To Do List

Video Tutorial

Introduzione

Questo progetto è un'applicazione web che simula una **To-Do List**, cioè una lista di cose da fare. È pensata per essere **facile da usare**, così chiunque può organizzare le proprie attività senza difficoltà. Allo stesso tempo, è costruita in modo **ordinato e completo**, in modo da garantire un funzionamento **stabile e affidabile**. L'app permette di **aggiungere nuovi compiti**, tenerli sotto controllo, segnare quali sono stati **completati** e rimuovere quelli già svolti. Questo aiuta l'utente a gestire meglio il proprio tempo e a non dimenticare gli impegni importanti.

L'obiettivo è aiutare l'utente a **organizzare meglio le proprie attività quotidiane**, permettendo di aggiungere nuovi compiti, visualizzarli, segnarli come completati oppure eliminarli con facilità.

Per realizzarla sono stati usati diversi strumenti. La parte grafica, cioè ciò che l'utente vede e usa, è stata creata con **HTML, CSS e JavaScript**, per renderla chiara e semplice da navigare. La parte che gestisce il funzionamento dell'app è stata sviluppata con il linguaggio **C# usando .NET Core**, che permette di creare delle **API** per comunicare tra le varie parti dell'app.

Infine, i dati (cioè le attività) vengono salvati in un **database chiamato SQL Server**, che si occupa di conservarli in modo organizzato.

Funzionalità Dell'app

- **Aggiunta di task**
  L'utente può creare nuove attività inserendo vari dettagli importanti come il **titolo**, una **descrizione** che spiega meglio il compito, la **data e l'ora di scadenza**, la **categoria di appartenenza** e l'**utente assegnato**. Questo permette di organizzare le attività in modo molto preciso, rendendo semplice tenere traccia di ogni compito e delle sue caratteristiche principali.
- **Visualizzazione delle task**
  Le attività possono essere filtrate e visualizzate in base a criteri diversi: per **categoria**, per **utente** oppure combinando entrambe le opzioni. Questa funzionalità consente all'utente di trovare rapidamente ciò che serve, organizzando la lista in modo che mostri solo le attività più rilevanti al momento, facilitando la gestione quotidiana.
- **Riepilogo task e filtri attivi**
  Nella parte superiore dell'interfaccia, all'interno della **navbar**, è possibile vedere il numero totale delle attività presenti e il conteggio degli **utenti** o delle **categorie selezionate** nei filtri. Questa funzione offre una panoramica immediata dello stato delle attività e delle impostazioni correnti, aiutando l'utente a capire rapidamente quali elementi sta visualizzando.

- **Gestione utenti e categorie**
  L'app permette di **aggiungere o eliminare utenti e categorie** direttamente dalla navbar, offrendo una gestione dinamica e flessibile dell'organizzazione delle attività. In questo modo è possibile aggiornare facilmente chi partecipa alla lista e quali categorie sono utilizzate, mantenendo l'app sempre personalizzata in base alle proprie esigenze.
- **Scelta del tema grafico**
  L'utente ha la possibilità di scegliere tra un **tema chiaro** e uno **scuro** per l'interfaccia, così da adattare l'aspetto dell'app al proprio gusto personale o alle condizioni di luce ambientale. Questa funzionalità migliora l'esperienza d'uso, permettendo una lettura più comoda e un design più piacevole.
- **Pagina delle task completate**
  È presente una seconda pagina raggiungibile dalla navbar, dedicata esclusivamente alle attività già **completate**. Questa separazione permette di tenere distinta la lista delle cose da fare da quelle concluse, facilitando l'organizzazione e dando la possibilità di consultare facilmente i compiti portati a termine.
- **Modifica, eliminazione e visualizzazione dettagli task**
  Una volta creata, ogni attività può essere **modificata** per aggiornare informazioni o correggere dettagli, **eliminata** se non più necessaria o **visualizzata in dettaglio** per avere tutte le informazioni a portata di mano. Questo garantisce un controllo completo sulle attività e ne facilita la gestione anche in caso di cambiamenti.
- **Gestione sotto-task**
  Ogni attività può avere delle sotto-attività collegate, chiamate **sotto-task**, che aiutano a suddividere compiti complessi in parti più piccole e gestibili. Questa funzione è utile per organizzare meglio le attività articolate, consentendo all'utente di monitorare ogni singolo passaggio necessario per completare il lavoro.
- **Segnare task come completate**
  Le attività possono essere contrassegnate come **completate** tramite un semplice click, spuntando la casella dedicata. Questo permette di tenere traccia di ciò che è stato fatto e di spostare automaticamente le attività concluse nella pagina apposita, mantenendo la lista delle cose da fare sempre aggiornata.
- **Pulsante informazioni**
  Un pulsante **"info"** è disponibile per fornire all'utente una guida rapida sull'utilizzo delle varie funzionalità dell'app. Questa funzione è particolarmente utile per chi usa l'app per la prima volta o per chi ha bisogno di un supporto veloce senza dover cercare altrove.

