Definizione di REST API: principi fondamentali e caratteristiche.

Un'API è un insieme di regole che definisce il modo in cui i programmi possono connettersi e comunicare tra loro. Come si intuisce dal nome, un’API REST trasferisce su ogni richiesta lo stato di una transazione. Questo approccio presenta vantaggi a livello di progettazione, performance e risorse rispetto agli altri. L’architettura REST è stata sviluppata per più di vent’anni ed è molto diffusa nelle architetture basate sui servizi e distribuite.

Significato di REST

REST è l'acronimo di "Representational State Transfer". È uno stile architettonico per la creazione di servizi web che forniscono un modo standardizzato di comunicazione su Internet tra sistemi informatici diversi. REST si basa su un insieme di principi di progettazione, piuttosto che su una tecnologia o un protocollo specifici, ed è ampiamente utilizzato nella creazione di API basate sul Web.

Una delle caratteristiche principali di REST è l'uso di metodi HTTP come [GET, POST, PUT e DELETE](https://medium.com/@9cv9official/what-are-get-post-put-patch-delete-a-walkthrough-with-javascripts-fetch-api-17be31755d28) per rappresentare diverse operazioni che possono essere eseguite sulle risorse. Queste risorse possono essere qualsiasi tipo di dato o funzionalità esposta dal servizio web e sono tipicamente rappresentate utilizzando un formato di dati standardizzato, solitamente JSON.

I sei principi di progettazione REST

Le REST API si basano su sei regole fondamentali che ne definiscono il comportamento e la struttura. Il primo principio è l’**interfaccia uniforme**: tutte le richieste devono seguire uno schema coerente, con risorse identificate da URI e metodi HTTP standard. Questo rende le API prevedibili e facili da usare.

Il secondo principio è la **separazione tra client e server**. Il client non deve conoscere la logica interna del server, ma solo come accedere alle risorse. Questo permette di sviluppare frontend e backend in modo indipendente.

Il terzo è la **stateless**, cioè ogni richiesta è autonoma e non dipende da richieste precedenti. Il server non conserva lo stato del client, semplificando la gestione e migliorando la scalabilità.

Il quarto principio riguarda la **cache**: le risposte possono essere memorizzate per migliorare le prestazioni. Il server deve indicare se una risorsa può essere messa in cache e per quanto tempo.

Il quinto è l’**architettura a livelli**, che consente di inserire intermediari tra client e server (come proxy o bilanciatori) senza che il client ne sia consapevole. Questo aumenta la flessibilità e la sicurezza.

Infine, il sesto principio è il **codice su richiesta** (*code on demand*), opzionale. Il server può inviare codice eseguibile al client, come script JavaScript, per estendere le funzionalità.

Confronto tra REST e altre architetture (SOAP, GraphQL).

Le API REST sono stateless, cache e facili da documentare. Sono ideali per applicazioni web moderne, perché permettono una comunicazione chiara tra client e server, con una curva di apprendimento relativamente bassa. Inoltre, supportano diversi formati di dati, come JSON e XML, e si integrano bene con strumenti come Swagger/OpenAPI.

**SOAP**, invece, è un protocollo più rigido e formale, basato esclusivamente su XML. Utilizza il WSDL (Web Services Description Language) per definire in modo preciso le operazioni disponibili. SOAP è molto usato in ambiti enterprise, soprattutto quando servono transazioni complesse e un alto livello di sicurezza, come nei sistemi bancari. Offre funzionalità avanzate come la gestione degli errori e la cifratura dei messaggi, ma ha una curva di apprendimento più alta e richiede più risorse, sia in termini di banda che di sviluppo.

**GraphQL** è una tecnologia più recente, pensata per superare alcuni limiti di REST. A differenza di REST, dove spesso servono più chiamate per ottenere dati da entità correlate, GraphQL permette di fare una singola richiesta e ricevere esattamente i dati desiderati. Questo lo rende molto efficiente, soprattutto per applicazioni mobile o frontend dinamici. Tuttavia, può essere più complesso da implementare, soprattutto per quanto riguarda la cache e la sicurezza. Inoltre, non sempre è la scelta migliore per query molto complesse o per sistemi che devono garantire alte performance.

