Building CRUD API Servers

Writing API servers that manage data

- Let's create an API server that can perform CRUD actions on some collection of data.

Game Night!

- We will create an API to manage our game nights.
- It will track:
  - The name of the game we will play
  - The name of the person hosting the event
  - The address of the event
  - The date and time the game will start
  - The minimum number of players
  - The maximum number of players

API Definition

We are going to make an API that has the ability to CRUD (Create, Read, Update, and Delete) games.

The first thing we should do is design our API to support all of these and to follow a common convention.

Resource

We will follow these guidelines while building our API:

- GameNight is the model we are going to manage.
- If an endpoint uses the `GET` verb we expect the endpoint to return the same resource each time and not modify it.
- If an endpoint uses `POST/PUT/DELETE` it will modify the resource in some way.
- `POST` will modify the "list of all games" resource by adding a new GameNight.
- `PUT` will modify a specific game night by supplying new values.
- `DELETE` will modify a specific game night by removing it from the "list of all game nights".

Endpoint	Purpose
GET /GameNights	Gets a list of all games
GET /GameNights/{id}	Gets the single specific game night given by its id
POST /GameNights	Creates a new game night, assigning a new ID for the game night. The properties of the game night are given as JSON in the BODY of the request
PUT /GameNights/{id}	Updates the single specific game night given by its id. The updated properties of the game night are given by JSON in the BODY of the request
DELETE /GameNights/{id}	Deletes the specific game night given by its id

Patterns!

This is a very typical pattern of API for a CRUD style application.

These URL patterns and VERB combinations are enough of a pattern that we can typically make some guesses as to what an API does by only looking at the URL +VERB *definition.*

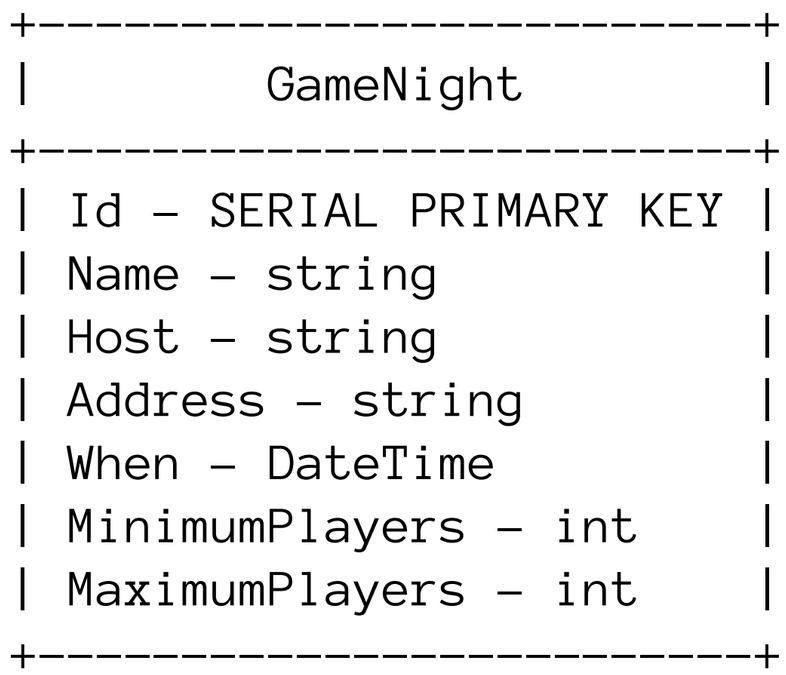
Creating Our Application

To generate an app with API and database support:

```
dotnet new sdg-api -o GameNightWithFriends
```

Generating an ERD

Our ERD for this application is simple since it is only dealing with a single entity: a `GameNight`.



Generating our database and our tables

We can use a feature of `Entity Framework` we may not have used yet: `Migrations`

For more details on migrations you can see this lesson.

Migrations

`Migrations` are the ability for `EF` to detect changes to our `C#` models and *automatically* generate the required SQL to change the definition of the database.

This is the idea of **Code First** database modeling. What we had done before, creating our tables manually in `SQL` was considered **Database First**.

Creating a Migration

The first thing we do is define our model.

