

Table of Contents

[Requisiti 1](#_Toc175494076)

[Tecnologie 2](#_Toc175494077)

[Architettura 3](#_Toc175494078)

[Class Diagram 4](#_Toc175494079)

[UseCase Diagram 5](#_Toc175494080)

[Backend 5](#_Toc175494081)

[Design Patterns 5](#_Toc175494082)

[Policies e Token 9](#_Toc175494083)

[Database 12](#_Toc175494084)

[Unit Tests 13](#_Toc175494085)

# Requisiti

Il progetto consiste in un servizio web che cerca di risolvere il problema di condivisione di file all’interno di una azienda.  
Quindi invece di dover setuppare dischi all’interno della rete aziendale tramite NFS/SMB si può ricorrere a questo servizio.

È costituito da una applicazione web che permette di caricare file da una interfaccia web e poterli scaricare da qualunque utente che abbia un account.

Il servizio funziona come un file storage, in cui ogni utente, dipendentemente dal proprio livello di autorizzazione, può leggere, caricare file o modificarli.

Agli utenti può essere assegnato uno tra I seguenti ruoli:

**Admin**:

* Gestisce il ruolo degli altri utenti. Il servizio richiede solo un primo controllo del ruolo, di conseguenza l’unica gestione dei ruoli attualmente disponibile è solo quella di promozione degli utenti da **Reader** a **Writer**
* In aggiunta ha anche i permessi che ha un **Writer**

**Writer**:

* Upload dei file
* Rinominare file
* Cancellare file
* Scaricare file

**Reader**:

* Scaricare file

Alla prima registrazione viene assegnato di default il ruolo di Reader.  
Per essere promosso a Writer c’è bisogno di un Admin che se ne occupi.  
Un Admin può essere solo settato da database.

# Tecnologie

**Frontend**

Il client è costituito da HTML, CSS e Javascript, con l’ausilio di Jquery per le richieste HTTP al backend.

**Backend**

Il servizio è implementato in C# attraverso .NET8.

Il servizio gestisce l’autenticazione e I livelli di autorizzazione dell’API.

L’autenticazione è una semplice funzione di hash che compara la password passata dal client a quella che si trova nel database.

L’autorizzazione si basa sul protocollo OAuth2, questo significa che a seguito dell’autenticazione, il server genera un token in formato json (chiamato Json Web Token o anche JWT), che viene rimandato al client.

Il client in ogni Request dovrà inviare al server questo token per potersi autorizzare.

Questo token contiene il nome dell’utente e il suo ruolo, e ha una scadenza di 1 giorno.

**Dati**

Come database viene utilizzato un database NoSQL, in particolare Cosmosdb, un servizio PaaS hostato sul cloud Azure.

Il servizio all’avvio si aspetta una stringa di connessione a questo servizio, se non è presente il servizio si avvia comunque e salverà i dati InMemory.

**Test**

Come framework di test è stato usato xUnit.

In aggiunta sono stati utilizzati:

Autofixture: libreria per generare oggetti con valori randomici

Fluent Assertion: libreria che migliora la lettura e la scrittura delle assertion fatte dai test

Nsubstitute: libreria per creare i mock nei test

# Architettura

**A blue and black text on a white background

Description automatically generated**

Il servizio utilizza il pattern MVC, per cui abbiamo un client che si occupa di mostrare i dati richiesti al Controller del servizio, i quali sono salvati in un database.

# Class Diagram

**A diagram of a software company

Description automatically generated**

Il package Models si occupa di contenere tutti i modelli che vengono utilizzati all’interno del servizio.

Il package Settings espone dei dati utilizzati per configurare l’applicazione, di conseguenza sono principalmente dei valori statici.

Il package Extension contiene dei metodi di estensione per configurare alcune delle funzionalità del servizio.  
In particolare la classe AuthExtension si occupa di configurare il comportamento del token di autenticazione, e le policy applicate agli endpoint necessari.

Il package Controllers contiene i controller a cui vengono fatte le richieste, e sfrutta i servizi Repository per interfacciarsi con le basi di dati.

