.NET Core -ACCENTURE- 26MAY21

Avisos

ACTIVIDAD 5 - CREAR ENTIDADES DE DOMINIO

Su progreso

email task2

Salvo test database documento PDF

web-listar-juegos

Actividad 5 - Entidad Player

Definir entidad game insertar datos

Video Controller entidad game

Video Definir game player

Video Definir entidad ship parte 1

Video Definir entidad ship parte 2

Video Definir entidad salvo

Video Definir entidad score

Ahora que hemos configurado el contexto de la base de datos podemos comenzar a crear las entidades que conformarán nuestra aplicación. Lee sobre entity framework en https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx ya que será el ORM que usaremos.

Definir la entidad Player

En la carpeta Models agrega una nueva clase llamada Player, la entidad player debe tener un campo Email de tipo String que será llenado con una dirección de correo. Como la entidad se guardará en la base de datos recuerda crear el campo de tipo long Id. Sigue el ejemplo https://docs.microsoft.com/en-us/ef/core/modeling/keys#conventions para ver una muestra de como crear una clase y leer sobre las convenciones de definición de claves, además puedes encontrar más información sobre las clases en https://www.tutorialsteacher.com/csharp/csharp-class y en https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/concepts/object-oriented-programming. Tambien puedes ver https://www.entityframeworktutorial.net/efcore/entity-framework-core-console-application.aspx en el apartado creating the model.

Una vez creada la entidad debemos agregarla al contexto de la base de datos, para ello agregamos una propiedad de tipo DbSet a la clase SalvoContext, mira el ejemplo https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro?view=aspnetcore-2.2#create-the-database-context-1 tomando como referencia la clase Course. Puedes revisar de nuevo el ejemplo de las clases Student y Enrollment ya que también se aplica.

Luego de crear la entidad ejecuta en la consola nuget (herramientas, luego en administrador de paquetes nugget, pulsando consola de administrado de paquetes) los comandos:

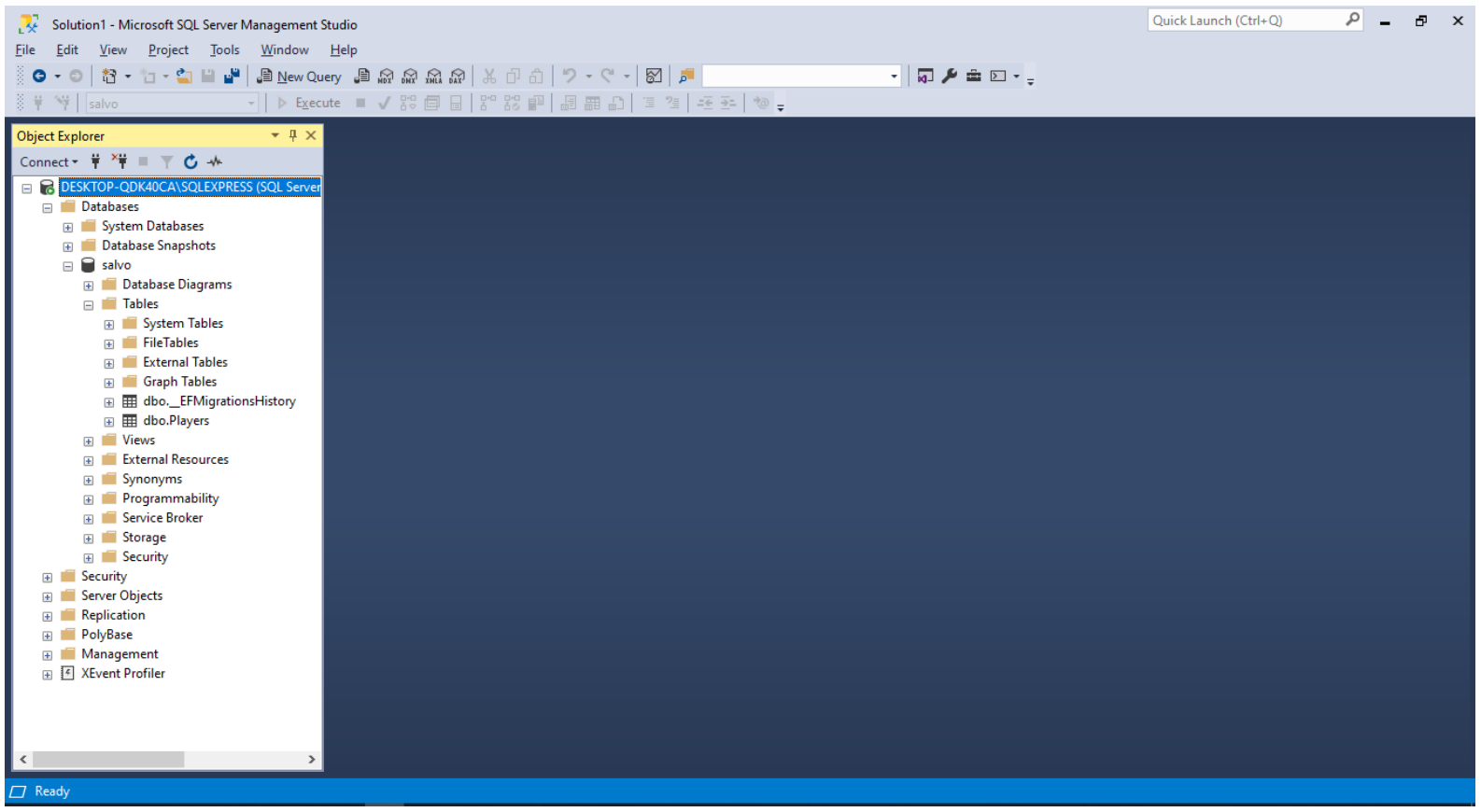
```
Add-Migration InitialCreate
Update-Database
```

Tal como se muestra en el ejemplo https://docs.microsoft.com/en-us/aspnet/core/data/ef-rp/migrations?view=aspnetcore-2.2&tabs=visual-studio#create-an-initial-migration-and-update-the-db

NOTA: si se muestra el mensaje "Both Entity Framework 6.x and Entity Framework Core commands are installed" solo hay que colocar EntityFrameworkCore\ antes de cada comando (EntityFrameworkCore\Add-Migration InitialCreate)

