1. Setting Up the Projects

1. Install VSCode and coding frameworks. Kick one of the following links. Windows: Install the .NET Coding Pack - Windows Mac: Install the .NET Coding Pack - macOS. Or go to the following page and click the corresponding buttons.

   https://code.visualstudio.com/docs/csharp/get-started

2. Create a new folder named Todos for the solution.
3. Open the folder in VSCode.
4. Open a Terminal window for the solution folder.
5. Install Entity Framework's tools if you haven't already.

   ```
   dotnet tool install --global dotnet-ef
   ```

6. Create a class library project named *Todos.Data*.

   ```
   dotnet new classlib --name Todos.Data
   ```

7. Create a class library project named *Todos.Common*.

   ```
   dotnet new classlib --name Todos.Common
   ```

8. Create an API project named *Todos.MiniAPI*.

   ```
   dotnet new webapi -minimal --name Todos.MiniAPI
   ```

9. Open a terminal window for the *Todos.Data* folder and install EF's and AutoMapper's NuGet packages.

   ```
   dotnet add package Microsoft.EntityFrameworkCore.SqlServer
   ```

   ```
   dotnet add package Microsoft.EntityFrameworkCore.Tools
   ```

   ```
   dotnet add package AutoMapper
   ```

10. Add a reference to the *Todos.Common* project.

    ```
    dotnet add reference ..\Todos.Common\Todos.Common.csproj
    ```

11. Open a terminal window for the *Todos.MiniAPI* folder and install EF's and AutoMapper's NuGet packages.

    ```
    dotnet add package Microsoft.EntityFrameworkCore.Design
    ```

    ```
    dotnet add package AutoMapper
    ```

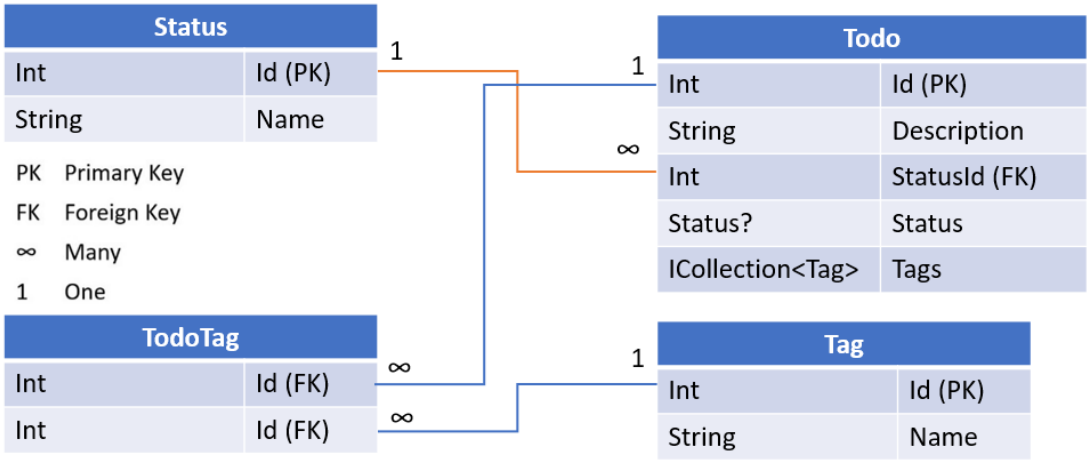
12. Add a reference to the *Todos.Data* project.

    ```
    dotnet add reference ..\Todos.Common\Todos.Common.csproj
    ```

2. Entity Framework

Adding the Entities

1. Add a folder named Entities to the *Todos.Data* project.
2. Add three entity classes according to the schema below.



3. Add a file named *Usings.cs* for global usings to necessary namespaces in the installed and linked class libraries.

```
global using System.Collections.Generic;

global using System.Threading;

global using Microsoft.EntityFrameworkCore;

global using System.Linq.Expressions;

global using AutoMapper;

global using TodoMiniAPI.Data.Entities;
```

4. Execute the **dotnet build** command for the *Todos.Data* project and fix any errors.

Adding the EF Database Context to the Data project

For EF to recognize the entity tables as entities, you must configure them in a database context class that EF uses to create the migration script that creates the database.