Il package Services contiene la definizione delle diverse repository per accedere alle basi di dati (CosmosDB e in-memory), la Factory per istanziare la classe per comunicare con CosmosDB e l’helper per generare il token di autenticazione.

# UseCase Diagram

**A diagram of a software system

Description automatically generated**

Inizialmente uno user alla pagina di login può decidere se loggarsi o registrarsi.

Dopo che questo è accaduto, il servizio invia un token al client che gli permette di autenticarsi in ogni successiva richiesta.

Un admin è l’unico a poter gestire i ruoli degli altri user.

Un reader può solamente vedere e scaricare i file.

Un writer può caricare, scaricare e modificare i file.

Ogni operazione eseguita sui file sottintende una gestione dei dati dei file attraverso il servizio.

# Backend

## Design Patterns

I design pattern utilizzati principalmente dal servizio sono:

* Dependency injection (implementata dal framework)
* Repository
* Factory

La **dependency injection** è utilizzata estensivamente all’interno di tutto il progetto.

.NET ci espone un container di default nel quale possiamo registrare i servizi e il loro lifespan (Transient, Scoped, Singleton).

Questi servizi vengono tutti registrati nella classe Program contenente la funzione Main().

Di seguito possiamo vedere qualche esempio:

A screen shot of a computer program

Description automatically generated

Il servizio può essere avviato sia con che senza una stringa di connessione, questo perché grazie al polimorfismo l’implementazione delle classi che si occupano di interfacciarsi con il database cambiano opportunamente, in funzione della presenza o meno di un valore “CosmosDbConnectionString” all’interno del file di configurazione.

In particolare, le classi che si occupano di interfacciarsi con le basi di dati sono le interfaccie **IRepositoryData** e **IRepositoryUsers**.

Da notare come le classi che implementano il servizio in memoria abbiano uno scope Singleton, in questo modo la classe che si occupa di salvare i dati è sempre la stessa all’interno dell’applicazione, fino a quando non viene riavviata.

Il pattern **Repository** è un pattern che si basa sull’idea di creare un intermediario tra la business logic e il database.

In questo modo abbiamo la possibilità di definire solo le firme dei metodi che devono interfacciarsi con elementi esterni per poi implementarli a seconda dei nostri bisogni.

Vediamo l’esempio dell’interfaccia **IRepositoryData**, il cui compito è quello di eseguire le query per gestire i File.

A screen shot of a computer program

Description automatically generated

In questo caso noi definiamo solamente le azioni che la nostra interfaccia espone, come il dover scaricare un file attraverso **DownloadFile()** passando come parametro il nome del file, o ottenere la lista di tutti i file attraverso **GetAllFiles()**

Dato che noi vogliamo supportare sia un database Cosmos che una raccolta dati in memoria, possiamo vedere come l’implementazione cambia nelle classi che implementano questa interfaccia:

A screen shot of a computer program

Description automatically generatedNell’implementazione di cosmos ci aspettiamo che venga eseguita una chiamata **UpsertItemAsync()** al database Cosmos.

A screenshot of a computer program

Description automatically generated

Nell’implementazione **InMemory**, viene solamente istanziato un array vuoto di file in memoria, e in caso di **Upsert()** il file viene aggiunto all’array.

Infine, il pattern **Factory** ci permette di creare un instanza della classe che si deve occupare della connessione a CosmosDB utilizzando una certa configurazione.

A screen shot of a computer

Description automatically generated

In particolare configuriamo un **CosmosClient** utilizzando la stringa di connessione presente nella configurazione e definiamo che ogni richiesta che viene passata al servizio deve essere formattata in modo tale che le proprietà siano scritte in CamelCase, questo perché quando definiamo una classe, le proprietà sono scritte in PascalCase, e quando viene fatta una richiesta al servizio, di default vengono utilizzate le proprietà così come sono definite all’interno della classe.

Se non si definisce questa formattazione, il servizio non è in grado di distinguere le proprietà.

## Policies e Token

Per determinare quali endopint possono essere chiamati da quale utente, si fa utilizzo di policy di autenticazione.