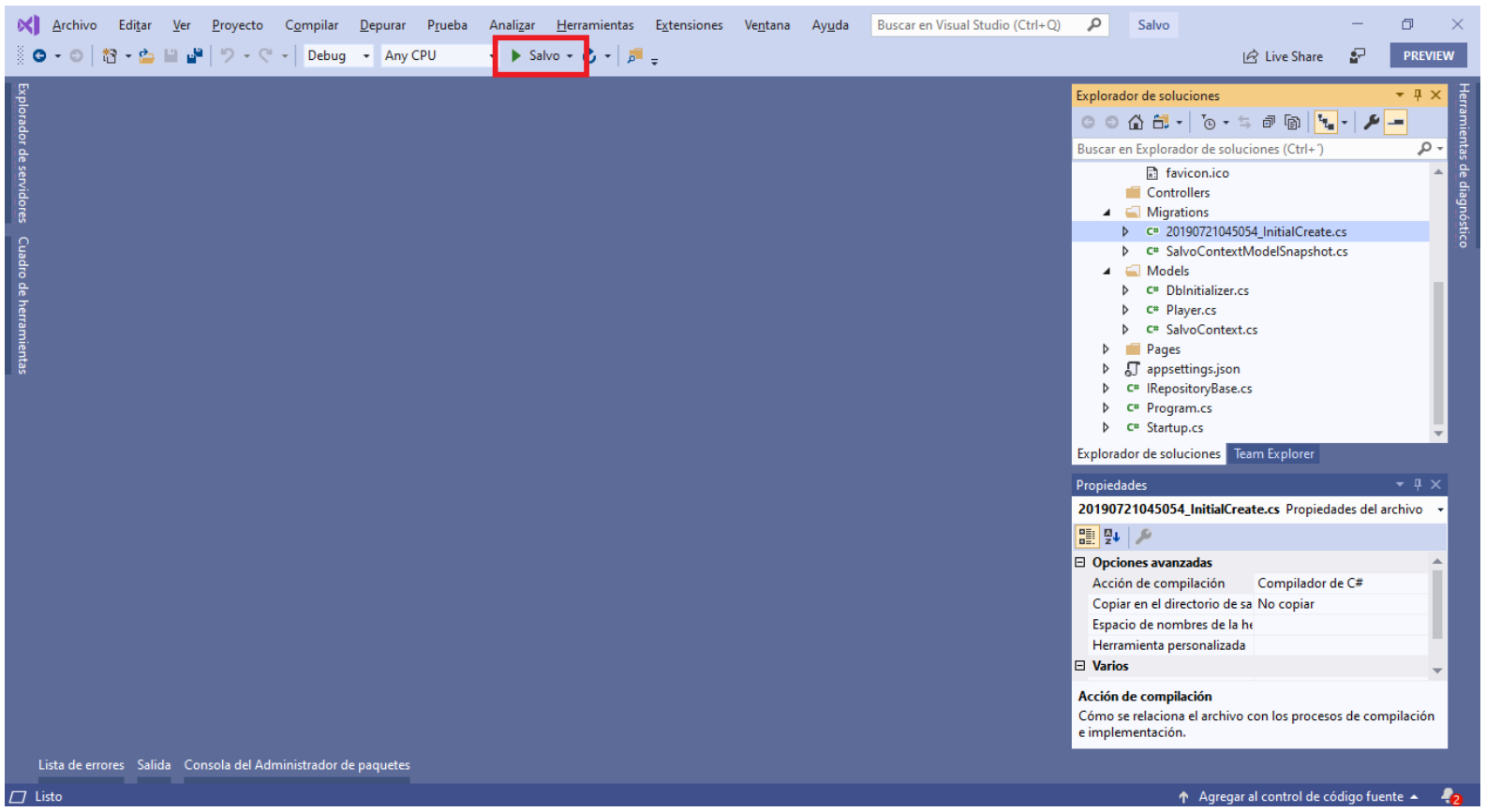
El primer comando crea los migrations con el nombre que indiquemos y el segundo los ejecuta en la base de datos, puedes leer más sobre los migration en https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/index , a partir de ahora cada vez que se crea una nueva entidad se deberá ejecutar el comando Add-migration add{NombreEntidad}Entity. Podremos ver que se ha creado una carpeta llamada Migrations en donde estarán todos los migrations que se vayan creando a medida de que crezca la aplicación y se agreguen entidades.

Es buen momento para ejecutar el programa Sql Management Studio, éste te permitirá explorar la base de datos Salvo y ver las tablas creadas. Al abrirlo debes dejar seleccionada la autenticación de windows y presionar conectar. Una vez que se conecta a la base de datos puedes ir al explorador de objetos a la izquierda en databases y mirar el esquema salvo, luego en tables para poder ver la tabla dbo.players.



Ahora se debe crear la data de prueba, en la carpeta Models agregar una nueva clase llamada DbInitializer al crearse la clase debemos agregar el tipo de clase static (static class DbInitializer), esto nos permitirá utilizar la clase sin tener que instanciarla ya que solo se utiliza con el propósito de insertar la data de prueba. Sigue el ejemplo https://docs.microsoft.com/en-us/aspnet/core/data/ef-rp/intro?view=aspnetcore-2.2&tabs=visual-studio#add-code-to-initialize-the-db-with-test-data fijandote como queda el método Initialize. **Importante**: No coloques la linea de código con la instrucción context.Database.EnsureCreated();. **Debes utilizar el archivo 'Salvo test database' proporcionado al inicio de la actividad para ver los datos de los players a crear**. Por último y siguiendo el mismo ejemplo modificaremos la clase Program ubicada en el archivo Program.cs especificamente el método main, fijándote que ahora existe una variable llamada host la cual tendrá la instancia generada del Web Host para luego llamar el método run desde la misma, puedes leer sobre el Web Host en https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/web-host?view=aspnetcore-2.2

Ejecuta la aplicación, para ello seleccionar la configuración Salvo en la barra de herramientas del Visual Studio, luego presionar el botón verde con el símbolo reproducir.



Verifica que no hayan errores, de lo contrario revisa el código y vuelve a ejecutar. Para comprobar que se han guardado debes entrar en el SQL Management Studio y sobre la tabla dbo.Players pulsar con el botón derecho del mouse para luego pulsar seleccionar las primeras 100 filas, deberás ver los registros almacenados.

Al finalizar envía tu proyecto comprimiendo la carpeta del mismo, recuerda presionar con el botón derecho del mouse sobre el proyecto y pulsar la opción limpiar antes de crear el archivo comprimido.

Definir Player - Envía el proyecto en un .zip

Definir la entidad Game

Repite el proceso para crear la entidad Player con la entidad Game, se debe crear la clase en la carpeta Models (recordar que Game solo tiene id y fecha de creación, para ver cómo trabajar con fechas puedes leer https://docs.microsoft.com/en-us/dotnet/api/system.datetime?view=netcore-2.2#assigning-a-computed-value busca información sobre el método addHours). Luego se debe agregar al contexto de la base de datos SalvoContext con una propiedad de tipo DbSet. Una vez se haya creado la entidad y se haya agregado al contexto el siguiente paso es ejecutar en la consola de administración de NuGet los comandos:

```
Add-Migration addGameEntity
Update-Database
```

Se deberá crear la tabla Game en la base de datos.

Si te equivocas en el nombre del migration o en al gún campo de la entidad Game no te preocupes, los migration se pueden revertir, solo debes ejecutar el comando Update-Database seguido del nombre del migration al que quieres revertir el estado de la base de datos (el migration anterior) https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/applying?tabs=vs#command-line-tools (mirar la pestaña visual studio):

```
Update-Database InitialCreate
```

Si quieres modificar el migration (debido a cambios en la entidad Game) primero hay que revertir el migration para que los cambios sean revertidos en la base de datos (mencionado en el paso anterior), luego eliminar el migration con el comando Remove-Migration para eliminar el último migration creado:

```
Remove-Migration
```

Finalmente se debe ejecutar de nuevo el comando para crear el migration seguido del comando para actualizar la base de datos. Puedes leer más sobre los migrations en https://www.entityframeworktutorial.net/efcore/entity-framework-core-migration.aspx

Nota: eliminar y volver a crear el migration solo se hace cuando estamos en la fase de creación de la entidad desde cero, más adelante si se quiere agregar un nuevo campo a la entidad o se modifica de alguna manera ha de crearse un nuevo migration con esos cambios.

