Hands-on lab

Lab: Entity Framework

September 2020

Exercise 1: The Star Wars API

Visit https://swapi.dev and familiarize yourself with this API.

Exercise 2: Build a Console application to clone the Star Wars movies

Step 1 - Build the EF model

Create a new Solution with a Console App (.NET Core) project called *StarWarsUniverseClone*.

Add a Class Library (.NET Standard) called *StarWarsUniverse.Domain*.

1. Add a library to the Domain project via NuGet
   `install-package newtonsoft.json`

   > Note: newtonsoft.json references the JSON.NET project, a hugely popular library for serializing objects from/to JSON strings.

2. Create a the following classes in the model project:

C#

```
using System;
using Newtonsoft.Json;

namespace StarWarsUniverse.Domain
{
    public abstract class Resource
    {
        public DateTime Created { get; set; }
        public DateTime Edited { get; set; }

        [JsonProperty(PropertyName = "url")]
        public string Uri { get; set; }
    }
}
```

   > Note: [JsonProperty(PropertyName = "url")] maps the json attribute *url* to the property *Uri*

3. Add another class

C#

```
using Newtonsoft.Json;
using System;

namespace StarWarsUniverse.Domain
```

```
{
    public class Movie : Resource
    {
        public string Title { get; set; }

        [JsonProperty(PropertyName = "episode_id")]
        public int EpisodeId { get; set; }

        [JsonProperty(PropertyName = "opening_crawl")]
        public string OpeningCrawl { get; set; }

        public string Director { get; set; }

        public string Producer { get; set; }

        [JsonProperty(PropertyName = "release_date")]
        public DateTime ReleaseDate { get; set; }
    }
}
```

4. Add another Class Library (.NET Standard) project *StarWarsUniverse.Data* to the solution.

5. Add the Entity Framework Core package for SQLServer to the Data project.
   `install-package Microsoft.EntityFrameworkCore.SqlServer`

6. Create a class called *StarWarsContext*, defining a *DbSet* of *Movie* objects. Furthermore, define ResourceUri to be PK using Fluent API. The database created should be named *StarWarsDB*.

> Tip: use OnModelCreating for configuring the PK and OnConfiguring for naming the generated DB.

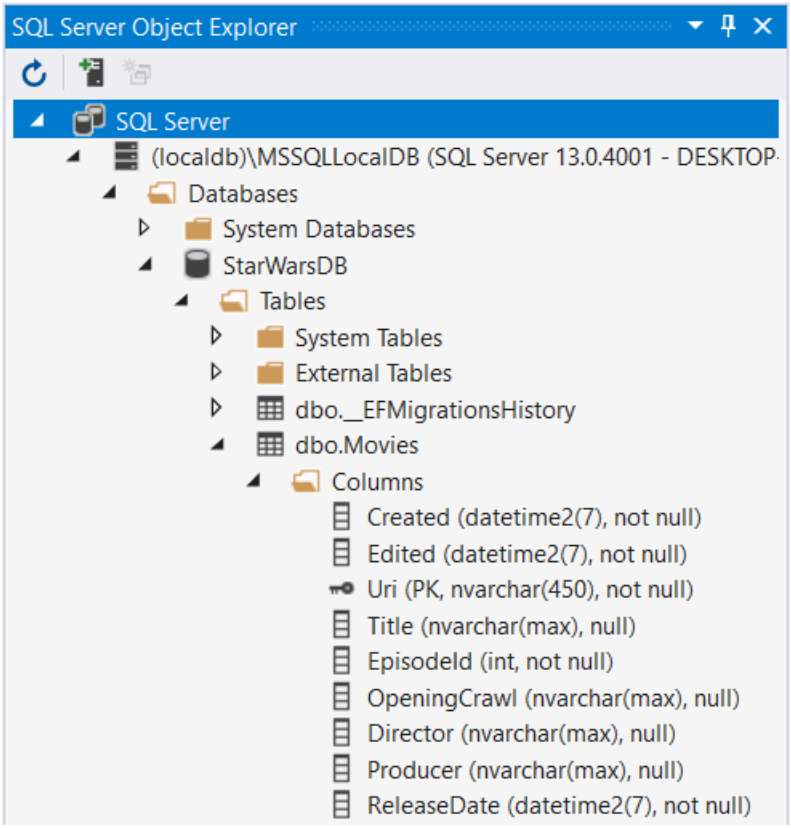
C# (incomplete)

```
public class StarWarsContext : DbContext
{
    ...
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Movie>().HasKey(m => m.Uri);
    }
}
```