1. Add a folder named *Contexts* to the *Todos.Data* project.
2. Add a **TodoContext** class to the folder and the **Todos.Data.Contexts** namespace.
3. Add using statements to the **Todos.Data.Entities** and **Microsoft.EntityFramework** to gain access to their classes.
4. Add **DbSets** for the **Todo**, **Status**, and **Tag** tables, not the **TodoTag** table, which EF creates automatically.
5. Add a constructor with a **DbContextOptions** parameter.

```
public TodoContext(DbContextOptions<TodoContext> options) :
base(options) { }
```

6. Override the **OnModelCreating** method with a **protected** access modifier.
7. Configure the many-to-many relationship between the **Todo** and **Tag** tables.

```
builder.Entity<Todo>()
        .HasMany(e => e.Tags)
        .WithMany(e => e.Todos)
        .UsingEntity<TodoTag>();
```

8. To load an entity's related data, you configure navigation properties in the **OnModelCreating** method.

```
builder.Entity<Todo>()
    .Navigation(e => e.Tags)
    .UsePropertyAccessMode(PropertyAccessMode.Property);


builder.Entity<Tag>()
    .Navigation(e => e.Todos)
    .UsePropertyAccessMode(PropertyAccessMode.Property);
```

9. Add a global using to the **Context** namespace in the *Usings.cs* file.

```
global using Todos.Data.Contexts;
```

10. Execute the **dotnet build** command for the *Todos.Data* project and fix any errors.

Adding EF to the API project

You need an executable program (the MiniAPI project) to add the migration and create the database. To achieve this, you must add a connection string to the *appsettings.json* file and configure EF in its *Program.cs* file.

1. Add a connection string for the database.

```
"ConnectionStrings": {
 "TodosConnection": "Server=(localdb)\\mssqllocaldb;Database=TodosDb"
},
```

2. Add configuration for EF above the **builder.Build** method call in the *Program.cs* file.

```
// SQL Server Service Registration
builder.Services.AddDbContext<TodoContext>(
    options =>
```

```
        options.UseSqlServer(builder.Configuration.GetConnectionString(
"TodosConnection")));
```

3. Add a file named *Usings.cs* for global usings to necessary namespaces in the installed and linked class libraries.

```
global using AutoMapper;

global using Microsoft.EntityFrameworkCore;

global using Todos.Data.Contexts;

global using Todos.Data.Entities;
```

4. Execute the **dotnet build** command for the *Todos.MiniAPI* project and fix any errors.
5. Open the Todos.MiniAPI.csproj file and remove this setting to avoid errors when creating the database.

```
<InvariantGlobalization>true</InvariantGlobalization>
```

6. Exec the **dotnet ef** command for the *Todos.MiniAPI* project to create the migration script in the *Todos.Data* project.

```
dotnet ef migrations add Initial -p ../Todos.Data
```

7. Check that the migration was created in the added **Migrations** folder in the *Todos.Data* project.
8. Exec the **dotnet ef** command for the *Todos.MiniAPI* project to run the migration script in the *Todos.Data* project to generate the database.

```
dotnet ef database update
```

9. Install the SQL Server (mssql) extension (if you haven't already.)
    a. Open the extensions tab in Visual Studio Code.
    b. Search for SQL Server (mssql).
    c. Click its Install link.
10. Open the SQL Server extension.
    1. Click the plus (+) button in the Connections field.
    2. Paste in the server's name from the connection string and remove one backslash.
11. (localdb)\mssqllocaldb
    1. Press enter twice.
    2. Select **Integrated** in the selection list and press enter a last time.
    3. Click on the SQL server in the connection list.
    4. Click the *Databases* node and then the *TodosDb* node to open the database.
    5. Click the *Tables* node to list all tables in the *TodosDb* database.