El último paso es crear la data de prueba de la entidad Game, de igual manera que con la entidad Player se debe modificar la clase DbInitializer de manera que permita agregar la data de la entidad Game. Ten en cuenta que antes se verificaba si habían Players creados para terminar la ejecución de la función con la instrucción return, se tendrá que modificar para que permita ingresar Games aunque existan ya Players creados. **Recuerda que debes utilizar el archivo 'Salvo test database' proporcionado al inicio de la actividad para ver los datos de los Games a crear**.

Implementar el controlador de Game

Un controlador es una clase con métodos para ejecutarse cuando se reciben solicitudes con patrones de URL específicos. Hay dos tipos comunes de controladores:

- Los controladores MVC definen métodos que obtienen datos de una solicitud, modifican opcionalmente los datos adjuntos a un objeto Modelo y finalmente devuelven los nombres de las vistas. Un nombre de vista es una cadena que nombra un archivo de plantilla HTML. El aspecto de los archivos de plantilla depende del sistema de plantillas que elija utilizar. El archivo de plantilla es HTML con un código de plantilla especial, por ejemplo, {{myData}}, que inserta los datos del modelo en el HTML final que se enviará al navegador web
- Los controladores REST definen métodos que obtienen datos de una solicitud, modifican opcionalmente los datos en una base de datos y finalmente devuelven un objeto que se puede convertir a JSON para enviar de vuelta al cliente web.

Crearemos el segundo tipo de controlador, para ello primero implementaremos el patrón repositorio para acceder a la base de datos, lee sobre el mismo en https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-implemenation-entity-framework-core#using-a-custom-repository-versus-using-ef-dbcontext-directly asi como también en https://blog.kylegalbraith.com/2018/03/06/getting-familiar-with-the-awesome-repository-pattern/

Sigue este ejemplo https://code-maze.com/net-core-web-development-part4/ en el título "Repository Pattern Logic" y en el título "Repository User Classes". No se creará la clase RepositoryWrapper mencionada en el título "Repository Wrapper" que indican en el artículo para mantener la simplicidad. Primero crearemos una nueva carpeta llamada Repositories en donde se crearán a su vez todas las clases que menciona el artículo, en resúmen se debe crear la interfaz IRepositoryBase, la clase RepositoryBase, la interfaz IGameRepository y la clase GameRepository. Cuando se cree la clase RepositoryBase y GameRepository se debe recordar que en el proyecto Salvo el contexto de la base de datos se llama SalvoContext en vez de RepositoryContext:

protected SalvoContext RepositoryContext { get; set; }.

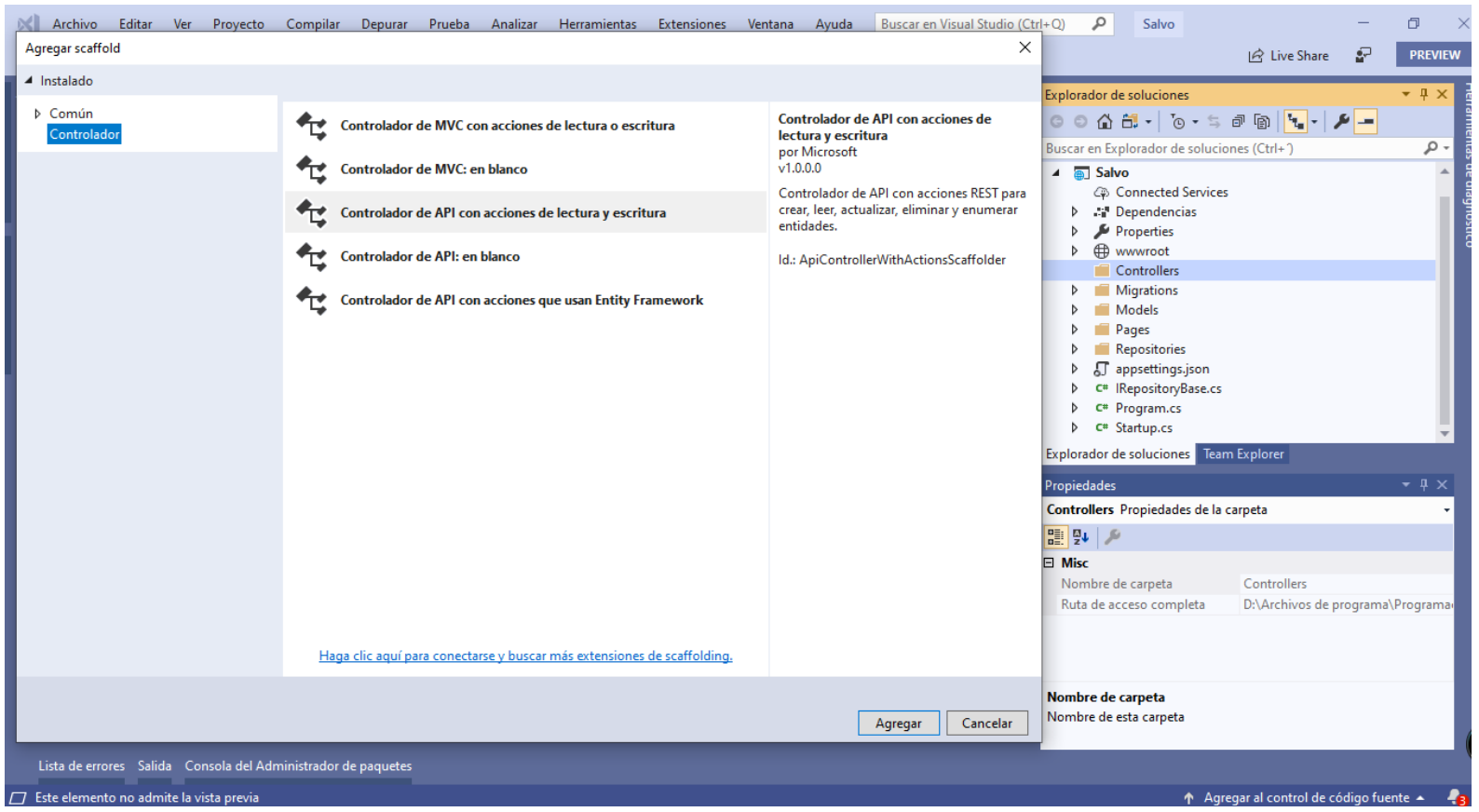
Cuando se indica en el ejemplo que se debe modificar la clase ServiceExtensions, ése código se colocará en la clase Startup:

```
services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
services.AddScoped<IGameRepository, GameRepository>();
```

Al colocar AddScoped indicamos que se agregue en el contenedor de inyección de dependencia (DI container) una instancia del servicio indicado que en este caso es GameRepository, luego cuando se necesite obtener una instancia de ése servicio solo se le debe indicar al DI Container que inyecte ésa dependencia. Más adelante se podrá apreciar al implementar el controlador de game. Puede leer un poco más en https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2#service-lifetimes

Puedes leer sobre las interfaces en https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface **Nota**: para agregar una interfaz es el mismo proceso que una clase, al abrirse la ventana para indicar el nombre se selecciona Interfaz.

Una vez se haya implementado el patrón repositorio para acceder a la base de datos es momento de crear el controlador de games. Se debe crear una nueva carpeta en el proyecto llamada Controllers, se pulsa sobre el botón derecho de la carpeta Controllers seguido de agregar, luego indicaremos controlador para que aparezca una ventana en la que se debe seleccionar la opción Controlador API con acciones de lectura y escritura, se pulsa agregar y seguidamente aparecerá un cuadro solicitando que se ingrese el nombre del controlador al que llamaremos GamesController.