Step 2 - Create an Initial Migration

1. First install the necessary EF tools in the Console project:
   `install-package microsoft.entityframeworkcore.tools`

2. Create a migration called “Initial” and create the database. This command will not work for the first time. Solve this problem and create the migration. Run it to create the database and verify it. Make sure the Data project is selected as the default project in the package manager console.



Tip: also have a look at the generated code in the folder Migrations. Familiarize yourself with it!

Step 3 – Retrieve data from the public API

1. Create a folder called *Repositories* (in the Data project) and define the following interface:

C#

```
using StarWarsUniverse.Domain;
using System.Collections.Generic;

namespace StarWarsUniverse.Data.Repositories
{
    public interface IMovieRepository
    {
        IList<Movie> GetAllMovies();
    }
}
```

2. The code that will call the swapi makes use of the HttpClient-class, which is located in the namespace System.Net.Http and converts the JSON-string to objects using JsonConvert in Newtonsoft.Json. Both namespaces can be added without installing any additional NuGet packages. Create a folder called *Api* in the *Repositories* folder.
   Now write a class in the *Api* folder called *MovieApiRepository* that implements this interface.

C#

```
using Newtonsoft.Json;
using StarWarsUniverse.Domain;
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Net.Http.Headers;

namespace StarwarsUniverse.Data.Repositories.Api
{
    public class MovieApiRepository : IMovieRepository
    {
        private readonly HttpClient _httpClient;

        public MovieApiRepository()
        {
            _httpClient = new HttpClient { BaseAddress = new
Uri("http://swapi.dev") };
            _httpClient.DefaultRequestHeaders.Accept.Clear();
            _httpClient.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("application/json"));
        }

        public IList<Movie> GetAllMovies()
        {
            var url = "/api/films/";
            var allMovies = new List<Movie>();
            ResultsPage<Movie> resultsPage = null;

            HttpResponseMessage response = _httpClient.GetAsync(url).Result;
            if (response.IsSuccessStatusCode)
            {
                string content = response.Content.ReadAsStringAsync().Result;
                resultsPage =
JsonConvert.DeserializeObject<ResultsPage<Movie>>(content);
                allMovies = resultsPage.Results;
            }

            return allMovies;
        }
```

```
    }

    internal class ResultsPage<T>
    {
        public int Count { get; set; }
        public string Next { get; set; }
        public string Previous { get; set; }
        public List<T> Results { get; set; }
    }
}
```

> Note:
> The URL itself should end with a slash (/), otherwise the API returns an error (301 Moved Permanently).
> Hopefully in future versions of the API these issues will be resolved.

Step 4 – Write a Unit Test for the repository

Next we want to unit test our repository class. We will use NUnit.

1. Create a new Class Library (.NET Core) called StarWarsUniverse.Tests

> Note: a .NET Standard Library does not work here.

2. Add the following NuGet packages to this project

```
install-package NUnit
install-package NUnit3TestAdapter
install-package Microsoft.NET.Test.SDK
```

3. Now write a test for the MovieApiRepository. We will provide a skeleton here for you to complete.

C#

```
using NUnit.Framework;
using StarwarsUniverse.Data.Repositories.Api;

namespace StarWarsUniverse.Tests
{
    [TestFixture]
    public class MovieApiRepositoryTests
    {
        private MovieApiRepository _repo;

        [SetUp]
        public void SetUp()
        {
            _repo = new MovieApiRepository();
```

```
        }

        [Test]
        public void GetAllMovies()
        {
            //Act
            // TODO: call GetAllMovies

            //Assert
            Assert.Fail("No test yet: there should be a least 6 movies");
        }
    }
}
```

4. Run your test using Visual Studio, verify that your test succeeds.

> Note: the API is only aware of the original movies and the prequels (6 in total)

Step 5 – Seed with data

With the MovieApiRepository in place, we can now retrieve all movies and store them inside our own database, while preventing duplicates. Entity Core introduces a standard way of seeding your database using migrations. A quick introduction can be found in the official docs (go read it now):

https://docs.microsoft.com/en-us/ef/core/modeling/data-seeding

Once you added the entity objects via the HasData-method, you create a new migration which uses this info to create code to populate the database. We will do this now for the Movie class.

