INFRASTRUTTURA CON EF SU X,Y, Z:

Partiamo con il presupposto che sia già stata creata la parte di Core della nostra applicazione, quindi che abbiamo già a nostra disposizione tutta la parte che concerne i modelli delle entità coinvolte. Nel caso specifico quindi suppongo di avere già a disposizione le classi delle entità X Y Z.
Ora possiamo quindi occuparci dell'infrastruttura che dovrà andare a definire l'effettiva persistenza delle nostre entità sfruttando l'entity Framework.
Dobbiamo andare a definire le Repositories, le quali si occuperanno di effettuare tutte le operazioni inerenti al db per la gestione delle varie entità, quindi l'aggiunta, rimozione e modifica.
Avremo di conseguenza le classi XRrepository, YRepository e ZRepository alle quali posso far estendere delle interfacce create in precedenza per aumentare la manutenibilità del progetto.
Per poter far uso del package EF bisogna anche creare una classe contex che estenda la classe del framework DbContext.
Creerei quindi la classe XYZContext dove andrò a creare dei set come variabili locali che andranno poi a costituire le tabelle del database, utilizzando il tipo DbSet<> messo a disposizione dal framework. Creo così DbSet<X>, DbSet<Y>, DbSet<Z>.
Successivamente vado a sovrascrivere il metodo OnModelCreating della classe DbContext per poter andare a definire le proprietà che dovranno essere applicate alle tabelle definite prima in fase di creazione.
Qui ci si trova a dover fare una scelta sul come andare ad indicare proprietà utili alle tabelle del database, come chiave primaria, proprietà delle colonne o chiavi esterne per le relazioni tra tabelle.
Per farlo si hanno diverse opzioni: si possono definire attraverso annotazioni nelle entità, all'interno della classe Context oppure tramite la creazione di classi apposite chiamate Configurations.

La scelta ricade sulla terza opzione in quanto permette di tenere separate le classi delle entità dalle dipendenze del framework (problema del primo metodo) e consente una separazione dei ruoli di ogni classe (problema del secondo metodo).
Andiamo quindi a creare le classi XConfiguration, YConfiguration e ZConfiguration che estendono IEntityTypeConfiguration<X/Y/Z>.
In ogni classe andiamo a definire il metodo Configure che prende un builder (di tipo EntityTypeBuilder<X,Y,Z>) e va a definire per le varie entità da trasformare in tabelle qual è la chiave primaria, proprietà delle colonne e le relazioni, che siano da 1 ad n, n ad n ecc., indicandone anche la chiave esterna.
Nella classe Context, nella funzione OnModelBuilding, andiamo ad utilizzare il metodo ApplyConfigurationsFromAssembly che va a cercare tra i file binari del progetto quelli che estendono IEntityTypeConfiguration per applicare le configurazioni lì definite.

Una volta definito il DbContext e le configurazioni, si utilizzano le migrazioni di EF Core per creare e aggiornare lo schema del database.

Motore di binding e validazione delle richieste:

Quando una richiesta HTTP arriva alla Web API, entra in gioco il **motore di model binding**, che ha il compito di estrarre i dati presenti nella richiesta e trasformarli in **oggetti C#** utilizzabili all'interno delle varie classi che implementano logica di business e che lavorano con i suddetti oggetti. I dati possono provenire dal **body della richiesta**, dalla **route**, dalla **query string** o dagli **header**, e vengono mappati automaticamente sulle proprietà dei parametri del metodo dell'action. Questo processo avviene **prima dell'esecuzione dell'action** e consente al controller di lavorare direttamente con **oggetti tipizzati**, evitando la gestione manuale dei dati grezzi della richiesta.

Il binding lavora tipicamente su **DTO di request**, che permettono di separare il **modello esposto all'esterno** dalle **entità di dominio** interne all'applicazione. Una volta completata la fase di binding, si passa alla **fase di validazione del modello**. La validazione ha lo scopo di verificare che i dati ricevuti rispettino **regole di correttezza e coerenza** prima che la **logica applicativa** venga eseguita.