Al indicar el nombre se comenzará a generar el controlador, como se indicó que se debía crear con acciones de escritura y lectura se agrega un paquete al proyecto llamado Microsoft.VisualStudio.Web.CodeGeneration.Design que como puedes intuir por su nombre es el encargado de generar el código.

La clase GamesController reperesentará el controller de la entidad Game, es decir será la encargada servir las peticiones que tengan que ver con dicha entidad, listar (GET), crear (POST), eliminar (DELETE), modifcar (PUT) games y demás operaciones. Por ahora se programará el código necesario para listar los games que se encuentren en la base de datos. Sigue el ejemplo https://code-maze.com/net-core-web-development-part5/#getAllOwners que demuestra como obtener todos los owners en el método get del controller OwnerController, se debe tener en cuenta que en el caso de salvo sería game. En resúmen se debe modificar la ruta base del controller para que indique /api/games, modificar la interfaz IGameRepository para crear el método GetAllGames, modificar la clase GameRepository para implementar el método GetAllGames y por último modificar de nuevo la clase GamesController para que el método Get retorne un IActionResult con los datos de los juegos **Nota**: recuerda que como no se creó el IRepositoryWrapper en el constructor de GamesController solo se recibirá por parámetro una IGameRepository:

```
private IGameRepository _repository;

public GamesController(IGameRepository repository)
{
    _repository = repository;
}
```

¡Ahora solo queda probarlo!, se debe ejecutar la aplicación como se hizo en la actividad anterior seleccionando la configuración Salvo en la barra de herramientas del Visual Studio, luego presionar el botón verde con el símbolo reproducir. Una vez se haya ejecutado el proyecto coloca en el navegador la url:

http://localhost:5000/api/games

Deberás ver la lista de juegos con id y fecha de creación:

```
[{"id":1,"creationDate":"2019-01-01T18:32:49.8716178"},
{"id":2,"creationDate":"2019-01-01T19:32:49.8725558"},
{"id":3,"creationDate":"2019-01-01T20:32:49.8725564"},
{"id":4,"creationDate":"2019-01-01T21:32:49.8725565"}]
```

Al finalizar envía tu proyecto comprimiendo la carpeta del mismo, recuerda presionar con el botón derecho del mouse sobre el proyecto y pulsar la opción limpiar antes de crear el archivo comprimido.

Definir Game - Envía el proyecto en un .zip

Definir entidad GamePlayer

Al tratar con colecciones de diferentes tipos de datos, tanto en clases de C# como en bases de datos, el código que escribes dependerá de la cardinalidad de la relación de los datos. Hay tres cardinalidades básicas:

- **Uno a uno**: Ejemplo: un jugador tendrá exactamente un número de identificación único que nadie más tiene.
- **Uno a muchos**: Ejemplo: un jugador puede tener varias direcciones de correo electrónico, pero cada correo electrónico pertenece a un solo jugador.
- **Muchos a muchos**: Ejemplo: un jugador puede jugar muchos juegos, y los juegos tienen más de un jugador.

Para esta tarea estaremos implementado el tercer tipo de relación. Puedes leer sobre ello en https://www.dummies.com/programming/sql/knowing-just-enough-about-relational-databases/ y también en https://docs.microsoft.com/en-us/ef/core/modeling/relationships#fully-defined-relationships

La entidad GamePlayer será la encargada de mantener la relación entre juegos y jugadores, para ello debes agregar una nueva clase llamada GamePlayer en la carpeta Models, la misma deberá tener id, otro llamado joinDate que tendrá la fecha de la creación de la entidad o que es lo mismo cuando un player entra a un juego, seguido de los campos necesarios para construir la relación con game y con player. Debes leer al completo https://www.entityframeworktutorial.net/efcore/one-to-many-conventions-entity-framework-core.aspx luego implementar la **convención número 4**, teniendo en cuenta que Student sería GamePlayer y Grade sería player o game.

Se debe comprender que en el ejemplo solo se tiene un lado de la relación, es decir Student se relaciona solo con Grade, en este caso GamePlayer (Student) se debe relacionar tanto con Game como con Player para ello debemos tener en GamePlayer un campo Player y otro campo Game, además deberemos modificar Player para tener una colección o lista de GamePlayer asi como tambien en Game (la lista se debe llamar GamePlayers en plural).

Una vez implementada la entidad recuerda agregarla al conexto de la base de datos SalvoContext con un nuevo campo de tipo DbSet. Como se agregó una nueva entidad en el modelo de datos se deberá ejecutar en la consola de administrador de paquetes NuGet el comando para agregar el nuevo migration addGamePlayerEntity y luego el comando para actualizar la base de datos:

```
Add-Migration addGamePlayerEntity
Update-Database
```

Nota: si tienes problemas con los migration recuerda los pasos en la actividad anterior para lidiar con ello.

Solo falta agregar la data de prueba modificando la clase DbInitialize, **Recuerda que debes utilizar el archivo 'Salvo test database' proporcionado al inicio de la actividad para ver los datos a crear**. Para éste caso se deberá buscar en la base de datos las entidades de Player y Game creadas con anterioridad (ya que el objeto GamePlayer debe ser inicializado con ellos), ésto se logra utilizando la función find() del contexto de la base de datos puedes leer https://www.entityframeworktutorial.net/querying-entity-graph-in-entity-framework.aspx ten en cuenta que el campo id de Player y Game es de tipo long por lo que al pasar el número por parámetro en la función find deberás indicar que es de ese tipo colocando una L al final find(1L).

Ejecuta la aplicación, ve al Managemente Studio para ver si los datos fueron cargados en la tabla GamePlayers, si se cargaron los datos continúa de lo contrario revisa tu código modifícalo y ejecuta de nuevo.

Mostrar listado de juegos con jugadores

Ahora se requiere que la lista de juegos que muestra el controlador GamesController en el método GET (/api/games) muestre los jugadores de cada juego, si ahora se ejecuta el proyecto y se coloca en el navegador la url *http://localhost:5000/api/games* se podrá observar que en el campo gamePlayers indica null:

[{"id":1,"creationDate":"2019-01-01T00:23:39.8783606","gamePlayers":null}]

Esto es debido a que no hemos indicado en la consulta que retorna los juegos que queremos traer también dicha información, para ello se debe modificar el repositorio base RepositoryBase a manera que permita indicar qué entidades queremos incluir en la consulta. Primero agregamos un nuevo método a la interfaz IRepositoryBase:

```
IQueryable<T> FindAll(Func<IQueryable<T>, IIncludableQueryable<T, object>> includes = null);
```

Si se observa se notará que existe otro método llamado FindAll también, acá se usa la sobrecarga de métodos que nos permite tener varios métodos con el mismo nombre pero con distinto tipo y cantidad de parámetros. Lee más en https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/member-overloading. Esto quiere decir que existen dos manera de utilizar el método FindAll: sin parámetros y la data viene como está programada o con parámetros y se puede indicar que la data venga con las entidades que se especifiquen incluidas. Ahora solo falta implementar el método, para ello se debe agregar el mismo en RepositoryBase:

```
public IQueryable<T> FindAll(Func<IQueryable<T>, IIncludableQueryable<T, object>> includes = null)
{
    IQueryable<T> queryable = this.RepositoryContext.Set<T>();

    if (includes != null)
    {
        queryable = includes(queryable);
    }

    return queryable.AsNoTracking();
}
```