1. In OnModelCreating of StarWarsContext, add the following code:

C#

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Movie>().HasKey(m => m.Uri);

    IList<Movie> remoteMovies = new MovieApiRepository().GetAllMovies();
    modelBuilder.Entity<Movie>().HasData(remoteMovies.ToArray());
}
```

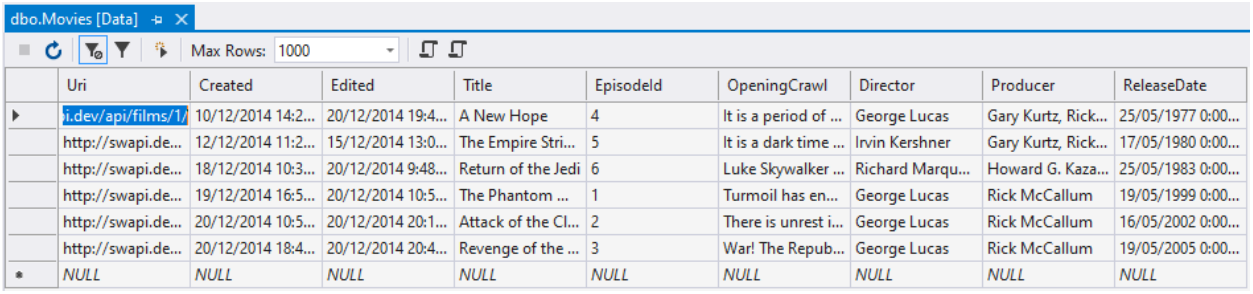
2. Rebuild your solution and verify that there are no compilation errors. Possibly you will have to import some namespaces.

3. Create a new migration called “clonemovies”

> Inspect the migration code for this migration. Do you understand what has happened?

4. Update your database to this new migration

5. Inspect the database, you should find 6 rows in the Movies table.


dbo.Movies [Data]

Max Rows: 1000

Uri	Created	Edited	Title	EpisodeId	OpeningCrawl	Director	Producer	ReleaseDate
i.dev/api/films/1/	10/12/2014 14:2...	20/12/2014 19:4...	A New Hope	4	It is a period of ...	George Lucas	Gary Kurtz, Rick...	25/05/1977 0:00...
http://swapi.de...	12/12/2014 11:2...	15/12/2014 13:0...	The Empire Stri...	5	It is a dark time ...	Irvin Kershner	Gary Kurtz, Rick...	17/05/1980 0:00...
http://swapi.de...	18/12/2014 10:3...	20/12/2014 9:48...	Return of the Jedi	6	Luke Skywalker ...	Richard Marqu...	Howard G. Kaza...	25/05/1983 0:00...
http://swapi.de...	19/12/2014 16:5...	20/12/2014 10:5...	The Phantom ...	1	Turmoil has en...	George Lucas	Rick McCallum	19/05/1999 0:00...
http://swapi.de...	20/12/2014 10:5...	20/12/2014 20:1...	Attack of the Cl...	2	There is unrest i...	George Lucas	Rick McCallum	16/05/2002 0:00...
http://swapi.de...	20/12/2014 18:4...	20/12/2014 20:4...	Revenge of the ...	3	War! The Repub...	George Lucas	Rick McCallum	19/05/2005 0:00...
NULL	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*

Step 6 – Retrieve the data from the database

1. Create a folder called *Db* in the *Repositories* folder.
   Now write a class *MovieDbRepository that also implements the IMovieRepository*. This repository should get its data from the database. The *StarwarsContext* can be injected as a parameter of the constructor.

2. Now write code in Program.cs that executes the following scenario:

   Print all movies, including title, episode id and release date, and sort them by episode id.
   The code should use the *MovieDbRepository* to retrieve the data.

   Example output:

```
=== Star Wars Movies ===
Episode 1 - The Phantom Menace
        Released: 19/05/1999
Episode 2 - Attack of the Clones
        Released: 16/05/2002
Episode 3 - Revenge of the Sith
        Released: 19/05/2005
Episode 4 - A New Hope
        Released: 25/05/1977
Episode 5 - The Empire Strikes Back
        Released: 17/05/1980
Episode 6 - Return of the Jedi
        Released: 25/05/1983
========================
```

Step 6 – Unit Test MovieDBRepository

EF Core makes it possible to create unit tests for classes that access a database. You can do this by using an SQLite in memory database. Go read about it here:

https://docs.microsoft.com/en-us/ef/core/miscellaneous/testing/sqlite

In order to apply the rules in the article, perform the following steps:

1. Add the following NuGet package to the StarWarsUniverse.Tests project:

```
install-package Microsoft.EntityFrameworkCore.Sqlite
```

2. Add the following constructors to the StarWarsContext class. This allows to inject options to change the database provider.

C#

```
public StarWarsContext() { }

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

3. Add an if-test to OnConfiguring that checks if there are already options configured. If not, use the SQLServer database.

C#

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer(
            "Server = (localdb)\\mssqllocaldb; Database = StarWarsDB;
Trusted_Connection = True; ");
    }
}
```

4. Create a class called StarWarsDbTestBase. This class creates an in memory database and ensures it is created before each test and cleaned up afterwards. This has the advantage to be very fast and repeatable.

C#

```
using Microsoft.Data.Sqlite;
using Microsoft.EntityFrameworkCore;
using NUnit.Framework;
using StarWarsUniverse.Data;

namespace StarWarsUniverse.Tests
{
    public class StarWarsDbTestBase
    {
        protected static SqliteConnection Connection;
        protected static StarWarsContext Context;

        [SetUp]
```

```
        public void BeforeEachTest()
        {
            Connection = new SqliteConnection("DataSource=:memory:");
            Connection.Open();
            Context = CreateContext();
            Context.Database.EnsureCreated();
        }

        [TearDown]
        public void AfterEachTest()
        {
            Context?.Dispose();
            Connection?.Close();
        }

        protected StarWarsContext CreateContext()
        {
             var options = new DbContextOptionsBuilder<StarWarsContext>()
                    .UseSqlite(Connection)
                    .EnableSensitiveDataLogging()
                    .Options;
             return new StarWarsContext(options);
        }
    }
}
```

5. Now write MovieDbRepositoryTests class using the following skeleton.

> Note: the unit test itself will be the same as for MovieApiRepository, but this time you check your data to be retrieved from a (mocked) database.

C#

```
using NUnit.Framework;
using StarWarsUniverse.Data;
using StarWarsUniverse.Data.Repositories.Db;

namespace StarWarsUniverse.Tests
{
    [TestFixture]
    public class MovieDbRepositoryTests : StarWarsDbTestBase
    {
        [Test]
        public void GetAllMoviesShouldReturnEveryMovie()
        {
            //Arrange
            var repo = new MovieDbRepository(Context);

            //Act
```

```
            var returnedMovies = repo.GetAllMovies();

            //Assert
            // Write your unit test here!
        }
    }
}
```

Exercise 3: Add Star Wars Planets to the model

In this exercise you will repeat the steps from exercise 2, but for the Planet class. In the end you will have to tables populated: Planets and Movies. However, the relationship between the two is not yet defined (this will be done in exercise 4).

> Less steps will be given, so it is up to you to think and code correctly!

Step 1 : extend the data model

Add a class *Planet*

C#

```
using Newtonsoft.Json;
using System.Collections.Generic;

namespace StarwarsUniverse.Domain
{

    public class Planet : Resource
    {
        public string Name { get; set; }

        // Note: RotationPeriod, OrbitalPeriod and Diameter
        // return sometimes "unknown"
        // quick-fix -> use string types (instead of int)
        // longer fix (Todo) -> change JSON deserialisation to substitute
"unknown" with -1