In **ASP.NET**, la validazione può essere implementata definendo **regole direttamente sui DTO**, ad esempio tramite **annotazioni**, oppure delegandola a **servizi di validazione dedicati**. Un approccio molto diffuso è l'utilizzo di librerie come **FluentValidation**, che permettono di definire le regole di validazione in **classi separate**, mantenendo il codice più **leggibile e manutenibile**. In questo caso, per ogni DTO di request è possibile creare un **validator specifico** che descrive in modo dichiarativo i **vincoli sui singoli campi**, come l'**obbligatorietà** di alcune proprietà, la **validità dei valori numerici** o il **rispetto di formati specifici**.

Questo tipo di validazione consente anche di esprimere **regole condizionali o più complesse**, ad esempio verificando un campo solo se un altro è presente, oppure applicando **controlli personalizzati** tramite metodi dedicati. Le regole vengono eseguite **automaticamente dal framework** subito dopo il model binding, senza richiedere **interventi espliciti nei controller**. Se la validazione fallisce, il **ModelState** viene popolato con gli errori riscontrati e la richiesta può essere **interrotta** restituendo una **risposta di errore al client**, evitando così l'esecuzione della logica dell'action.

Per una migliore organizzazione del codice, questo approccio permette di mantenere i **controller semplici e focalizzati** sulla gestione del flusso della richiesta, mentre tutta la **logica di validazione viene centralizzata** in componenti separati. Inoltre, la validazione può essere estesa per includere **controlli più avanzati legati al dominio o allo stato del sistema**, contribuendo a rendere l'applicazione più **robusta e coerente** dal punto di vista dei dati gestiti.

Web API protetta con JWT staccato da un IDP:

Partiamo dal presupposto di voler realizzare una applicazione **Web API** che esponga delle risorse tramite endpoint HTTP e che non si occupi direttamente dell'autenticazione degli utenti, ma deleghi tale responsabilità a un **Identity Provider esterno**.

La Web API viene quindi configurata come **resource server**, mentre l'IDP ha il compito di autenticare l'utente e generare un **token JWT** firmato contenente le informazioni necessarie sotto forma di claims. Il client, una volta ottenuto il token dall'IDP, lo invia alla Web API all'interno dell'header `Authorization` utilizzando lo schema Bearer.

Dal punto di vista della Web API è necessario configurare il sistema di autenticazione utilizzando JWT Bearer, andando a specificare i parametri utili alla validazione del token, come la chiave di firma o la chiave pubblica dell'IDP, l'issuer, l'audience e la validità temporale del token. Una volta configurato il servizio di autenticazione, questo viene inserito nella pipeline delle richieste HTTP tramite i middleware di autenticazione e autorizzazione.

A questo punto gli endpoint dell'applicazione possono essere protetti tramite attributi di autorizzazione, permettendo l'accesso alle risorse solamente alle richieste che presentano un token valido. In questo modo l'applicazione Web API rimane disaccoppiata dalla gestione delle

credenziali e si occupa esclusivamente della validazione del token e dell'autorizzazione dell'utente.

Struttura di una applicazione Web API:

Partiamo dal presupposto di voler realizzare una applicazione **Web API** strutturata in modo tale da separare chiaramente i diversi ruoli delle componenti coinvolte, così da ottenere un codice più manutenibile e facilmente estendibile.

Nel **Core** dell'applicazione vengono definite le **entità di dominio**, che rappresentano i concetti principali del problema che si vuole modellare. Le entità contengono lo stato e, eventualmente, la logica di dominio, e non devono dipendere da framework o tecnologie esterne. Sempre nel Core vengono definite anche le **interfacce**, ad esempio quelle dei repository, che descrivono quali operazioni possono essere eseguite sulle entità senza specificarne l'implementazione concreta.

La parte di **Infrastructure** ha invece il compito di fornire le implementazioni concrete delle interfacce definite nel Core. In questo livello troviamo quindi i repository che utilizzano Entity Framework per la persistenza dei dati, il DbContext, le configurazioni delle entità e, in generale, tutte le classi che permettono l'accesso al database o ad altri servizi esterni. Questo livello dipende dal Core, ma ne rispetta le astrazioni.