6. The *__EFMigrationHistory* table keeps track of changes to the database from subsequent migrations. Change nothing in this table.

Adding DTOs

DTOs are shared between the API and the Web Application to have a common base when transferring objects over the internet. They are, therefore, added to the *Todos.Common* project, which is referenced from both projects and the *Todos.Data* project. Deleting an entity doesn't require a DTO, as the entity id is passed as a parameter to the API.

1. Add a folder named *DTOs* to the *Todos.Common* project.
2. Add a file named *TodoDTOs.cs* to the folder.
3. Add the namespace *Todos.Common.DTOs* to the file.
4. Add a class named **TodoPostDTO**, which sends data to the API when creating a new todo in the database. Add the **Description** and **StatusId** properties from their corresponding Entity class.
5. Add a class named **TodoPutDTO** that inherits the **TodoPostClass**, which sends data to the API when updating a todo in the database. Add the **Id** property from its corresponding Entity class.
6. Add a **TodoGetDTO** class that inherits the **TodoPutClass**, which fetches data from the API. Add the **Id** property from its corresponding Entity class.
7. Do the same for the **Tag**, **Status**, and **TodoTag** entities. **TodoTag** doesn't need a class for fetching data, as it only contains two ids.
8. Execute the **dotnet build** command for the *Todos.Common* project and fix any errors.

Adding the IEntity Interface

When working with generics, the generic type **TEntity** can represent any object. Therefore, you can't get to the object id directly. One way around this is to use an interface, a contract that requires one or more properties to exist in all objects represented by the generic types.

1. Add a file named *IEntity.cs* to the *Entities* folder in the *Todos.Data* project.
2. Add the **Todos.Data.Entities** namespace.
3. Add a public interface named **IEntity**.
4. Add an **int** property named **Id**.
5. Implement the interface in the **Todo**, **Tag**, and **Status** entity classes.
6. Execute the **dotnet build** command for the *Todos.Data* project and fix any errors.

Configuring AutoMapper

To automatically convert between entity and DTO objects, you need to configure AutoMapper, which does it for you by calling its **Map** method. You add the configuration to the API project because it's the application using the database.

1. Open the *Usings.cs* in the *Todos.MiniAPI* project and add a global **using** to the **Common.DTOs** namespace.

   ```
   global using Todos.Common.DTOs;
   ```

2. Open the *Program.cs* in the *Todos.MiniAPI* project.

3. Add a method named **ConfigureAutoMapper** with an **IServiceCollection** parameter named **services** at the end of the file (above the **WeatherForecast** record), registering AutoMapper's service so it can be injected through constructors.

   ```
   void ConfigureAutoMapper(IServiceCollection services) { }
   ```

4. Add a configuration object inside the method using AutoMapper's **MapperConfiguration** class. This object defines the possible object conversions between entities and DTOs.

   ```
   var config = new MapperConfiguration(cfg => { });
   ```

5. Add mappings between the **Todo** entity class and corresponding DTOs inside the configuration object's curly braces.

   ```
   cfg.CreateMap<Todo, TodoPostDTO>().ReverseMap();
   ```

   ```
   cfg.CreateMap<Todo, TodoPutDTO>().ReverseMap();
   ```

   ```
   cfg.CreateMap<Todo, TodoGetDTO>().ReverseMap();
   ```

6. Add mappings between the rest of the entity and DTO classes.

7. Create an AutoMapper mapping below the configuration object.

   ```
   var mapper = config.CreateMapper();
   ```

8. Register the mapping as a service to inject it through constructors.

   ```
   services.AddSingleton(mapper);
   ```

9. Call the method above the **builder.Build** method call earlier in the file.

   ```
   ConfigureAutoMapper(builder.Services);
   ```

Adding the Generic DbService

You add a generic **DbService** class to reuse the CRUD code for all entities to avoid code duplication. It needs to be asynchronous because the Entity Framework methods are. The service can be injected through constructors to reuse the code and get an object served from the runtime instead of creating it with the **new** keyword. You inject the **TodoContext** class and the **IMapper** interface to gain access to the database and AutoMapper to convert objects. No records are added, updated, or removed from the database before the **_db.SaveChangesAsync** method is called.