El código indica que se buscará en el repository context la lista de la entidad a la que se haga referencia con T, además si por parámetro se indicó alguna entidad a incluir en el resultado includes != null entonces se incluyen. Esto se llama cargar datos relacionados, puedes leer sobre ello en https://docs.microsoft.com/en-us/ef/core/querying/related-data/eager acá se implementa un poco más complicado pero es debido a que tenemos nuestra capa de abstracción GameRepository y RepositoryBase. Puedes leer sobre la abastracción en https://www.geeksforgeeks.org/c-sharp-abstraction/

Como ya se tiene el repositorio base listo para permitir incluir data relacionada es hora de indicar que por cada Game se debe obtener (incluir) los GamePlayers asociados, para ello se debe modificar el repositorio de game (IGameRepository) de manera que tengamos la operación disponible para los juegos colocando:

```
IEnumerable<Game> GetAllGamesWithPlayers();
```

Luego se debe implementar el método en GameRepository:

```
public IEnumerable<Game> GetAllGamesWithPlayers()
{
    return FindAll(source => source.Include(game => game.GamePlayers)
        .ThenInclude(gameplayer => gameplayer.Player))
    .OrderBy(game => game.CreationDate)
    .ToList();
}
```

Se puede apreciar que el método no cambia mucho a GetAllGames, solo indicamos que se quiere incluir de cada game sus GamePlayer asociados game.GamePlayers y por cada gamePlayer su player. Por último se debe modificar el controlador de game GamesController para que llame al nuevo método GetAllGamesWithPlayers en vez de GetAllGames:

```
//antes
var games = _repository.GetAllGames();
//después
var games = _repository.GetAllGamesWithPlayers();
```

Solo queda probar, ejecuta la aplicación (puedes apretar F5 y visual studio ejecutará la aplicación con la configuración de inicio que esté seleccionada) ve a la url *http://localhost:5000/api/games* y se apreciará que indica un error*:*

SyntaxError: JSON.parse: end of data after property value in object at line 1 column 156 of the JSON data

Ésto se debe a que existe una relación circular entre Game y GamePlayers y Newtonsoft.Json (librería que serializa/transforma las entidades a Json) genera dicho error al encontrarla. Newtonsoft serializa la entidad Game, por lo que recorre todas sus propiedades transformándolas a JSON, al encontrarse con la propiedad gamePlayers se mete para serializar cada una entonces ve que GamePlayer a su vez tiene la la propiedad Game por lo que la serializa recorriendo todas sus propiedades y una de ellas es GamePlayer y así sucesivamente hasta que se realizan muchos ciclos y se genera el error. Para corregir ésto se puede utilizar el decorador [JsonIgnore] como muestra en https://docs.microsoft.com/en-us/ef/core/querying/related-data/serialization se debe observar que es la misma documentación que habla sobre incluir data relacionada. Si bien es cierto que con dicha técnica se puede solventar el problema en ésta práctica usaremos un acercamiento diferente ya que se implementará el patrón de diseño DTO (Data Transfer Object), puedes leer sobre esto en https://docs.microsoft.com/en-us/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-5.

Primero se creará una clase llamada GameDTO en la carpeta Models que en un principio contendrá los mismos campos que Game, luego al igual que con Game se creará otra clase llamada GamePlayerDTO con los mismos campos que GamePlayer, por último se creará una clase llamada PlayerDTO de igual manera con todos los campos de Player. Ahora se debe pensar qué campos de cada DTO debemos modificar o quitar, en principio GameDTO debe cambiar su listado de gamePlayers al tipo GamePlayerDTO, de GamePlayerDTO solo se quiere mostrar Id, JoinDate y Player teniendo en cuenta que el tipo de Player ahora será PlayerDTO y que PlayerDTO solo tendrá los campos Id y Email. Se puede apreciar que de ésta manera se eliminan las referencias circulares al eliminar las propiedades problemáticas en cada DTO. Solo falta indicar al controller que utilice éstos objetos en vez de los objetos/entidades de dominio directamente, mira el ejemplo https://arquitectosinbloques.wordpress.com/2017/09/06/usando-el-patron-dto-en-net/ en el título llenando los DTO, se debe tener en cuenta que hay tres DTO a crear uno dentro de otro.

Corregido ésto se ejecuta la aplicación (F5) y se coloca en el navegador la url *http://localhost:5000/api/games*, se debe mostrar el listado de juegos con GamePlayer y Player:

[{"id":1,"creationDate":"2019-07-28T00:23:39.8783606","gamePlayers:[{"id":1,"joinDate":"2019-07-28T00:26:22.6004604","player":{"id":1,"email":"j.bauer@ctu.gov"}}]

Al finalizar envía tu proyecto comprimiendo la carpeta del mismo, recuerda presionar con el botón derecho del mouse sobre el proyecto y pulsar la opción limpiar antes de crear el archivo comprimido.

Definir GamePlayer - Envía el proyecto en un .zip

Mostrar el listado de juegos en una página web

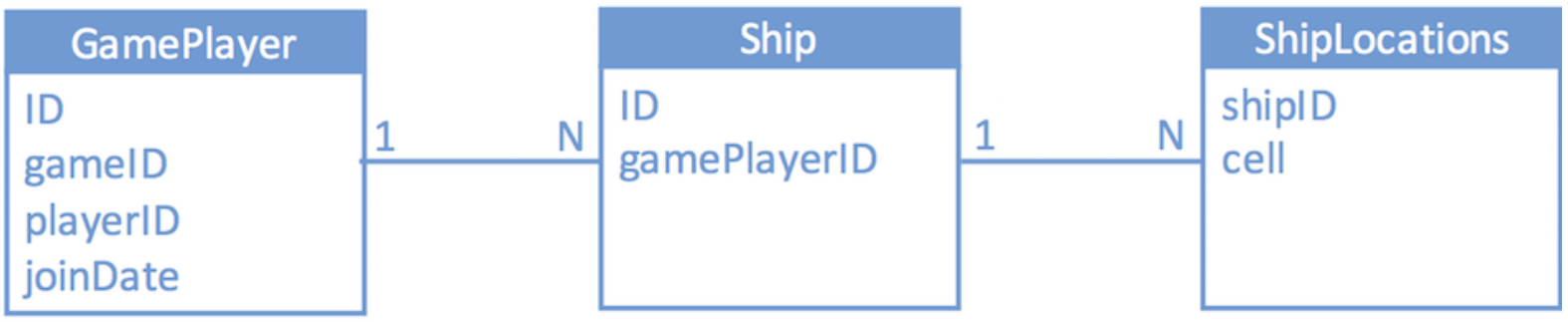
Descarga el paquete web-listar-juegos y copiar su contenido en la carpeta wwwroot, se puede eliminar todo lo que estaba en dicha carpeta así como también la carpeta Pages. Una vez copiado el contenido en la carpeta ejecutar el proyecto y colocar en el navegador la url *https://localhost:5001/index.html* se debe ver la página principal del juego Salvo con el listado de juegos creados.

email task3

Definir la entidad Ship

Afortunadamente, esta actividad no implica tantos nuevos conceptos como la tarea anterior. Esta es una oportunidad para aplicar y practicar lo que has aprendido, con ligeras variaciones.

Relaciones uno-a-muchos / One-to-Many



Un barco tiene varias ubicaciones, por ejemplo, H3, H4 y H5. Es decir, tienes una relación de uno a muchos entre barcos y ubicaciones.