Successivamente troviamo il livello di **Application** nel quale vengono definiti i modelli di DTO, modelli di richiesta e risposta, che spiegherò in seguito in quanto saranno utilizzati nel successivo livello.
La parte fondamentale del livello Application consiste però nella definizione dei **Services**, ovvero il ponte di collegamento tra le repository (che ricordiamo gestiscono la persistenza) e gli endpoint (che gestiremo nel prossimo livello), andando ad implementare tutta la **logica di business** necessaria alla gestione dell'applicativo.
Per comodità si potrebbero inserire in questo livello le astrazioni inerenti ai livelli precedenti, in modo da averle tutte in un unico posto.

Il livello **Web API** rappresenta il punto di ingresso dell'applicazione ed è responsabile della gestione delle richieste HTTP. Al suo interno troviamo i **controller**, che hanno il compito di esporre gli endpoint, ricevere le richieste, delegare l'esecuzione della logica ai livelli sottostanti e restituire una risposta al client. I controller non devono contenere logica di business, ma limitarsi a coordinare il flusso della richiesta.

Infine, per la comunicazione tra client e Web API vengono utilizzati i **DTO**, che rappresentano i modelli di input e output delle richieste, aiutati dal processo di **model binding**. I DTO permettono di disaccoppiare il formato dei dati esposti all'esterno dalle entità di dominio, evitando di esporre direttamente le entità e consentendo di controllare in modo più preciso quali informazioni vengono scambiate.

All'interno della Web API viene inoltre configurata la **pipeline dei middleware**, che si occupa di aspetti trasversali come autenticazione, autorizzazione, gestione degli errori e logging. La Web API utilizza la **dependency injection** per ottenere le dipendenze necessarie, come i repository o i servizi applicativi, senza conoscere i dettagli dell'implementazione.

Questa organizzazione consente di separare chiaramente le responsabilità tra le varie parti in gioco e di realizzare una applicazione Web API coerente, scalabile e facilmente manutenibile.

Token JWT: composizione, parti e claim:

Partiamo dal presupposto di voler utilizzare un **token JWT** come meccanismo per trasportare informazioni di autenticazione e autorizzazione tra un client e una applicazione protetta. Un JWT è un token compatto e auto-contenuto, rappresentato come una stringa in base 64 composta da **tre parti distinte**, separate da un punto.

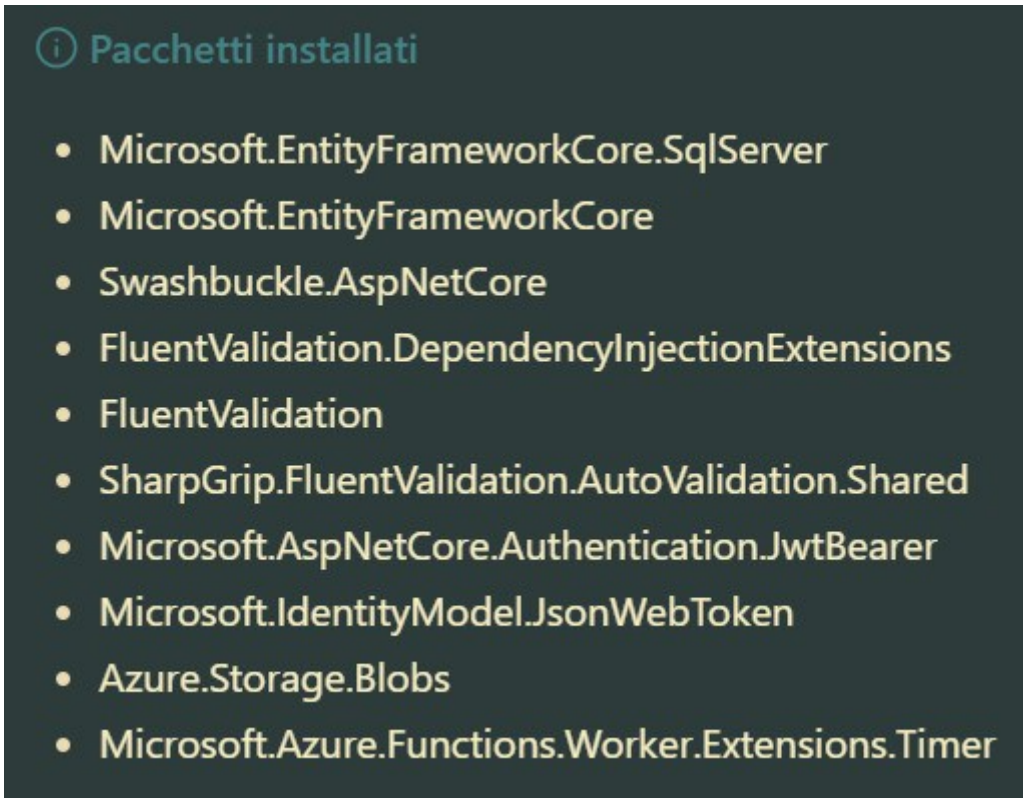
La prima parte è l'**header**, che contiene le informazioni relative al tipo di token e all'algoritmo utilizzato per la firma. In genere vengono specificati il tipo di token, che è JWT, e l'algoritmo di firma utilizzato, ad esempio HMAC o RSA. Questa parte serve alla risorsa che riceve il token per capire come deve essere verificata la firma.