Metodi HTTP principali: GET, POST, PUT, DELETE (con esempi d’uso)

Questi metodi permettono al client di leggere una risorsa, crearla, cancellarla o modificarla e sono:

**GET:**è il Metodo più comune nelle richieste API. Serve a richiedere al server una vista dei dati.

Esempio

app.MapGet("/api/studenti", () => Results.Ok(studenti))

.WithName("GetStudenti")

.WithOpenApi(op => new(op)

Questo codice ottiene tutti gli studenti e restituisce la lista completa degli studenti con i corsi a cui sono iscritti.

Autenticazione e sicurezza (API Key, JWT – anche solo a livello introduttivo).

Per proteggere le API e controllare chi può accedere alle risorse, si usano meccanismi di autenticazione. I due più comuni sono le **API Key** e i **JWT (JSON Web Token)**.

Le **API Key** sono stringhe uniche che identificano un’applicazione o un utente. Vengono inviate insieme alla richiesta e il server le verifica. Sono semplici da implementare e utili per applicazioni che non gestiscono utenti, ma non sono molto sicure se non si usa HTTPS.

I **JWT**, invece, sono token firmati digitalmente che contengono informazioni sull’utente (come ID e permessi). Dopo il login, il server genera un JWT e lo invia al client, che lo usa per autenticarsi nelle richieste successive. I JWT sono più sicuri e flessibili, e si adattano bene alle API RESTful moderne perché non richiedono sessioni sul server.

**POST**è un metodo utilizzato per chiedere al server di generare una nuova risorsa. Il client indica al server tutti i dati della nuova risorsa.

app.MapPost("/api/studenti", (Studente nuovoStudente) =>

{

nuovoStudente.Id = studenti.Count > 0 ? studenti.Max(s => s.Id) + 1 : 1;

studenti.Add(nuovoStudente);

return Results.Created($"/api/studenti/{nuovoStudente.Id}", nuovoStudente);

})

Questo codice crea un nuovo studente e lo aggiunge alla lista.

**PUT**è il metodo utilizzato per aggiornare una risorsa sul server. Il client deve indicare la risorsa da aggiornare e invia i dati aggiornati nel body della richiesta. Se la risorsa non esiste, viene creata.

app.MapPut("/api/studenti/{id}", (int id, Studente studenteAggiornato) =>

{

var studenteEsistente = studenti.FirstOrDefault(s => s.Id == id);

if (studenteEsistente is null)

return Results.NotFound();

studenteEsistente.Nome = studenteAggiornato.Nome;

studenteEsistente.Cognome = studenteAggiornato.Cognome;

return Results.NoContent();

})

.WithName("UpdateStudente")

.WithOpenApi(op => new(op)

Questo codice aggiorna uno studente modificandone nome e cognome

**DELETE**è il metodo utilizzato per ordinare al server di cancellare una determinata risorsa

app.MapDelete("/api/studenti/{id}", (int id) =>

{

var studenteDaRimuovere = studenti.FirstOrDefault(s => s.Id == id);

if (studenteDaRimuovere is null)

return Results.NotFound();

studenti.Remove(studenteDaRimuovere);

return Results.Ok();

})  
Questo codice elimina uno studente e lo rimuove dalla lista

Codici di stato HTTP più comuni

**200 OK** indica che tutto è andato a buon fine. Questo è il codice che ci si aspetta quando si fa una POST o una GET correttamente.

**201 CREATED** viene restituito quando una risorsa è andata a buon fine ed è stato creato un qualcosa. (POST)

**204 NO CONTENT** la richiesta è andata a buon fine ma il server non restituisce alcun contenuto. È utile per operazion come il DELATE.

**400 BAD REQUEST** significa che la richiesta contiene dati non validi. Può uscire se mancano dei dati obbligatori o se è sbagliato il formato del JSON.