1. Add a folder named *Services* in the *Todos.Data* project.

2. Add a file named *DbService.cs* to the *Services* folder in the *Todos.Data* project.

3. Add the **Todos.Data.Services** namespace.

4. Add a public class named **DbService**.

5. Inject the **TodoContext** class and the **IMapper** interface through a constructor and save the objects in class-level **private** variables named **_db** and **_mapper** to have access to them in all methods in the class. You make the variables private to encapsulate them inside the class and restrict their accessibility to the class only.

6. Add an asynchronous method named **SingleAsync** that returns one record from the database with an **id** parameter.

7. **TEntity** is the EF entity class representing the table from which you want to fetch data.

8. **TDto** is the data transfer (POCO) class returning the data to the client requesting the data through the API.

9. Restrict **TEntity** to only classes implementing the **IEntity** interface and **TDto** to classes.

10. **async Task<TDto>** specifies that the method is asynchronous and returns a **TDto** object, which can be any entity class converted to a DTO.

    ```
    public async Task<TDto> SingleAsync<TEntity, TDto>(int id) where
    TEntity : class, IEntity where TDto : class
    ```

    ```
    {
    ```

    ```
    }
    ```

11. Inside the **SingleAsync** method, Use the EF database object **_db** to access the data with the **SingleOrDefault** LINQ method.

    ```
    var entity = await _db.Set<TEntity>().SingleOrDefaultAsync(e => e.Id
    == id);
    ```

12. Use Auto Mapper's **IMapper** interface's **Map** method.

    ```
    return _mapper.Map<TDto>(entity);
    ```

13. Add an asynchronous **GetAsync** method, similar to **SingleAsync**, that returns all records as a **List** collection for the table represented by the generic **TEntity** type with the **ToListAsync** LINQ method.

14. Add an asynchronous **AddAsync** method, similar to **SingleAsync**, that adds and returns the object represented by the generic **TEntity** type as a record to the table **TEntity** represents. First, you use AutoMapper to convert the DTO to an entity and then call the **AddSync** LINQ method (the reverse order from previous methods.)

15. Add a *synchronous* method named **Update**, identical to the **AddAsync** method that calls the LINQ Update instead. Replace **async Task<TEntity>** with **void** as it won't return any data. It's *synchronous* because it updates an object that already exists in memory.

16. Add an asynchronous **DeleteAsync** method similar to **SingleAsync**. It only specifies the entity to remove with the provided **id** parameter and returns a **bool** indicating whether the record was removed.

17. Fetch the record with the **SingleOrDefault** LINQ method and the value in the **id** parameter and store it in a variable named **entity**.

18. Return **false** if the **entity** variable is **null** (no matching record was found.)

19. Call the **_db.Remove** method to mark the record for deletion in the database.

20. Add a parameterless asynchronous **SaveChangesAsync** method that returns a **bool** indicating whether the record's changes were saved. The **_db.SaveChanges** method you call returns 1 if successful, 0 if nothing was changed, and -1 if unsuccessful.

```
public async Task<bool> SaveChangesAsync() => await
_db.SaveChangesAsync() >= 0;
```

21. Add a **using** statement for the **Todos.Data.Services** namespace in the *Usings.cs* file.

22. Open the *Program.cs* in the *Todos.MiniAPI* project.

23. Add a method named **RegisterServices** with an **IServiceCollection** parameter named **services** at the end of the file (above the **WeatherForecast** record.)

