Relazione Tecnica sul Progetto di Danza

Applicazione MVC con API REST

Studenti: **[Vladimir Marco Corda, Simone Grazioli]**

1 settembre 2025

Indice

Capitolo 1

Introduzione

1.1 Obiettivi del Progetto

Il progetto nasce con l'obiettivo di supportare la gestione di eventi legati al mondo della danza, fornendo una piattaforma digitale per:

- registrazione e autenticazione degli utenti;
- creazione, approvazione e visualizzazione di eventi;
- gestione delle partecipazioni e delle recensioni.

1.2 Tecnologie Utilizzate

L'applicazione è sviluppata in **ASP.NET Core MVC** con **Visual Studio**, seguendo l'architettura **Model-View-Controller (MVC)** e si interfaccia con un **server REST** attraverso un set di **API** documentate con **Swagger**.

Capitolo 2

Struttura Tecnica

2.1 Architettura MVC

- **Model**: rappresenta gli oggetti principali (`User`, `Event`, `Participation`).
- **Controller**: gestisce le richieste degli utenti e chiama i metodi API.
- **View**: interfaccia utente che visualizza i dati ricevuti.

2.2 Sicurezza e Autenticazione

- Autenticazione tramite cookie e claims (ruolo `User o Admin`).
- Password gestite in modo sicuro con `BCrypt`.
- Endpoint protetti con `[Authorize]`.
- Controlli per impedire accessi a dati di altri utenti, se non sono loggati o se lo sono ma provano ad accedere a dati di un'altro utente. Nella sezione 3.1 è presente un esempio

Capitolo 3

Interazione Vista-Controller

3.1 Flusso dei Dati

Nell'intero progetto si hanno più viste, file html che posso passare il contenuto dei loro campi ai controller. Attraverso un form che chiama il metodo nel controller.

3.1.1 Esempio di Vista di Registrazione

Listing 3.1: Vista `MyEvent.cshtml`

```
<h2>Registrazione</h2>

<div asp-validation-summary="All" class="text-danger"></
   div>

<form asp-action="Register" method="post">
    <!-- Campi obbligatori -->
    <div class="form-group">
        <label asp-for="Name"></label>
        <input asp-for="Name" class="form-control" />
        <span asp-validation-for="Name" class="text-
           danger"></span>
    </div>

  ...

    <!-- Password -->
    <div class="form-group">
        <label asp-for="Password"></label>
```

```
        <input asp-for="Password" type="password" class=
            "form-control" />
        <span asp-validation-for="Password" class="text-
            danger"></span>
    </div>

    <div class="form-group">
        <label asp-for="ConfermaPassword"></label>
        <input asp-for="ConfermaPassword" type="password
            " class="form-control" />
        <span asp-validation-for="ConfermaPassword"
            class="text-danger"></span>
    </div>

    <button type="submit" class="btn btn-primary">
       Registrati</button>
</form>

<p class="mt-3">
    Hai gi  un account? <a asp-action="Login">Accedi</a
       >
</p>
```

3.2 Gestione dei Modelli

3.2.1 Controller e Modelli

I dati passati vengono inviati al controller, che richiede come input un modello ”standard” ad esempio in questo caso ”User”, visto più avanti.

Listing 3.2: Controller della vista della pagina di registrazione

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(User model)
```

3.2.2 Sottomodelli per Specifiche Esigenze

Spesso vengono usati sotto-modelli di un originale; creati per non creare problemi col database, nel caso in cui non si possa creare un oggetto dai modelli principali senza inizializarne alcuni attributi, per rispettare i vincoli referenziali o perché non sono necessarie tutte quelle informazioni.

Listing 3.3: Modello Modifica utente

```
public class ModifyUser
{
    public int? Id { get; set; }
    public string? Name { get; set; }
    public string? Surname { get; set; }
    public string? Aka { get; set; }
    public string? InstaProfile { get; set; }

    [EmailAddress(ErrorMessage = "Inserisci un'email
       valida")]
    public string? Email { get; set; }

    [DataType(DataType.Password)]
    public string? Password { get; set; }
}
```

3.3 Vista degli Eventi Personali

3.3.1 Struttura della Vista

La vista seguente mostra tutti gli eventi proposti dall'utente autenticato:

Listing 3.4: Vista `MyEvent.cshtml`

```
@model List<Event>
<div class="container py-5">
    <h1 class="mb-4">Eventi Proposti da te</h1>

    @if (TempData["Message"] != null)
    {
        <div class="alert alert-info mb-4">@TempData["
           Message"]</div>
    }