En la tarea anterior, se implementó una relación de muchos a muchos entre Games y Players al crear un objeto GamePlayer que implementa dos relaciones de uno a muchos entre entidades:

- **Game** tiene una relación uno-a-muchos con GamePlayer
- **Player** tiene una relación uno-a-muchos con GamePlayer
- **Game** tiene una relación mucho-a-mucho con Player

GamePlayer tendrá una relación uno-a-muchos con Ship, por ende lo primero que se debe realizar es crear la entidad Ship, la cual tendrá id y un tipo además de indicar la relación con GamePlayer siguiendo la **convención 4** como se realizó en la actividad anterior https://www.entityframeworktutorial.net/efcore/one-to-many-conventions-entity-framework-core.aspx. Deberias terminar con que GamePlayer tiene un campo llamado Ships de tipo ICollection<Ship> y en Ship tienes el inverso de la relación hacia GamePlayer con un campo llamado GamePlayerId y otro llamado GamePlayer. Ahora bien, cada Ship tiene un listado de locaciones, sería conveniente crear en la entidad Ship una colección o lista de locaciones (public ICollection<string> Locations { get; set; }), el problema es que entity framework no permite colecciones de tipos primitivos(string,int,long,etc..) por ello se deberá crear una nueva entidad llamada ShipLocation que tendrá id y un campo llamado Location y como te puedes dar cuenta su relación sería de uno-a-mucho con Ship ya que un Ship puede tener muchas locaciones, así que debes terminar con que la entidad Ship tiene un campo llamado Locations de tipo ICollection<ShipLocations> y en la entidad ShipLocation deberás tener un campo ShipId y otro llamado Ship.

Una vez implementadas las entidades recuerda agregarlas al conexto de la base de datos SalvoContext con un nuevo campo de tipo DbSet. Como se agregaron nuevas entidades en el modelo de datos se deberá ejecutar en la consola de administrador de paquetes NuGet el comando para agregar el nuevo migration addShipEntity y luego el comando para actualizar la base de datos:

```
Add-Migration addShipEntity
Update-Database
```

Nota: si tienes problemas con los migration recuerda los pasos en la actividad definir la entidad game para lidiar con ello.

Al ejecutarse el migration se debe ver en la base de datos dos nuevas tablas creadas, Ships y ShipLocations. Ahora se deben llenar con los datos de prueba, **Recuerda que debes utilizar el archivo 'Salvo test database' proporcionado al inicio de la actividad para ver los datos a crear.** Como siempre se debe modificar la clase DbInitializer para agregar los Ships de igual manera que en la actividad anterior, para éste caso se deberá buscar en la base de datos las entidades GamePlayer creadas con anterioridad (ya que se deben agregar los Ship a cada GamePlayer), ésto se logra utilizando la función find() del contexto de la base de datos.

Ejecuta la aplicación, ve al Managemente Studio para ver si los datos fueron cargados en la tabla GamePlayers, si se cargaron los datos continúa de lo contrario revisa tu código modifícalo y ejecuta de nuevo.

Implementar el controlador de vista de juego GameView

Se debe crear el controlador de juego GamePlayersController para tener un nuevo método que responda a las URL que solicitan información sobre un juego específico desde el punto de vista de un jugador específico. Esa información por ahora debe incluir:

- El jugador del juego: nombre y su gamePlayer ID
- El jugador oponente del juego: nombre y su gamePlayer ID
- Qué barcos tiene el jugador en ese juego (destroyer,patroalboat,etc..)
- Dónde están ubicados esos barcos

Ya que tenemos que buscar los datos en la base de datos se deberá crear un nuevo repositorio en la carpeta Repositories, debes guiarte de la práctica de la creación de la entidad Game (Implementar el controlador de Game) en el que se crean IGameRepository y GameRepository. Se debe crear una interfaz llamada IGamePlayerRepository que tendrá el método:

public GamePlayer GetGamePlayerView(int idGamePlayer);

Seguido se crea una nueva clase llamada GamePlayerRepository que tendrá la implementación del método *GetGamePlayerView*:

return FindAll(source => source.Include(gamePlayer => gamePlayer.Ships)
.ThenInclude(ship => ship.Locations)

```
                .Include(gamePlayer => gamePlayer.Game)
                    .ThenInclude(game => game.GamePlayers)
                        .ThenInclude(gp => gp.Player)
                )
            .Where(gamePlayer => gamePlayer.Id == idGamePlayer)
            .OrderBy(game => game.JoinDate)
            .FirstOrDefault();
```

Además de heredar la clase BaseRepository (de tipo GamePlayer) e implementar la interfaz IgamePlayerRepository. Recuerda agregar el constructor.

Por último se debe agregar al contenedor de dependencia en la clase Startup con addScope:

```
services.AddScoped<IGamePlayerRepository, GamePlayerRepository>();
```

NOTA: es buen momento para repasar sobre cómo consultar la base de datos con entity framework, https://www.entityframeworktutorial.net/efcore/querying-in-ef-core.aspx

Para crear el Controlador el primero paso es generar un GamePlayersController recuerda la actividad de implementar player en la que se creó un controller, se debe pulsar con el botón derecho en la carpeta controllers, luego agregar y seleccionar controlador, luego Controlador Api con acciones de lectura y escritura y colocar el nombre GamePlayersController. Una vez allí modificar Route() por api/gamePlayers para indicar que esa será la ruta principal del controller, además se debe agregar el repositorio IGameRepository y crear el constructor de GamePlayersConstroller que recibe un IGameRepository).

Lo siguiente es modificar el método Get(int id) que tiene el decorador [HttpGet("{id}", Name = "Get")] puedes leer sobre modificadores en https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-2.2&tabs=visual-studio#routing-and-url-paths-21. Lo que indica el modificador es que cuando en la url se coloque /api/gamePlayers/5 ese 5 sea tomando como {id} /api/gamePlayers/{id} y luego insertado en el parámetro int id Get(int id) del método. Lo que debes modificar en principio es el atributor name del modificador para que indique "GetGameView" [HttpGet("{id}", Name = "GetGameView")]. Esto se hace ya que no podemos tener dos controller con el mismo punto de acceso, fijate que en el controller GamesController existe un método [HttpGet("{id}", Name = "Get")].

Guíate por el método Get (siendo: public IActionResult Get()) de GamesController para indicar que debe retornar un IActionResult, así como en ese método Get este método GetGameView usará DTOs para mostrar la información requerida quedando el JSON de la siguiente manera:


```
{
    "id":1
    ,"creationDate":"2019-01-01T00:23:39.8783606"
    ,"gamePlayers": [ {
        "id":1
        ,"joinDate":"2019-01-01T00:26:22.6004604"
        ,"player":{"id":1,"email":"j.bauer@ctu.gov"}
     } , {
        "id":2
        ,"joinDate":"2019-01-01T00:26:22.6014251"
        ,"player":{"id":2,"email":"c.obrian@ctu.gov"}
    } ]
    ,"ships": [
        {"id":1,"type":"Destroyer","locations":[{"id":1,"location":"H2"},{"id":36,"location":"H2"},
        {"id":37,"location":"H3"}]},{"id":24,"type":"PatroalBoat","locations":[{"id":41,"location":"B4"},
        {"id":42,"location":"B5"}]},{"id":25,"type":"Submarine","locations":[{"id":38,"location":"E1"},
        {"id":39,"location":"F1"},{"id":40,"location":"G1"}]}
    ]
}
```


Primero se deben crear los DTOs de Ship y ShipLocation en la carpeta Models, ShipDTO y ShipLocationDTO. Al igual que con los otros DTOs solo debemos mostrar lo necesario, para ShipDTO solo crear las propiedades Id ,Type y Locations (recuerda que la lista debe ser de tipo ShipLocationDTO) de igual manera para ShipLocationDTO solo crear Id y Location. Como no vamos a retornar un GamePlayer pero se, se debe crear tambien un DTO llamado GameViewDTO el cual tendrá Id (del gameplayer), CreactionDate (del juego del gameplayer), GamePlayers (lista de tipo GamePlayerDTO) y Ships (lista de tipo ShipDTO), éste DTO mostrará la vista del juego como se indica el inicio de la actividad.