24. Register the **DbService** class as a service inside the method so that the runtime can serve up objects of it when the class is injected into a constructor. The **AddScoped** method creates a new object for each call to the API.

```
void RegisterServices(IServiceCollection services)
{
    services.AddScoped<DbService>();
}
```

25. Call the **RegisterServices** method above the **builder.Build** method call.

26. Open the *Usings.cs* in the *Todos.MiniAPI* project and add a global **using** to the **Data.Services** namespace.

```
global using Todos.Data.Services;
```

27. Execute the **dotnet build** command for the *Todos.Data* project and fix any errors.

3. The Mini API

You can register the API's HTTP endpoints in the *Program.cs* file, but if you have many endpoints, it can get messy quickly. To avoid this, you can create reusable services for each entity the API handles, which is what we'll implement.

Configure CORS

You must enable CORS (Cross-Origin Resource Sharing) to allow applications to call the API.

1. Open the *Porgram.cs* file in the *Todos.MiniAPI* project to configure access for any application (normally, you only give access to specific domains or subdomains.) Add the configuration above the **builder.Build** method call.

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

2. Use the previous configuration when adding the CORS middleware above the **app.Run** method call.

```
app.UseCors("CorsAllAccessPolicy");
```

Adding the ApiExtensions Class

Instead of duplicating code in all the endpoint classes you add later, reuse the endpoint registration code through extension methods that call the database when an endpoint is called from another application.

1. Add a folder named *Extensions* to the *Todos.MiniAPI* project.
2. Add the **Todos.MiniAPI.Extensions** to the file.
3. Add a **public static** class named **ApiExtensions** to the folder. Extension methods must be placed in a **public static** class.
4. Add a global **using** to the namespace in the *Usings.cs* file.

Adding the HttpSingleAsync Method to the ApiExtensions Class

IResult, the method's return type, is a special interface that returns a response object with the relevant data (applicable) from the API.

1. Add a folder named *Extensions* to the *Todos.MiniAPI* project.
2. Add the **Todos.MiniAPI.Extensions** namespace to the file.

3. Add a **public static** class named **ApiExtensions** to the folder. Extension methods must be placed in a **public static** class.

4. Add an asynchronous generic **public static** method named **HttpSingleAsync** defined by the generic **TEntity** and **TDto** types and return an **IResult** to the class.

    a. Pass in an object of the **DbService** class to gain access to the database.

    b. Pass in the resource's (a record's) id to fetch as an int parameter.

```
public static async Task<IResult> HttpSingleAsync<TEntity, TDto>(
DbService db, int id) where TEntity : class, IEntity where TDto :
class { }
```

5. Await the result from a call to the **DbService's SingleAsync** method inside the **HttpSingleAsync** method to fetch the resource from the database. Store the resource in a variable named **result**.

6. Return the HTTP not found (404) result by returning the result from a call to the predefined **Results.NotFound** response method.

7. Return the record from the **result** variable by returning a call to the **Results.Ok** method, passing it the **result** variable.

8. Add the **Todos.MiniAPI.Extensions** namespace as a global using in the *Usings.cs* file.

9. Execute the **dotnet build** command for the *Todos.MiniAPI* project and fix any errors.

Adding the Register Extension Method to the ApiExtensions Class

This extension method registers the individual API endpoints (URIs) called by other applications.

1. Add a generic **public static** method named **Register,** defined by the generic types **TEntity**, **TPostDto**, **TPutDto**, and **TGetDto**, representing the entity and its corresponding DTOs when called. It has one **WebApplication** parameter named **app** used to register the endpoint with the API.

```
public static void Register<TEntity, TPostDto, TPutDto, TGetDto>
(this WebApplication app) where TEntity : class, IEntity where
TPostDto : class where TPutDto : class where TGetDto : class { }
```

2. Use reflection to fetch the name of the entity class use it in the endpoint URI path when registering it.

```
var node = typeof(TEntity).Name.ToLower();
```

3. Register an HTTP Get node that fetches one record from the database by calling the **HttpSingleAsync** method with the generic **TEntity** and **TGetDto** types. You cannot add the URIs *{id}* parameter in the interpolated string because the compiler would interpret it as if you wanted to inject a value from a variable named **id**, which isn't the case. Try it and see what happens.

```
app.MapGet($"/api/{node}s/" + "{id}", HttpSingleAsync<TEntity,
TGetDto>);
```

4. Execute the **dotnet build** command for the *Todos.MiniAPI* project and fix any errors.

Adding the IEndpoint Interface

This interface ensures all entity services have a **Register** method that registers its endpoints with the API.