La seconda parte è il **payload**, che contiene le informazioni vere e proprie trasportate dal token sotto forma di **claim**. Le claim rappresentano affermazioni relative all'utente o al contesto di utilizzo del token. Tra le claim standard troviamo ad esempio l'identificativo dell'utente, il soggetto del token, l'issuer che ha emesso il token, l'audience a cui il token è destinato e la data di scadenza. Oltre alle claim standard è possibile definire anche claim personalizzate, come ruoli o permessi applicativi, che vengono utilizzate dall'applicazione per gestire l'autorizzazione.

La terza parte è la **signature**, che viene calcolata a partire dall'header e dal payload utilizzando una chiave segreta o una chiave privata. La firma garantisce l'integrità del token e permette alla Web API di verificare che il contenuto non sia stato alterato e che il token provenga effettivamente da un issuer attendibile.

Nel complesso il JWT consente di trasportare in modo sicuro e stateless le informazioni necessarie all'autenticazione e all'autorizzazione, evitando la necessità di mantenere uno stato lato server.

Per quanto concerne C#, è possibile definire all'interno dei controller di un'applicazione asp.net l'annotazione [authorize] per indicare che quel path è accedibile solamente autenticandosi ed eventualmente si può anche definire un **ruolo** necessario per accedervi.



OAuth2: flussi di autorizzazione

OAuth2 è un protocollo di autorizzazione che permette a un'applicazione di accedere a risorse protette senza dover gestire direttamente le credenziali dell'utente. Il meccanismo si basa sull'uso di token di accesso rilasciati da un Authorization Server, che stabiliscono cosa un client può fare e per quanto tempo. Per adattarsi a contesti diversi, OAuth2 definisce vari flussi, ciascuno pensato per uno specifico tipo di applicazione e per diversi requisiti di sicurezza.

Uno dei flussi più utilizzati è l'**Authorization Code Flow**. In questo caso il client non gestisce direttamente l'autenticazione dell'utente, ma lo reindirizza verso l'Authorization Server. Qui l'utente si autentica e fornisce il consenso all'accesso alle risorse richieste. Al termine di questa fase, l'Authorization Server non restituisce subito un token, ma un codice di autorizzazione. Questo codice viene poi utilizzato dal client, tramite una comunicazione diretta e sicura con l'Authorization Server, per ottenere l'access token. In questo modo il token non viene mai esposto al browser o all'utente, rendendo il flusso particolarmente adatto alle applicazioni server-side, dove la sicurezza è un aspetto fondamentale.

Il **Client Credentials Flow** viene invece utilizzato in scenari in cui non è presente un utente finale. In questo caso è il client stesso a rappresentare l'identità che richiede l'accesso alle risorse. Il client si autentica presso l'Authorization Server utilizzando le proprie credenziali e ottiene un access token che gli consente di accedere alle API o ai servizi protetti. Questo flusso è tipico delle comunicazioni server-to-server, ad esempio quando un servizio backend deve interagire con un altro servizio in modo automatico, senza alcun intervento umano.

In generale, ogni flusso OAuth2 risponde a esigenze diverse e viene scelto in base al contesto di utilizzo, alla tipologia dell'applicazione e ai requisiti di sicurezza. L'obiettivo comune è sempre quello di garantire un accesso controllato alle risorse, limitando i privilegi concessi e riducendo l'esposizione delle informazioni sensibili.