Por último se debe modificar el método GetGameView(int id) de GamePlayersController para que retorne el JSON indicado utilizando los DTOs recientemente creados (tomando como base GameViewDTO) guiándote por la actividad anterior de creación del GamesController. Ten en cuenta que para generar el GameViewDTO ya no se usará .select() porque no es una lista lo que retorna el método getGamePlayerView sino un único gamePlayer, puedes guardarlo en una variable (gp) luego creas una nueva variable que será la que retornarás y allí creas el nuevo GameViewDTO tomando los datos de esa variable que almacena el gamePlayer (gp).

Ejecuta la aplicación (F5) y se coloca en el navegador la url:

http://localhost:5000/api/gamePlayers/1

Se debe mostrar la vista del juego con los datos del juego, los gameplayers y los ships.

Definir Ship - Envía el proyecto en un .zip

web-mostrar-juego

Mostrar la vista del juego en una página web

Descarga el paquete web-mostrar-juego y copiar su contenido en la carpeta wwwroot (sobrescribir todo), una vez copiado el contenido en la carpeta ejecutar el proyecto y colocar en el navegador la url *https://localhost:5001/game.html?gp=1* se debe ver la página del juego con los barcos posicionados.

email task4

Definir la entidad Salvo

Esta tarea es muy similar a la tarea anterior ya que se implementará de nuevo una relación de uno-a-muchos (one-to-many) con la entidad Salvo, GamePlayer tiene muchos Salvos y a su vez Salvo tiene muchos SalvoLocations. Además no se tendrá que crear un nuevo repositorio ya que se usará el repositorio GamePlayerRepository ni tampoco un controller ya que se usará GamePlayersController.

Algo importante es que se necesita que en la vista de juego (lo que retorna GetGameView de la clase GamePlayersControllers) se muestren todos los salvos, tanto del jugador como del oponente y por cada salvo se muestre la información del jugador al que pertenece para diferenciarlos):


```
{
    "id":1
    ,"creationDate":"2019-01-01T00:23:39.8783606"
    ,"gamePlayers": [ {
        "id":1
        ,"joinDate":"2019-01-01T00:26:22.6004604"
        ,"player":{"id":1,"email":"j.bauer@ctu.gov"}
     } , {
        "id":2
        ,"joinDate":"2019-01-01T00:26:22.6014251"
        ,"player":{"id":2,"email":"c.obrian@ctu.gov"}
    } ]
    ,"ships": [
        {"id":1,"type":"Destroyer","locations":[{"id":1,"location":"H2"},{"id":36,"location":"H2"},
        {"id":37,"location":"H3"}]},{"id":24,"type":"PatroalBoat","locations":[{"id":41,"location":"B4"},
        {"id":42,"location":"B5"}]},{"id":25,"type":"Submarine","locations":[{"id":38,"location":"E1"},
        {"id":39,"location":"F1"},{"id":40,"location":"G1"}]}
    ]
    ,"salvos":[
        {
            "id":1,"turn":1,
```

```
                "player":{"id":1,"email":"j.bauer@ctu.gov"}
                ,"locations":[{"id":1,"location":"B5"},{"id":32,"location":"C5"},{"id":33,"location":"F1"}]
            }
            ,{
                "id":19,"turn":2
                ,"player":{"id":1,"email":"j.bauer@ctu.gov"}
                ,"locations":[{"id":34,"location":"F2"},{"id":35,"location":"D5"}]
            },
            {
                "id":17,"turn":2
                ,"player":{"id":2,"email":"c.obrian@ctu.gov"}
                ,"locations":[{"id":39,"location":"E1"},{"id":40,"location":"H3"},{"id":41,"location":"A2"}]
            },
            {
                "id":18,"turn":1
                ,"player":{"id":2,"email":"c.obrian@ctu.gov"}
                ,"locations":[{"id":36,"location":"B4"},{"id":37,"location":"B5"},{"id":38,"location":"B6"}]
            }
        ]}
}
```


Sigue los mismos pasos de la actividad anterior:

- Crear la entidad Salvo
- Crear la entidad SalvoLocation
- Modificar la entidad GamePlayer para agregar los salvos
- Agregar las entidades en el SalvoContext con una propiedad de tipo DbSet (**Nota:** te darás cuenta que genera un error al agregar el DbSet de salvo, ésto es porque Salvo es el nombre del namespace, por lo que en el tipo se debe colocar DbSet<Salvo.Models.Salvo> así se elimina la ambiguedad)
- Ejecutar los comandos para agregar el migration addSalvoEntity y actualizar la base de datos
- Agregar la data de prueba en DbInitialize, usando el archivo [Salvo test database]
- Ejecutar la aplicación para que se cree la data
- Actualizar el método GetGamePlayerView de la clase GamePlayerRepository para incluir los salvos y sus locaciones:

```
public GamePlayer GetGamePlayerView(int idGamePlayer)
{
   return FindAll(source => source.Include(gamePlayer => gamePlayer.Ships)
                                .ThenInclude(ship => ship.Locations)
                            .Include(gamePlayer => gamePlayer.Salvos)
                                .ThenInclude(salvo => salvo.Locations)
                            .Include(gamePlayer => gamePlayer.Game)
                                .ThenInclude(game => game.GamePlayers)
                                    .ThenInclude(gp => gp.Player)
                            .Include(gamePlayer => gamePlayer.Game)
                                .ThenInclude(game => game.GamePlayers)
                                    .ThenInclude(gp => gp.Salvos)
                                        .ThenInclude(salvo => salvo.Locations)
                                .Include(gamePlayer => gamePlayer.Game)
                                    .ThenInclude(game => game.GamePlayers)
                                        .ThenInclude(gp => gp.Ships)
                                            .ThenInclude(ship => ship.Locations)
                                )
        .Where(gamePlayer => gamePlayer.Id == idGamePlayer)
        .OrderBy(game => game.JoinDate)
        .FirstOrDefault();
}
```

Las líneas :

```
.Include(gamePlayer => gamePlayer.Game)
    .ThenInclude(game => game.GamePlayers)
        .ThenInclude(gp => gp.Salvos)
            .ThenInclude(salvo => salvo.Locations)
.Include(gamePlayer => gamePlayer.Game)
    .ThenInclude(game => game.GamePlayers)
        .ThenInclude(gp => gp.Ships)
            .ThenInclude(ship => ship.Locations)
```

Se deben a que en el siguiente punto se deberá ir un nivel hacia atras en la iteración de los datos para obtener el game del gamePlayer que se solicita, luego obtener los gamePlayer para así tener los salvos de los dos y poder armar el listado de salvos, si no tenemos los includes la consulta en la base de datos no retornará esos campos.