1. Add a folder named *Interfaces* to the *Todos.MiniAPI* project.
2. Add a **public** interface named **IEndpoint**.
3. Add the **Todos.MiniAPI.Interfaces** above the interface.
4. Add a **void** method named **Register** with a **WebApplication** parameter named **app**. This parameter will give access to the application's registration process to register the entity endpoints callable from other applications.
5. Add a global **using** to the namespace in the *Usings.cs* file.
6. Execute the **dotnet build** command for the *Todos.MiniAPI* project and fix any errors.

Adding the StatusEndpoint API Service

This class is added as a service with the API to register HTTP endpoints for the Status entity so that other applications can make HTTP requests to the API and retrieve or modify data in the database table corresponding to the **Status** entity.

1. Add a folder named *Endpoints* to the *Todos.MiniAPI* project.
2. Add a **public** class named **StatusEndpoint**.
3. Add the **Todos.MiniAPI.Endpoints** above the class.
4. Add a **void** method named **Register** with a **WebApplication** parameter named **app**. This parameter will give access to the application's registration process to register the entity endpoints callable from other applications.
5. Add a global **using** to the namespace in the *Usings.cs* file.
6. Execute the **dotnet build** command for the *Todos.MiniAPI* project and fix any errors.

Registering the StatusEndpoint Class as a Service

To expose the endpoint to other applications calling the API, it must be registered with the API application in the *Program.cs* file.

The **AddTransient** method used when registering the endpoints as services creates a new object each time one is requested. **AddScoped,** on the other hand, remains for the duration of a complete request and response cycle.

The **IEndpoint** ensures objects of the **StatusEndpoint** class and other endpoint classes you add later.

1. Open the *Program.cs* file in the *Todos.MiniAPI* project.
2. Register the **IEndpoint** interface with the **StatusEndpoint** using the **AddTransient** method inside the **RegisterServices** method.

```
services.AddTransient<IEndpoint, StatusEndpoint>();
```

3. Add a method named **RegisterEndpoints** with a **WebApplication** parameter named **app** below the **RegisterServices** method.

4. To register the HTTP endpoints for all transient services that implement the **IEndpoint** interface, you must first find them by calling the **GetServices** method.

```
var endpoints = app.Services.GetServices<IEndpoint>();
```

5. Loop over the services in the **endpoints** variable and throw an **InvalidProgramException** if the current endpoint is **null**. If one is **null**, its HTTP endpoints are unavailable, and the application doesn't work properly and shouldn't start.

6. Inside the loop, call the **Register** method (defined by the **IEndpoint** interface and implemented in the **StatusEndpoint** class) on the service object in the loop.

7. Call the **RegisterEndpoints** method ***below*** the **builder.Build** method call earlier in the same file.

```
RegisterEndpoints(app);
```

8. Execute the **dotnet build** command for the *Todos.MiniAPI* project and fix any errors.

9. Execute the **dotnet run** command for the *Todos.MiniAPI* project to start the API.

10. Hold down **Ctrl** (**Command** on a Mac), click the URL in the **Terminal** window, or copy it and open it in a browser. This displays an empty page.

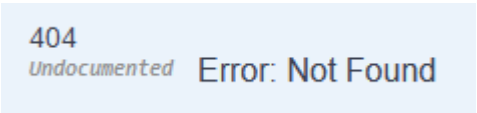
11. Add the URI */swagger/index.html* to the URL in the browser. This opens the Swagger page where you can test the API so far.

12. Click the blue area around the **Get** method for the Status endpoint.

13. Click the **Try it out** button and enter a positive number in the **id** textbox.

14. Click the blue **Execute** button to call the API from the browser whose Swagger page acts as a proxy for an application calling the API.