In un'applicazione ASP.NET, OAuth2 viene integrato configurando il sistema di autenticazione affinché l'API accetti e validi access token, tipicamente in formato JWT. L'applicazione non gestisce direttamente il processo di login, ma si affida a uno o più Authorization Server esterni per il rilascio dei token. ASP.NET si occupa di verificarne la validità controllando elementi come issuer, audience e firma, e utilizza le informazioni contenute nei claim per applicare regole di autorizzazione basate su policy. In questo modo è possibile supportare scenari diversi, come l'accesso tramite provider esterni o tramite token personalizzati, mantenendo una gestione della sicurezza centralizzata, modulare e coerente con i principi di OAuth2.

Pillole di Azure

La parte Azure del progetto introduce l'utilizzo di servizi cloud per gestire in modo più flessibile e scalabile alcune funzionalità dell'applicazione. In particolare viene utilizzato **Azure Blob Storage** come sistema di archiviazione per dati

Accanto allo storage dei file viene introdotto **Azure Queue Storage**, che ha il ruolo di gestire la comunicazione asincrona tra componenti dell'applicazione. La Web API può inserire messaggi all'interno di una coda quando deve delegare un'operazione a un processo separato, evitando di eseguire operazioni lunghe o bloccanti durante la gestione della richiesta HTTP. In questo modo si ottiene un maggiore disaccoppiamento tra chi produce il messaggio e chi lo consuma.

Le **Azure Functions** vengono utilizzate come componenti serverless che reagiscono agli eventi generati dall'applicazione, in particolare ai messaggi presenti nelle code. Le Functions permettono di eseguire logica applicativa in modo indipendente dalla Web API, scalando automaticamente in base al carico e senza la necessità di gestire direttamente l'infrastruttura. Questo consente di spostare fuori dalla Web API l'elaborazione asincrona, rendendo l'architettura complessivamente più scalabile e manutenibile.

Altri Consigli per l'esame

Nella gestione delle richieste API da parte dei controller, controllare la presenza dell'annotazione **[ApiController]** che permette di automatizzare i controlli di model binding all'interno dei metodi che altrimenti dovrebbero essere fatti con l'annotazione [FromBody] sui parametri dei metodi ed usando model.isValid per controllare prima di poter stabilire che tipo di risposta ritornare (se ok, oppure BadRequest). Inoltre la validazione automatica (usando ad esempio FluentValidation) non può aver luogo senza la presenza di di ApiController.

Nel file **appsettings.json** è possibile andare a definire nuove sezioni per memorizzare delle impostazioni di configurazione come ad esempio registrare la stringa di connessione al db e poi richiamarla usando il metodo **builder.configuration.GetConnectionString("nomestringa").** Altro utilizzo potrebbe essere quello di andare a definire informazioni riguardo l'uso di oauth per l'autenticazione, quali ClientID, TenantID, Authority ecc.

È importante ricordare che siccome il protocollo HTTP è stateless, per mantenere una sessione ed un login stateless, viene incluso all'interno dell'header HTTP il campo Authorization: Bearer <JWT token>.

Breve lista dei messaggi HTTP utili:

- 200 Ok response
- 201 Created
- 400 Bad Request
- 401 Unauthorized (bisogna autenticarsi)
- 403 Forbidden (autenticato ma non autorizzato)
- 404 Not Found
- 405 Method not Allowed
- 500 Internal Server Error
- 501 Not Implemented
- 502 Bad Gateway

Usare MSTest per creare un progetto dedicato al testing automatizzato.

```
namespace Unicam.Libreria.Tests
{
    [TestClass]
    0 riferimenti
    public class ValidatorsTest
    {
        [TestMethod]
        0 riferimenti
        public void AddLibroRequestValidator_Is_Ok()
        {
            //ARRANGE
            var validator = new AddLibroRequestValidator();
            var request = new AddLibroRequest();
            request.ISBN = "1234567890";
            request.IdAutore = 1;
            request.Nome = "Titolo";

            //ACT
            var result = validator.Validate(request);

            //ASSERT
            Assert.IsTrue(result.IsValid);
        }
    }
```