    @if (!Model.Any())
    {
        <div class="alert alert-info">
            <i class="fas fa-info-circle me-2"></i>Non
               hai ancora proposto eventi.
        </div>
    }
    else
```

```
{
    <div class="row">
        @foreach (var evento in Model)
        {
            <div class="col-md-4 mb-4">
                <div class="card event-card @(evento
                    .IsApproved ? "border-success" :
                    "border-warning")">
                    <div class="card-body">
                        <h5 class="card-title">
                            @evento.EventName</h5>
                        <p class="card-text price-
                            tag">
                            @(evento.EventPrice?.
                                ToString("C") ?? "
                                Gratis")
                        </p>
                        <p class="card-text text-
                            muted">
                            <small>
                                <i class="far fa-
                                    calendar-alt me-2
                                    "></i>@evento.
                                    EventDate.
                                    ToString("d")<br>
                                <i class="fas fa-map
                                    -marker-alt me-2"
                                    ></i>@evento.
                                    EventLocation
                            </small>
                        </p>
                        <span class="status-badge
                            badge @(evento.IsApproved
                            ? "bg-success" : "bg-
                            warning")">
                            @(evento.IsApproved ? "
                                Approvato" : "In
                                revisione")
                        </span>
                    </div>
                </div>
            </div>
        }
```

```
        </div>
    }

    <div class="text-center mt-4">
        <a asp-controller="Event" asp-action="CreateForm
           " class="btn btn-primary btn-lg">
            <i class="fas fa-plus-circle me-2"></i>
               Proponi un nuovo evento
        </a>
    </div>
</div>
```

3.3.2 Caratteristiche della Vista

Utilizza **Bootstrap** per il layout e mostra dinamicamente gli eventi recuperati dal controller.

Nelle prime righe possiamo vedere che se ci sono messaggi temporanei allora questi vengono stampati a schermo, inoltre vediamo come passata una lista di elementi, del modello Event, che rappresenta li eventi, possiamo utilizzare li attributi di questi per alterare dinamicamente la pagina.

3.4 Controller ed Interazioni - con esempi

3.4.1 Metodo MyEvent

Il metodo che fornisce i dati alla vista è il seguente:

Listing 3.5: Metodo `MyEvent` del Controller

```
[Authorize(Roles = "User")]
[HttpGet]
public async Task<IActionResult> MyEvent()
{
    var client = _httpClientFactory.CreateClient();
    var userId = User.FindFirstValue(ClaimTypes.
       NameIdentifier);
    int id = int.Parse(userId);

    client.DefaultRequestHeaders.Add("Cookie", Request.
       Headers["Cookie"].ToString());
    var response = await client.GetAsync($"https://
       localhost:7087/api/Api/MyEvents?id={id}");
```

```
    if (!response.IsSuccessStatusCode)
        return View("Error");

    var myEvents = await response.Content.
       ReadFromJsonAsync<List<Event>>() ?? new List<
       Event>();

    if (!myEvents.Any())
        TempData["Message"] = "Non hai ancora proposto
           alcun evento.";

    return View(myEvents);
}
```

3.4.2 Aspetti Rilevanti del Controller

Aspetti rilevanti:

- Viene recuperato l'`Id` dell'utente autenticato dai claims.
- I **cookie** di autenticazione vengono passati manualmente nella richiesta API.
- La sicurezza è garantita dal controllo sull'identità: un utente non può leggere eventi di altri.

3.4.3 Comunicazione Controller e APi con esempio

Inoltre qui possiamo vedere la logica che si estende a tutto il programma di come avviene la comunicazione tra Controller ed ApiController. La riga :

```
    var response = await client.GetAsync($"https://
       localhost:7087/api/Api/MyEvents?id={id}");
```

(In questo caso specifico è un Get, ma potrebbe essere anche un Post, Push, Delete)

chiama il controller Api che rimanda un codice. In base al codice ritornato il controller MyEvent procederà a svolgere altre operazioni.

Nel codice seguente possiamo vedere, in cima i tipi di ritorni, dove viene creata la variabile MyEvents la chiamata al database effetuata, utilizzando LINQ

Come possiamo vedere se viene ritornato OK() - codice 200 - Success con esso vengono passati una lista di eventi. Questo risultato verrà inviato a chi ha chiamato il controller, quindi MyEvent attraverso il Get.