- Crear los DTO de las nuevas entidades siguiendo los lineamientos de Ship (cuida los tipos de datos), ten en cuenta que el DTO de Salvo debe tener el campo Player de tipo (PlayerDTO).
- Modifcar el DTO GameViewDTO para agregar el campo Salvos (cuida el tipo de dato SalvoDTO)
- Actualizar el método GetGameView de la clase GamePlayersControllers para que llenar el nuevo campo del DTO GameViewDTO llamado Salvos, teniendo en cuenta que se deben ver los salvos de ambos jugadores (similar al campo gamePlayers). Lee sobre el método SelectMany para retornar solo una lista y no una lista sobre otra en https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.selectmany?view=netcore-2.2

Ejecuta la aplicación, una vez se haya ejecutado el proyecto coloca en el navegador la url:

https://localhost:5001/api/gamePlayers/1

Se debe ver el JSON con la vista del juego como se indicó al principio.

Definir Salvo - Envía el proyecto en un .zip

web-mostrar-juego-2

Mostrar la vista del juego con salvos

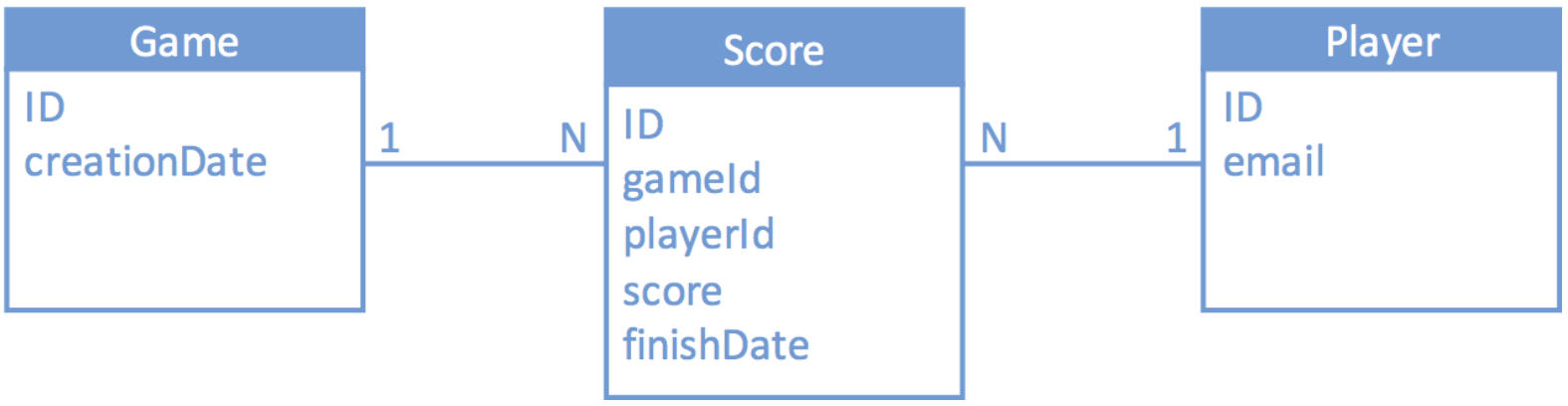
Descargar el paquete web-mostrar-juego-2 y copiar su contenido en la carpeta wwwroot (sobrescribir todo), una vez copiado el contenido en la carpeta ejecutar el proyecto y colocar en el navegador la url https://localhost:5001/game.html?gp=1 se debe ver la página del juego con los barcos posicionados y los salvos posicionados.

email task5

Definir entidad Score

Se necesita tener un tablero de líderes en la aplicación para ello se tiene que implementar:

- Una tabla de base de datos con el puntaje de cada jugador del juego y cuándo terminó el juego
- Cambiar el controlador GamesController para incluir puntuaciones en la lista de datos de juegos
- Mostrar el listado en la aplicación



Esta es otra oportunidad para aplicar y adquirir fluidez con la base de datos y los conceptos en bucle de la tarea anterior.

Se debe tener en cuenta que esta tarea debe trabajar tanto con Players como con GamePlayers. Un Player en un Game obtiene un puntaje, pero si el puntaje se guarda en la base de datos de esa manera, entonces para hacer la tabla de líderes, el código tendría que obtener todos los GamePlayers del juego, lo que significa todos los Games y Players, solo para obtener una lista de puntajes.

Ya que con anterioridad se ha creado una entidad simirlar(GamePlayer) deberías poder crear la nueva entidad Score sin problemas:

- Crear la clase Score, recuerda las relaciones con Player y con Game, ya que en C# no se permite que la clase tenga el mismo nombre de un campo, se cambiará el nombre del campo score por point (ya que point puede ser 1, 0 y 0.5 el tipo de dato será double)
- Agregarla en el contexto de la base de datos SalvoContext
- Ejecutar los comandos para crear el migration addScoreEntity y actualizar la base de datos
- Crear la data de prueba (deberás obtener los games y players para poder crear los scores)
- Ejecutar la aplicación para que se creen los datos

Actualizar el controller

Ya que los datos se encuentran disponibles se deben mostrar en el JSON que retorna el método Get del controlador GamesController. Específicamente, cuando se crea el JSON para listar los Games, se crea un JSON para cada Game. Cuando se crea el JSON del Game, se crea un JSON para cada GamePlayer del Game. Este es el código en el controlador que se debe cambiar para agregar un nuevo campo con la puntuación para el Game, si es que hay una puntuación para el Game en la base de datos.

Probablemente se necesitará agregar dos métodos a sus clases de entidades:

- Un método **GamePlayer.GetScore()** que devuelve un puntaje que el controlador puede llamar para obtener el puntaje, si existe alguno. null significa que el juego no está terminado.
- Un método **Player.GetScore(Game)** que devuelve un puntaje al que puede llamar GamePlayer.

Será necesario actualizar el método GetAllGamesWithPlayers de GameRepository para que retorne por cada Player el listado de Score.

Para buscar los Scores en un Player y retornar el del Game indicado, si es que lo hay, o retornar nulo, vea un método útil en las listas llamado FirstOrDefault() en https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.firstordefault?view=netcore-2.2, también será util utilizar el operador ternario cuando se cree el DTO de GamePlayer https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/conditional-operator, se debe tener en cuenta que se debe modificar el DTO gamePlayerDTO agregando un campo llamado Point nullable (double? Point) lee sobre ello en https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/nullable-types/

Ejecuta la aplicación, luego colocar en la url:

https://localhost:5001/api/games

Se debe mostrar el JSON de los juegos con la información del puntaje en cada gamePlayer.


```
[{
    "id":1
    ,"creationDate":"2019-01-01T00:23:39.8783606"
    ,"gamePlayers":[
        {
        "id":1,"joinDate":"2019-01-01T00:26:22.6004604"
        ,"player":{"id":1,"email":"j.bauer@ctu.gov"}
        ,"point":1.0
        }
    ]
}]
```


Definir Score - Envía el proyecto en un .zip

web-listar-juegos-2

Mostrar el listado de juegos en una página web

Descarga el paquete web-listar-juegos-2 y copiar su contenido en la carpeta wwwroot. Una vez copiado el contenido en la carpeta ejecutar el proyecto y colocar en el navegador la url *https://localhost:5001/index.html* se debe ver la página principal del juego Salvo con la tabla de lideres.

SECCIÓN ANTERIOR
Actividad 4 - Configurar el contexto de la base de datos

SIGUIENTE SECCIÓN
Actividad 6 - Implementar la seguridad

Ir a...

Mantente en contacto

MindHub

Resumen de retención de datos

einiciar tour para usuario en esta página