**401 UNAUTHORIZED** il client non è autorizzato ad accedere alla risorsa. Di solito serve un token.

**404 NOT FOUND** richiesta non esistente.

**500 INTERNAL SERVER ERROR** qualcosa è andato storto nel server come ad esempio un errore nel codice, nel database o nel middleware.

**Descrizione del progetto sviluppato**

Il progetto consiste nella realizzazione di una **Web API RESTful** utilizzando **.NET 8 Minimal API**, con l’obiettivo di simulare un sistema di gestione studenti e corsi.

**Obiettivi principali**

* Implementazione **operazioni CRUD complete** per due entità correlate: Studente e Corso.
* Gestione della **relazione molti-a-molti** tra studenti e corsi.
* Fornire un **endpoint di ricerca** per ottenere tutti i corsi a cui è iscritto uno studente.
* Integrazione **Swagger** per la documentazione interattiva.
* Aggiungere un **middleware personalizzato** per la gestione dell’accesso tramite token.
* Applicare una **policy CORS** per rendere l’API accessibile da frontend esterni.

**Struttura del progetto**

* **Entità principali**:
  + Studente: contiene Id, Nome, Cognome, e una lista di CorsiIscritti.
  + Corso: contiene Id, NomeCorso, Descrizione.
* **Endpoint implementati**:
  + GET /api/studenti – Ottiene tutti gli studenti.
  + GET /api/studenti/{id} – Ottiene uno studente per ID.
  + POST /api/studenti – Crea un nuovo studente.
  + PUT /api/studenti/{id} – Aggiorna nome e cognome.
  + DELETE /api/studenti/{id} – Elimina uno studente.
  + GET /api/studenti/{studenteId}/corsi – Ottiene i corsi di uno studente.
  + GET /api/corsi – Ottiene tutti i corsi.
  + GET /api/corsi/{id} – Ottiene un corso per ID.
  + POST /api/corsi – Crea un nuovo corso.
  + PUT /api/corsi/{id} – Aggiorna nome e descrizione.
  + DELETE /api/corsi/{id} – Elimina un corso e aggiorna gli studenti iscritti.
  + POST /api/login – Effettua il login e restituisce un token.

**Sicurezza e Middleware**

* È stato implementato un **middleware personalizzato** che controlla la presenza e validità dell’header Access in ogni richiesta.
* Solo l’endpoint /api/login è esente da controllo.
* Il token AB12345 viene restituito in caso di login corretto e deve essere usato per accedere agli altri endpoint.

**Documentazione Swagger**

* Swagger UI è abilitato tramite UseSwagger() e UseSwaggerUI().
* Ogni endpoint è documentato con .WithOpenApi() che include:
  + Summary e Description
  + Tipi di risposta (Produces)
  + Tipo di input (Accepts)

L’interfaccia è accessibile da: **http://localhost:7112/swagger/index.html**

Diagramma UML

Il progetto include un diagramma UML delle entità, che rappresenta:

* Le classi Studente e Corso
* La relazione molti-a-molti
* I tipi di attributi e le cardinalità

**Test e simulazioni**

* Gli endpoint sono testabili con Postman.
* Le chiamate devono includere l’header Access: AB12345 per superare il middleware di sicurezza.

Documentazione degli endpoint con esempi di chiamata

Immagine che contiene schermata, testo, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene schermata, testo, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.Immagine che contiene schermata, software, testo, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.Immagine che contiene schermata, testo, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.Immagine che contiene schermata, testo, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.Immagine che contiene schermata, testo, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.Immagine che contiene software, Software multimediale, testo, Software per la grafica

Il contenuto generato dall'IA potrebbe non essere corretto.Immagine che contiene schermata, testo, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.Immagine che contiene schermata, testo, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.Immagine che contiene schermata, software, Software multimediale, testo