In the **Models** directory, create the `GameNight.cs` file and define all the fields.

```
public class GameNight
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Host { get; set; }
    public string Address { get; set; }
    public DateTime When { get; set; }
    public int MinimumPlayers { get; set; }
    public int MaximumPlayers { get; set; }
}
```

Next step, inform our `DatabaseContext` of this model

After this code:

```
public partial class DatabaseContext : DbContext
{
```

Add this statement to let the `DatabaseContext` know we want to track `GameNight` in a `GameNights` table:

```
public partial class DatabaseContext : DbContext
{
    public DbSet<GameNight> GameNights { get; set; }
```

Next up: generate a migration

Any time we change the **properties** of a *model* **OR** we create a **new** model we must generate a *Database Migration* and *update* our database.

Before we can generate a migration we should do a quick check to make sure our code is free of syntax errors:

```
dotnet build
```

Then:

```
dotnet ef migrations add AddGameNights
```

Naming migrations

The name of our migration should attempt to capture the database structure change we are making.

In this case we are **Add** ing the `GameNights` table.

Next up: ensure your migration is good

This step is **IMPORTANT**

Do **not** skip it.

Most project questions of **why is my app not working** relates to a broken migration.

Checking our migration

You should have at least two new files in `Migrations`, one ending in `_AddGameNights.cs`.

Open that file and ensure the `Up` method has code for creating a table and defining columns.

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "GameNights",
        columns: table => new
        {
            Id = table.Column<int>(nullable: false)
                .Annotation("Npgsql:ValueGenerationStrategy", NpgsqlValueGenerationStrategy.IdentityByDefaultColumn),
            Name = table.Column<string>(nullable: true),
            Host = table.Column<string>(nullable: true),
            Address = table.Column<string>(nullable: true),
            When = table.Column<DateTime>(nullable: false),
            MinimumPlayers = table.Column<int>(nullable: false),
            MaximumPlayers = table.Column<int>(nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_GameNights", x => x.Id);
        });
}
```

Let's walk through this code

Update the database with this migration change

To *run* the migration against our database:

```
dotnet ef database update
```

This should create our `GameNights` table for us!

Time to make a controller!

To run the code generator:

```
dotnet aspnet-codegenerator controller
                            --model GameNight
                            -name GameNightsController
                            --useAsyncActions
                            -api
                            --dataContext DatabaseContext
                            --relativeFolderPath Controllers
```

NOTE This would normally be all on one line

GameNightsController.cs

Let's review this code. There is a lot there!

NOTE There is also a detailed walk through of this code in the handbook lesson.

Let's run the application and see what it can do!

Validation

Adding a check to make sure we have at *least* two players.

Add this to the `PutGameNight` and to the `PostGameNight` methods.

```
// Add a check to make sure we have enough players.
if (gameNight.MinimumPlayers < 2)
{
    return BadRequest(new { Message = "You need at least 2 players!" });
}
```

Adding associated Players

Our Game Night app is a success! Now we want to update it to keep track of data of the players that attended each game night.

Create a Player model

```
namespace GameNightWithFriends.Models
{
    public class Player
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int GameNightId { get; set; }
        public GameNight GameNight { get; set; }
    }
}
```

And add the information to our Database Context

```
public DbSet<Player> Players { get; set; }
```

Creating a whole new controller versus a nested/additional method on a current controller

Adding to the existing controller

```
// Adding Players to a game night
// POST /api/GameNights/5/Players
[HttpPost("{id}/Players")]
public async Task<ActionResult<Player>> CreatePlayerForGameNight(int id, Player player)
//                                       |       |
//                                       |       Player deserialized from JSON from the body
//                                       |
//                                      GameNight ID comes from the URL
```

```
// Adding Players to a game night
// POST /api/GameNights/5/Players
[HttpPost("{id}/Players")]
public async Task<ActionResult<Player>> CreatePlayerForGameNight(int id, Player player)
//                                      |       |
//                                      |       Player deserialized from JSON from the body
//                                      |
//                                      GameNight ID comes from the URL
{
    // First, lets find the game night (by using the ID)
    var gameNight = await _context.GameNights.FindAsync(id);

    // If the game doesn't exist: return a 404 Not found.
    if (gameNight == null)
    {
        // Return a `404` response to the client indicating we could not find a game night with this id
        return NotFound();
    }

    // Associate the player to the given game night.
    player.GameNightId = gameNight.Id;

    // Add the player to the database
    _context.Players.Add(player);
    await _context.SaveChangesAsync();

    // Return the new player to the response of the API
    return Ok(player);
}
```