Tecnologie Utilizzate

Database – SQL Server

Per la gestione dei dati è stato utilizzato **SQL Server**, un sistema di gestione di basi di dati relazionali (RDBMS) robusto e professionale. Le sue caratteristiche principali nel progetto sono:

- **Gestione relazionale dei dati**, con l'uso di foreign key tra le entità.
- **Stored Procedure** per tutte le operazioni CRUD e la logica di business (es. completamento automatico delle sottotask).
- **Supporto all'integrità referenziale**, che ha permesso di mantenere coerenza tra i dati.
- Interfacciamento semplice con .NET tramite le librerie `System.Data.SqlClient`.

Strumenti utilizzati:

- **SQL Server Management Studio (SSMS)** per la progettazione, scrittura ed esecuzione di query e stored procedure.

Backend/API – ASP.NET Core Web API (C#)

Per la creazione delle API è stato utilizzato **ASP.NET Core**, un framework multipiattaforma e ad alte prestazioni per lo sviluppo di applicazioni web.

Caratteristiche principali:

- Struttura a **controller** separati per ogni entità (Task, SottoTask, Utente, Categoria).
- Utilizzo di **DTO (Data Transfer Object)** per il passaggio dei dati da/verso il frontend.
- Accesso al database tramite **stored procedure** per sicurezza e prestazioni.
- **Routing RESTful** per ogni endpoint (`GET`, `POST`, `PUT`, `DELETE`).
- **Configurazione CORS** per permettere chiamate da frontend esterni.

Strumenti utilizzati:

- **Visual Studio / Visual Studio Code** per lo sviluppo e il debugging.
- **Postman** per testare manualmente le API.

Frontend – HTML, CSS, JavaScript

Per la realizzazione dell'interfaccia utente è stato utilizzato un insieme di tecnologie web standard: **HTML, CSS e JavaScript**. Questi linguaggi permettono di creare pagine web interattive, responsive e facili da usare.

Caratteristiche principali:

- **HTML** definisce la struttura e i contenuti della pagina, organizzando elementi come bottoni, form, liste e sezioni.
- **CSS** si occupa dello stile grafico, gestendo colori, layout, spaziature e la personalizzazione del tema chiaro/scuro.
- **JavaScript** aggiunge interattività, come la gestione degli eventi (clic sui pulsanti, selezione filtri), la comunicazione con il backend tramite richieste HTTP (fetch/AJAX) e l'aggiornamento dinamico dei dati mostrati.

L'interfaccia è stata progettata per essere semplice e intuitiva, con una barra di navigazione che permette di accedere rapidamente alle funzioni principali, e una gestione fluida della visualizzazione delle task.

Strumenti utilizzati:

- Editor di codice (Visual Studio Code) per scrivere e organizzare i file HTML, CSS e JS.
- Browser moderni (Chrome, Firefox) per testare e fare il debug dell'interfaccia.
- Strumenti di sviluppo integrati nel browser per analizzare l'andamento del codice e risolvere problemi di rendering o comportamento.

Progettazione

Struttura delle tabelle del Database

Il database dell'applicazione è stato progettato in **SQL Server** seguendo una struttura relazionale, per garantire l'integrità dei dati.

Tabelle principali:

1. **Utente**

- `IdUtente` (INT, PK, IDENTITY): identificatore univoco.

- `Nome` (VARCHAR(50)): nome dell'utente.

2. **Categoria**

- `IdCategoria` (INT, PK, IDENTITY): identificatore univoco.
- `Descrizione` (VARCHAR(100)): descrizione della categoria.