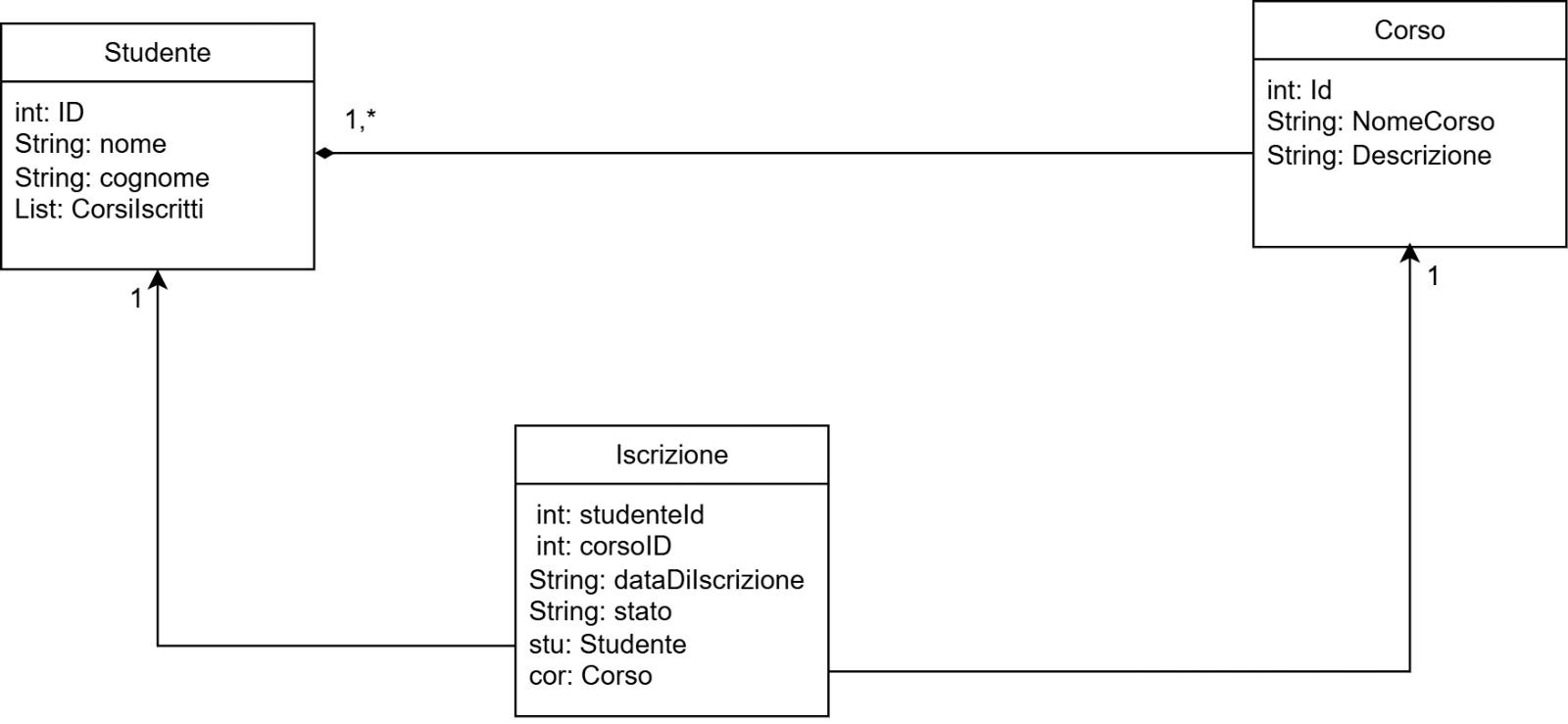
Il contenuto generato dall'IA potrebbe non essere corretto.Immagine che contiene schermata, testo, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.Immagine che contiene schermata, testo, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.Immagine che contiene schermata, software, Software multimediale, testo

Il contenuto generato dall'IA potrebbe non essere corretto.

Diagramma UML



Link delle repository di github

[rzavanelli-blip/Esame-finale-.NET-C-e-REST-API](https://github.com/rzavanelli-blip/Esame-finale-.NET-C-e-REST-API)