15. You should get a *404 Not Found* message, as the database has no data yet. You might recall that you implemented a call to the **NotFound** method in the **HttpSingleAsync** method for this purpose.



16. Close the browser or the APIs tab.

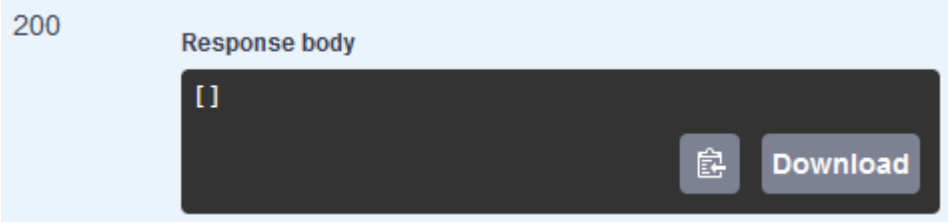
17. Press **Ctrl-C** (**Command+C**) in the **Terminal** window to stop the API application.

18. Add an **HttpGetAsync** method, similar to **HttpSingleAsync**, to the **ApiExtensions** class that returns all statuses by calling the **db.GetAsync** method in the **DbService** class.

19. Add a method call to the **app.MapGet** method in the **Register** method to add the get endpoint that returns all records in a table.

```
app.MapGet($"/api/{node}s", HttpGetAsync<TEntity, TGetDto>);
```

20. Start the API again and try the new **Get** endpoint. It should return *200 OK* and an empty list.



21. Close the browser tab and stop the API with **Ctrl-C** (**Command+C**) in the **Terminal** window.

22. Add an **HttpPostAsync** method, similar to **HttpGetAsync**, to the **ApiExtensions** class, but it has a second **TPostDto** parameter, which is the DTO object sent from the application calling the API.

    a. Add a try/catch block.

    b. Add a variable named **entity** that collects the object (record) returned from a call to the **db.AddAsync** method in the **DbService** class inside the try block.

    c. Below, call the **SaveChangesAsync** method in the **DbService** class inside an if statement to save the record in the database.

    d. Inside the if block, return **Result.Created** (*201 Created*) with the URI to the created resource.

```
if (await db.SaveChangesAsync())
{
    var node = typeof(TEntity).Name.ToLower();
    return Results.Created($"/{node}s/{entity.Id}",
    entity);
}
```

    e. Below the catch block, return **Results.BadRequest** with the name of the entity.

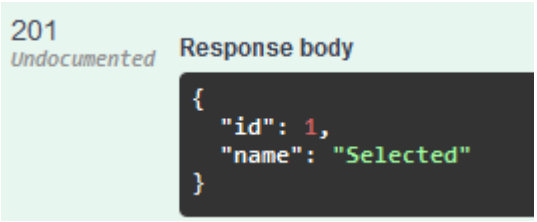
```
return Results.BadRequest($"Couldn't add the
{typeof(TEntity).Name} entity.");
```

23. Add a method call to the **app.MapPost** method in the **Register** method to add the status post endpoint.

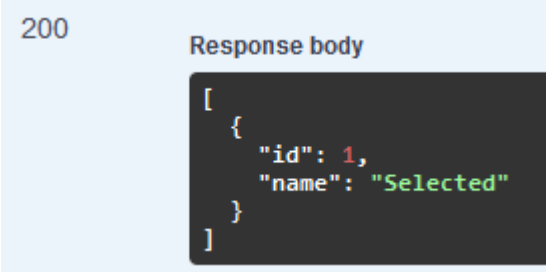
```
app.MapPost($"/api/{node}s", HttpPostAsync<TEntity, TPostDto>);
```

24. Start the API again and try the new **Post** endpoint. It should return *201 Created* and the created record. Enter a value for the **name** parameter.

```
{
    "name": "Selected"
}
```