```
[Authorize(Roles = "User")]
[HttpGet("MyEvents")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)
   ]
[ProducesResponseType(StatusCodes.
   Status401Unauthorized)]
[ProducesResponseType(StatusCodes.Status403Forbidden
   )]
public async Task<IActionResult> MyEvents(int id)
{
    try
    {
        //Controllo se non cerca di accedere alle
           info di un'altro utente
        var userId = User.FindFirstValue(ClaimTypes.
           NameIdentifier);

        int loggedUserId = int.Parse(userId);

        // Controlla che l'id passato coincida con
           quello dell'utente autenticato
        if (loggedUserId != id)
            return Forbid();

        var MyEvents = await _context.Events
            .Where(e => e.OrganizerId == id)  //
               CORRETTO: filtra gli eventi
            .ToListAsync();

        return Ok(MyEvents ?? new List<Event>());
    }
    catch (Exception ex)
    {
        // Log dell'errore
        Console.WriteLine($"Errore durante il
           recupero degli eventi: {ex.Message}");
        return StatusCode(500, "Errore interno del
           server");
    }
```

```
    }
```

3.4.4 Gestione Eliminazione Utenti e Consistenza dei Dati

Nel seguente esempio vediamo un metodo `HttpPost` chiamato `DeleteUsers`, il quale consente di eliminare uno o più utenti dal database. A differenza del caso precedente, in cui il metodo restituiva semplicemente una lista di eventi, qui il controller API deve eseguire più operazioni dipendenti tra loro nello stesso metodo.

Tali operazioni devono andare a buon fine o fallire insieme, in modo da garantire la consistenza del database. Infatti, quando viene richiesto di eliminare un utente:

- vengono rimossi i record di partecipazione associati a quell'utente;
- vengono rimossi o modificati gli eventi organizzati dall'utente (se non approvati vengono eliminati, mentre se già approvati viene impostato come organizzatore l'amministratore corrente);
- infine viene eliminato l'utente stesso.

```
    [HttpPost("delete")]
    public async Task<IActionResult> DeleteUsers([
       FromBody] idActionRequest request)
    {
        if (request == null || request.idSelected.Length
            == 0 || request.AdminId == null)
            return BadRequest("Nessun utente selezionato
               .");

        try
        {
            // Trovo tutti gli utenti selezionati da
               eliminare
            var usersToDelete = await _context.Users
                .Where(u => request.idSelected.Contains(
                   u.Id))
                .ToListAsync();

            // Partecipazioni: tolgo le partecipazioni
               dell'utente eliminato
```

```
            var partecipazioniDaRimuovere = await
               _context.Participations
                .Where(p => request.idSelected.Contains(
                   p.ParticipationUserId))
                .ToListAsync();
            _context.Participations.RemoveRange(
               partecipazioniDaRimuovere);

            // Eventi: tolgo gli eventi dell'utente
               eliminato (se non approvati),
            // se approvati metto come organizer l'admin
                corrente
            var eventiUtenti = await _context.Events
                .Where(e => request.idSelected.Contains(
                   e.OrganizerId))
                .ToListAsync();

            foreach (var evento in eventiUtenti)
            {
                if (!evento.IsApproved)
                {
                    _context.Events.Remove(evento);
                }
                else
                {
                    evento.OrganizerId = (int)request.
                       AdminId;
                }
            }

            _context.Users.RemoveRange(usersToDelete);
            await _context.SaveChangesAsync();

            return Ok($"Eliminati n.{usersToDelete.Count
               } utenti con successo.");
        }
        catch (Exception ex)
        {
            return StatusCode(500, "Errore durante l'
               eliminazione degli utenti.");
        }
    }
```

Capitolo 4

Configurazione dell'Applicazione

4.1 Program.cs e Gestione della Sicurezza

Il file `Program.cs` configura l'applicazione, in particolare il sistema di autenticazione basato su cookie.

Listing 4.1: Estratto di Program.cs

```
builder.Services.AddAuthentication(
   CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.Cookie.Name = "TempAuthCookie";
        options.LoginPath = "/Account/Login";
        options.AccessDeniedPath = "/Account/
           AccessDenied";

        options.Events = new CookieAuthenticationEvents
        {
            OnRedirectToLogin = ctx =>
            {
                if (ctx.Request.Path.StartsWithSegments
                   ("/api"))
                {
                    ctx.Response.StatusCode =
                       StatusCodes.Status401Unauthorized
                       ;
                    return Task.CompletedTask;
                }
```

```
                if (ctx.Request.Path == "/" || ctx.
                   Request.Path.StartsWithSegments("/
                   Home"))
                {
                    ctx.Response.Redirect(options.
                       LoginPath);
                }
                else
                {
                    ctx.Response.Redirect(options.
                       AccessDeniedPath);
                }
                return Task.CompletedTask;
            },