Queste policy sfruttano il payload dei Json Web Token per determinare il livello di autorizzazione di un utente.

Un JWT è costituito da tre parti scritte in formato Json. La prima è l’header del token, contenente informazioni sulla struttura del token stesso.

La seconda è il payload, contenente le informazioni dell’utente, in particolare il suo username e il suo ruolo.  
Nell’esempio seguente possiamo vedere come il sub (subject, ossia lo username) “admin” abbiamo come ruolo “Admin”

La terza parte serve al server per verificare che le prime due parti non siano state modificate.

A screenshot of a computer

Description automatically generated

Il framework per determinare se uno user può chiamare un certo endpoint fa utilizzo di policy definite nel metodo di estensione **AddCustomAuthorization()**

A screen shot of a computer program

Description automatically generated

Vediamo come siano state definite due policy.

La prima è **AdminUserPolicy**, per cui ci si aspetta che il claim chiamato **Role** abbia come valore **Admin**.

La seconda è **WriterUserPolicy**, per cui ci si aspetta che il claim chiamato **Role** abbia come valori **Admin** oppure **Writer**.

Vediamo l’esempio della funzione **Upload()** disponibile dal controller **Blob**.

Per applicare la policy ci basta definire l’attributo **Authorize()** passano il nome della policy che vogliamo applicare.

Nel nostro caso, la funzione **Upload()** può essere chiamata solo da uno user che presenta un JWT contenente la proprietà **Role** uguale ad **Admin** o **Writer**.

**A screen shot of a computer program

Description automatically generated**

# Database

Come database è stato utilizzato un database NoSql, data la facilità di implementazione e la non necessità di creare relazioni all’interno della base di dati.

L’implementazione per l’accesso alla base di dati, si trova in **CosmosRepositoryData**, **CosmosdRepositoryUser** e **CosmosLinqQuery**.

Le classi repository si aspettano di ricevere in input una classe per istanziare il Client per connettersi al database e una classe per gestire la query che viene eseguita nel database.

Di seguito l’esempio di **CosmosRepositoryData**

A screen shot of a computer

Description automatically generated

Un container in Cosmos è l’equivalente di una tabella in un database Sql.

La classe **ICosmosLinqQuery** viene iniettata come parametro per rendere più semplice la scrittura di Unit test.

Questa classe contiene solamente un metodo che permette di creare query contro il database, ritornando una lista generics. In questo modo questo metodo può essere mockato negli Unit test per tornare una lista custom, senza effettivamente comunicare con il database.

A screen shot of a computer program

Description automatically generated

# Unit Tests

Gli unit test sono stati scritti con xUnit.

Ogni test contiene un **sut** (system under test) che è il soggetto del nostro test.

Ogni test esegue delle assertion scritte con l’aiuto di [FluentAssertion](https://fluentassertions.com/) per renderne più semplice la scrittura e la lettura.

Essendo ogni servizio e controller nell’applicazione istanziato con l’aiuto della dependency injection ci risulta semplice testare un modulo alla volta mockando tutto il resto.

La libreria che genera i mock è [NSubstitute](https://nsubstitute.github.io/).

I nomi dei file seguono tutti la convenzione ***When\_Should*** dove when specifica cosa stiamo testando e il modo in cui stiamo testando, e should specifica come si dovrebbe comportare il nostro **sut**.

Di seguito l’esempio di un test, per cui testiamo che il controller che si occupa dei blob, ci ritorni Ok quando facciamo un upload.

**A screen shot of a computer program

Description automatically generated**

Il costruttore della classe di test viene chiamato prima di ogni test, quindi il nostro **sut** viene pulito tra un test e l’altro.

Possiamo vedere che passando al **sut** un file con nome **test.txt**, e tipo **text/plain**, ci aspettiamo che il modulo interno **RepositoryData** (opportunamente mockato) abbia ricevuto una singola chiamata che aveva come input proprio i dati passati inizialmente al **sut**.

Sono presenti 49 tests, divisi per classe (**sut**).

A screenshot of a computer

Description automatically generated

La copertura è dell’82%.A screenshot of a computer

Description automatically generated