25. Execute the two **Get** endpoints to verify the status was added to the database.



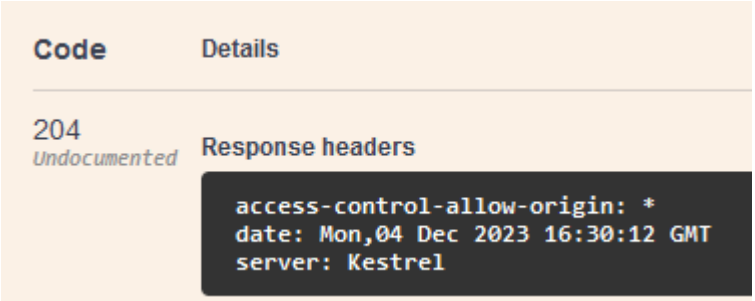
26. Click the SQL Server menu option in VSCode's sidebar menu.
    a. Expand the database under the **Connections** section.
    b. Expand the **Databases**, **TodosDb**, and **Tables** nodes.
    c. Right-click on the **Statuses** table and click the **Select Top 1000** option in the context menu.
    d. The record you added should be displayed in a table grid.
    e. Close the SQL Server tab in VSCode.
27. Add a **HttpPutAsync** method in the **ApiExtensions** class that uses **db.Update** and **db.SaveChanges** to update a record in the database. The method throws the same error as the **Post** method and returns **Results.NoNontent** (*204 No Content*) if successful.
28. Add a method call to the **app.MapPut** method in the **Register** method to add the status put endpoint.

```
app.MapPut($"/api/{node}s/" + "{id}", HttpPutAsync<TEntity,
TPutDto>);
```

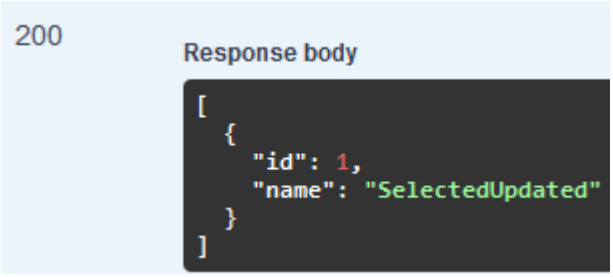
29. Start the API again and try the new **Put** endpoint. It should return *204 No Content* and the updated record. Enter the new value for the **name** parameter and the record's **id**.

```
{
  "name": "SelectedUpdated",
  "id": 1
```

}



30. Execute one of the **Get** endpoints to verify the updated value.



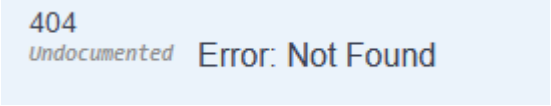
31. Add a **HttpDeleteAsync** method in the **ApiExtensions** class that uses **db.Delete** and **db.SaveChanges** to remove a record from the database. The method throws the same error as the **Put** method and returns **Results.NoNontent** (*204 No Content*) if successful. This method doesn't need a DTO as it won't return any data.

32. Add a method call to the **app.MapPut** method in the **Register** method to add the status put endpoint.

```
app.MapDelete($"/api/{node}s/" + "{id}", HttpDeleteAsync<TEntity>);
```

33. Start the API again and try the new **Delete** endpoint. It should return *204 No Content* and remove the record. Enter the id for the **id** parameter for the record you want to remove.

34. Execute one of the **Get** endpoints to verify the updated value.



35. Stop the API.
36. Add endpoints classes in the *Endpoints* folder for the remaining entities. The **TodoTagEndpoint** is a special case as it doesn't have a corresponding **TodoTagGetDTO** class, nor does the TodoTag entity have a dedicated **Id** primary key property. You could alter the entity to have an **Id** property as a primary key, remove the composite primary key configuration in the context class, and update the database. This change would open up for duplicate tags. Or, you could add special **Http** methods in the **TodoTagEndpoint** class that bypasses the regular registration flow.
    a. Register the classes in the **RegisterServices** method in the *Program.cs* file.
    b. Restart the API and test the endpoints for each added class.
37.