            OnRedirectToAccessDenied = ctx =>
            {
                if (ctx.Request.Path.StartsWithSegments
                   ("/api"))
                {
                    ctx.Response.StatusCode =
                       StatusCodes.Status403Forbidden;
                    return Task.CompletedTask;
                }
                ctx.Response.Redirect(ctx.RedirectUri);
                return Task.CompletedTask;
            }
        };
    });
```

4.2 Funzionalità Principali della Configurazione

Funzionalità principali:

- Autenticazione a cookie con gestione centralizzata.
- Distinzione tra richieste `API` e richieste standard: per le API non avviene redirect, ma viene restituito un codice HTTP (401 o 403).
- Ripulitura dei cookie all'avvio per garantire una sessione sicura.

- Swagger abilitato in fase di sviluppo per documentare le API.

Capitolo 5

Modelli del Database

5.1 Modello User

5.1.1 Struttura della Classe User

Il modello `User` rappresenta la struttura degli utenti nel sistema:

Listing 5.1: Classe User

```
public class User
{
    public int Id { get; set; }

    [Required(ErrorMessage = "Il nome    obbligatorio")]
    public string Name { get; set; }

    public string Surname { get; set; }
    public string? Aka { get; set; }
    public string? InstaProfile { get; set; }

    [Required(ErrorMessage = "L'email    obbligatoria")]
    [EmailAddress(ErrorMessage = "Inserisci un'email
       valida")]
    public string Email { get; set; }

    [Required(ErrorMessage = "La password
       obbligatoria")]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [Required]
```

```
    [DataType(DataType.Password)]
    [Compare("Password", ErrorMessage = "Le password non
        corrispondono")]
    [NotMapped]
    public string ConfermaPassword { get; set; } =
       string.Empty;

    [Required]
    public string Ruolo { get; set; } = "User";

    public virtual ICollection<Participation>
       Participations { get; set; } = new List<
       Participation>();
    public virtual ICollection<Event> OrganizedEvents {
       get; set; } = new List<Event>();
}
```

5.1.2 Caratteristiche del Modello User

Aspetti importanti:

- Uso delle `DataAnnotations` per validazione (es. email valida, password obbligatoria).
- Attributo `[NotMapped]` per evitare il salvataggio della conferma password nel database.
- Relazioni con partecipazioni ed eventi organizzati tramite `ICollection`.

5.2 Mapping del Database

5.2.1 Configurazione Entity Framework

Ogni modello principale User, Event, Partecipation definisce una tabella del database. Di seguito il codice che mappa le tabelle e le relazioni:

Listing 5.2: Configurazione del Database Context

```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using ProgettoTSWI.Models;

namespace ProgettoTSWI.Data
```

```
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<
            ApplicationDbContext> options)
            : base(options)
        {
        }
        public DbSet<User> Users { get; set; }

        public DbSet<Event> Events { get; set; }
        public DbSet<Participation> Participations { get
            ; set; }

        protected override void OnModelCreating(
            ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Participation>()
                .HasOne(p => p.User)
                .WithMany(u => u.Participations)
                .HasForeignKey(p => p.
                    ParticipationUserId);

            modelBuilder.Entity<Participation>()
                .HasOne(p => p.Event)
                .WithMany(e => e.Participations)
                .HasForeignKey(p => p.
                    ParticipationEventId);

            modelBuilder.Entity<Event>()
                .HasOne(e => e.Organizer)
                .WithMany(u => u.OrganizedEvents)
                .HasForeignKey(e => e.OrganizerId)
                .OnDelete(DeleteBehavior.Restrict); //
                    Previene la cancellazione cascata
        }
    }
}
```

Capitolo 6

Istruzioni per l'Installazione e Avvio

6.1 Requisiti di Sistema

6.1.1 Prerequisiti

Per eseguire correttamente il progetto, è necessario avere installato:

- **Visual Studio** (con workload `.NET e sviluppo web`) **oppure** il **.NET SDK**;
- **SQL Server / SQL Express / LocalDB**.

6.2 Configurazione Iniziale

6.2.1 Preparazione del Progetto

1. Aprire il progetto in Visual Studio (oppure posizionarsi nella cartella tramite terminale).
2. Modificare la stringa di connessione presente nel file `appsettings.json`, nella sezione:

6.2.2 Configurazione Stringa di Connessione

```
"ConnectionStrings": {
  "DefaultConnection": "Server=(localdb)\\MSSQLLocalDB;
  Database=DanceProj;
```