3. **Task**

- `IdTask` (INT, PK, IDENTITY): identificatore univoco.
- `Titolo` (VARCHAR(100)): titolo del task.
- `Descrizione` (VARCHAR(MAX)): dettagli della task.
- `Completata` (BIT): stato di completamento (true/false).
- `Scadenza` (DATETIME): data di scadenza.
- `IdUtente` (FK): riferimento all'utente.
- `IdCategoria` (FK): riferimento alla categoria.

4. **SottoTask**

- `IdSottoTask` (INT, PK, IDENTITY): identificatore univoco.
- `Titolo` (VARCHAR(100)): titolo del sottotask.
- `Completata` (BIT): stato del sottotask.
- `IdTask` (FK): riferimento alla task principale.

Relazioni

- Ogni utente può avere più task.
- Ogni categoria può essere collegata a più task.
- Ogni task può contenere più sottotask.

- Le foreign key garantiscono l'integrità referenziale.

- In caso di eliminazione di una task, le relative sottotask possono essere eliminate automaticamente tramite `ON DELETE CASCADE`.

Struttura delle API

Le **API RESTful** sono state sviluppate in **ASP.NET Core Web API** e rappresentano il livello intermedio dell'applicazione. Consentono al frontend di comunicare con il database, esponendo operazioni CRUD su tutte le entità principali.

Controller principali:

- **TaskController**
    - `GET /api/task` → tutte le task.
    - `GET /api/task/utente/{id}` → task di uno specifico utente.
    - `GET /api/task/utente/{idUtente}/categoria/{idCategoria}` → task filtrate.
    - `POST /api/task` → crea una nuova task.
    - `PUT /api/task/{id}` → modifica task esistente.
    - `DELETE /api/task/{id}` → elimina una task.
    - `POST /api/task/completa/{id}` → completa la task e le sue sottotask.
- **SottoTaskController**
    - `POST /api/sottotask` → crea una sottotask.
    - `PUT /api/sottotask/completa/{id}` → segna una sottotask come completata.
    - `DELETE /api/sottotask/{id}` → elimina una sottotask.
- **CategoriaController**
    - `GET /api/categoria` → tutte le categorie.
    - `POST /api/categoria` → crea una nuova categoria.
    - `DELETE /api/categoria/{id}` → elimina una categoria

- **UtenteController**
    - `GET /api/utente` → tutti gli utenti.
    - `POST /api/utente` → crea un nuovo utente.
    - `DELETE /api/utente/{id}` → elimina una categoria

Altri aspetti tecnici

- Utilizzo di **DTO (Data Transfer Object)** per separare i dati dal modello di dominio.
- Validazione dei dati in ingresso con `ModelState.IsValid`.
- Abilitazione di **CORS** per consentire le richieste da frontend:

```
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAll",
                                                    policy      =>
policy.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader());
});
```

Logica del codice

- Le API si collegano al database usando stored procedure, che gestiscono tutte le operazioni CRUD.
- Ogni richiesta del frontend viene ricevuta da un controller, che chiama la stored procedure corrispondente passando i parametri necessari.
- Esempio:

```
var sql = "EXEC AggiungiTask @Titolo, @Descrizione,
@Completata, @Scadenza, @IdUtente, @IdCategoria";
_context.Database.ExecuteSqlRaw(sql, param1, param2, ...);
```

- Se una task viene completata, un'apposita procedura completa automaticamente anche le sue sottotask.
- La logica lato backend è completamente separata dalla gestione della presentazione (frontend) e dell'archiviazione dati (database), secondo il modello a 3 livelli.

Implementazione

Progettazione e creazione del database SQL Server

Il primo passo è stato creare il database relazionale, con le tabelle: `Utente`, `Categoria`, `Task`, e `SottoTask`, tutte collegate da relazioni 1:N. La progettazione è stata seguita dalla scrittura delle stored procedure necessarie per:

- Inserimento e modifica di task e sottotask.
- Eliminazione controllata dei record.
- Completamento automatico delle sottotask quando una task viene completata.
- Filtro dei task per utente e categoria.

Tutte le procedure sono state testate tramite SSMS prima dell'integrazione nel backend.