```
  Trusted_Connection=True;
  MultipleActiveResultSets=true;
  TrustServerCertificate=True;
  Encrypt=False"
}
```

6.2.3 Configurazioni Specifiche

Adattare la stringa in base alla configurazione locale:

- LocalDB (default Visual Studio):

  ```
  Server=(localdb)\MSSQLLocalDB;Database=DanceProj;
  Trusted_Connection=True;
  ```

- SQL Server Express:

  ```
  Server=localhost\SQLEXPRESS;Database=DanceProj;
  Trusted_Connection=True;
  ```

- SQL Server con autenticazione SQL:

  ```
  Server=localhost,1433;Database=DanceProj;
  User Id=sa;Password=tuapassword;
  ```

6.3 Creazione e Popolamento del Database

6.3.1 Applicazione delle Migration

Il progetto contiene già le `migration` nella cartella `Migrations/`. Per creare il database e le relative tabelle:

- Da **Visual Studio → Package Manager Console**:

  ```
  Update-Database
  ```

- Oppure da **CLI (.NET)** nella cartella del progetto:

  ```
  dotnet ef database update
  ```

Questo comando crea il database se non esiste e applica automaticamente tutte le migration.

6.3.2 Popolamento dei Dati Iniziali

Dopo aver creato la struttura del database, è possibile popolarlo con i dati iniziali utilizzando il file `IinitialDataDatabase.sql` presente all'interno della cartella "DatiDatabase" del progetto:

1. **Metodo consigliato (Visual Studio)**:
   - Aprire **Visual Studio**
   - Menu → **View** → **SQL Server Object Explorer**
   - Connettersi a `(localdb)\MSSQLLocalDB` o alla propria istanza locale
   - Trovare il database `DanceProj` e cliccare destro → **New Query**
   - Aprire il file `IinitialDataDatabase.sql` e copiare il contenuto
   - Eseguire lo script con il pulsante **Execute** o **F5**
2. **Metodo alternativo (Command Line)**:

   ```
   sqlcmd -S (localdb)\MSSQLLocalDB -d DanceProj -i
   "DatiDatabase\IinitialDataDatabase.sql"
   ```

3. **Metodo PowerShell**:

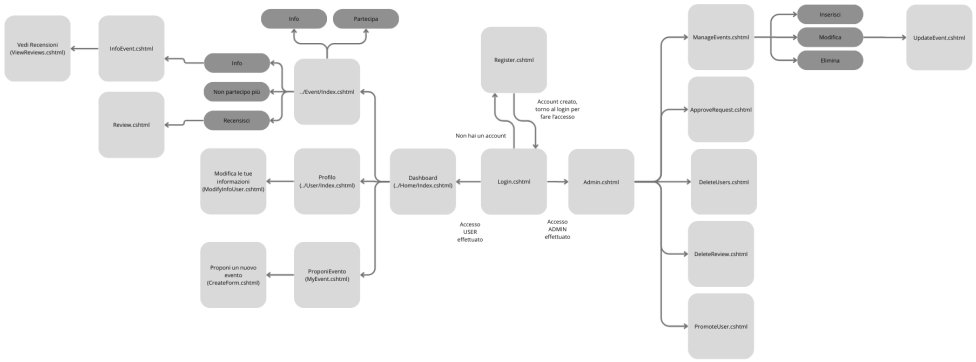
   ```
   Invoke-SqlCmd -ServerInstance "(localdb)\MSSQLLocalDB"
   -Database "DanceProj"
   -InputFile "DatiDatabase\IinitialDataDatabase.sql"
   ```

Nota importante: Il file contiene solo dati, non la struttura delle tabelle. Assicurarsi di aver prima eseguito le migration per creare le tabelle.

6.4 Avvio dell'Applicazione

6.4.1 Schema dell'Applicazione

Diagramma di navigazione dell'applicazione:



6.4.2 Procedura di Avvio

1. Avviare il progetto con `F5` in Visual Studio, oppure eseguendo:

   ```
   dotnet run
   ```

2. L'applicazione sarà disponibile su `localhost` agli indirizzi indicati in `launchSettings.json`, ad esempio:

   - `https://localhost:7087`
   - `http://localhost:5062`

6.5 Aggiornamenti Futuri

6.5.1 Gestione delle Modifiche al Modello

Se vengono modificate le classi `Model` (`User`, `Event`, `Participation`, ecc.), è necessario generare una nuova migration ed applicarla con i seguenti comandi:

```
dotnet ef migrations add NomeMigrazione
dotnet ef database update
```

Le migration già presenti non vanno rigenerate: è sufficiente eseguire nuovamente `Update-Database`.