Sviluppo delle API per la gestione delle attività

Il secondo step è stato lo sviluppo delle API. Per ogni entità è stato realizzato un **controller** che espone gli endpoint REST. Le chiamate del frontend (JavaScript) vengono gestite tramite:

- **SqlConnection** e **SqlCommand** per eseguire le stored procedure.
- Endpoint come `GET`, `POST`, `PUT`, `DELETE` per ogni entità.
- Endpoint dedicati come `POST /api/task/completa/{id}` che completano una task e tutte le sue sottotask.

È stato anche configurato il CORS per accettare richieste dal frontend su domini diversi.

Test e verifica

Durante lo sviluppo, le API sono state testate utilizzando **Swagger**, uno strumento integrato in ASP.NET Core che permette di esplorare, inviare richieste HTTP ed esaminare le risposte direttamente da browser.

Swagger offre:

- Interfaccia grafica automatica basata sugli endpoint definiti nei controller.
- Invio di richieste `GET`, `POST`, `PUT`, `DELETE` con parametri e body JSON.
- Visualizzazione immediata dei codici di stato HTTP (es. 200, 400, 404).
- Test diretto della logica applicativa e del collegamento con il database.

La documentazione Swagger è stata generata automaticamente grazie all'inclusione del servizio:

csharp

CopiaModifica

```
builder.Services.AddSwaggerGen();
```

e l'attivazione nel pipeline HTTP:

csharp

CopiaModifica

```
app.UseSwagger();

app.UseSwaggerUI();
```

Costruzione del frontend con HTML/CSS/JavaScript

La fase successiva dello sviluppo ha riguardato la realizzazione dell'interfaccia utente utilizzando HTML, CSS e JavaScript per creare una pagina web chiara, funzionale e responsiva.

Caratteristiche principali:

- **HTML** è stato utilizzato per strutturare gli elementi visivi, come moduli di inserimento task, liste dinamiche, bottoni e barre di navigazione.

- **CSS** ha permesso di definire lo stile grafico, gestendo colori, font, spaziature e layout, oltre a implementare la possibilità di passare da un tema chiaro a uno scuro per migliorare l'esperienza utente in diverse condizioni di luce.

- **JavaScript** ha aggiunto interattività, consentendo la gestione degli eventi (clic, selezioni), la comunicazione asincrona con le API backend tramite fetch e l'aggiornamento in tempo reale della visualizzazione delle task senza ricaricare la pagina.

Il frontend è stato progettato per essere semplice da navigare, con un menu di navigazione intuitivo che permette di accedere velocemente alle diverse sezioni, come la lista delle task attive e quella delle task completate.

Strumenti utilizzati:

- Visual Studio Code per la scrittura e l'organizzazione dei file.

- Browser moderni come Chrome e Firefox per il test e il debug tramite strumenti di sviluppo integrati.

Problemi incontrati (DB/API)

- **Errore CORS**: risolto attivando AllowAnyOrigin() nella policy.

- **Problemi con le foreign key**: inizialmente alcuni inserimenti fallivano a causa di vincoli violati, risolti con controlli lato API.

- **Sincronizzazione Task/SottoTask**: gestita tramite logica SQL in una stored procedure per evitare inconsistenze.

Conclusione

Il progetto della **To-Do List** ha permesso di mettere in pratica i concetti dell'**architettura a 3 livelli**, con particolare attenzione alla progettazione del **database relazionale** e allo sviluppo delle **API RESTful**.

Il **database**, realizzato in SQL Server, è stato progettato per garantire integrità, coerenza e scalabilità. L'utilizzo di **stored procedure** ha reso più sicura ed efficiente la gestione delle operazioni, semplificando anche la logica nel backend.

Le **API**, sviluppate in ASP.NET Core, hanno fornito un'interfaccia chiara e ben strutturata per la comunicazione tra frontend e database. L'impiego di **controller separati**, **DTO**, e la configurazione di **Swagger** hanno facilitato il testing e lo sviluppo modulare.

Durante il progetto sono state affrontate e risolte varie problematiche, come la gestione del CORS, la sincronizzazione tra task e sottotask, e il controllo dell'integrità referenziale. Il risultato finale è stato un'applicazione solida, facilmente estendibile e ben organizzata.

Questa esperienza ha rafforzato le competenze pratiche su:

- Modellazione e normalizzazione di un database,
- Sviluppo di API con architettura REST,
- Integrazione tra livelli separati di un'applicazione moderna