        [JsonProperty("rotation_period")]
        public string RotationPeriod { get; set; }

        [JsonProperty("orbital_period")]
        public string OrbitalPeriod { get; set; }

        public string Diameter { get; set; }

        public string Climate { get; set; }
```

```
        public string Gravity { get; set; }

        public string Terrain { get; set; }

        [JsonProperty("surface_water")]
        public string SurfaceWater { get; set; }

        public string Population { get; set; }
    }
}
```

Step 2 : Change StarWarsContext

Add a *Planets* property of type DbSet<Planet>

Step 3 : Write up PlanetApiRepository with Unit Test

This class should obey the following interface:

C#

```
using StarWarsUniverse.Domain;
using System.Collections.Generic;

namespace StarWarsUniverse.Data.Repositories
{
    public interface IPlanetRepository
    {
        IList<Planet> GetAllPlanets();
    }
}
```

In the unit test (called PlanetApiRepositoryTests), you verify there are at least 60 planets coming from swapi (Attention: swapi returns the planets in multiple resultpages).

Step 4 : Write up seed data

Modify seed data code to fetch all planets in order to store them in the Planets table.

Step 5 : Migrate

Create a new migration called cloneplanets and update your database. There should be a new Planets table containing all planets fetched from swapi.

Exercise 4 : Create a many to many relationship

In this exercise we will create a many to many relationship between Movies and Planets. Therefore we will introduce a new class called MoviePlanet which act as a join entity. We will also introduce a new migration to add this model change and to populate the database accordingly.

Step 1 : Create MoviePlanet and change Movie and Planet

Create a new class with the following properties:

C#

```
namespace StarWarsUniverse.Domain
{
    public class MoviePlanet
    {
        public string MovieUri { get; set; }
        public Movie Movie { get; set; }
        public string PlanetUri { get; set; }
        public Planet Planet { get; set; }
    }
}
```

Furthermore we will add properties to the classes Movie and Planet. First we change Movie:

C# : Movie

```
[JsonIgnore]
public List<MoviePlanet> MoviePlanets { get; set;} = new List<MoviePlanet>();

[JsonProperty(PropertyName = "planets")]
public List<string> PlanetUris { get; set; }
```

Property MoviePlanets acts as a navigation property so you can retrieve all planets for a given movie. These planets are fetched from swapi and stored in PlanetUris.

Notes

1. MoviePlanets is not available in JSON, therefore we can't map it. Hence attribute [JsonIgnore]. PlanetUris won't be mapped to the database table Movies. We will configure this later in StarWarsContext with Fluent API.

2. Actually if you look at class Movie, it serves two purposes: first it act as (de)serialization object from a JSON string and secondly its instances are entities from the EF Core framework. In a more elaborate architecture you would introduces separate classes for this (e.g. EFMovie and JSONMovie) and/or use special mapping frameworks like AutoMapper.

Now we change Planet is similar manner:

C# : Planet

```
[JsonIgnore]
public List<MoviePlanet> MoviePlanets { get; set;} = new List<MoviePlanet>();

[JsonProperty(PropertyName = "films")]
public List<string> MovieUris { get; set; }
```

Step 2 : Configure relationship

Modify OnModelCreating in StarWarsContext with the following lines:

C#

```
modelBuilder.Entity<Movie>().Ignore(movie => movie.PlanetUris);
modelBuilder.Entity<Planet>().Ignore(planet => planet.MovieUris);
```

This makes sure list of Uris won't be mapped in the database tables (it makes no sense because it duplicates the data in MoviePlanet).

Also add the following lines to define primary keys in MoviePlanet and thus create the many to many relationship:

C#

```
modelBuilder.Entity<MoviePlanet>().HasKey(moviePlanet => new
{
    moviePlanet.MovieUri, moviePlanet.PlanetUri
});
```

Finally you will have to add code to seed the MoviePlanets table. For every movie, look into every planet and if a planet appears in the movie, add a new MoviePlanet object. You only have to fill out MovieUri and PlanetUri:

```
// this code needs to be placed in a nested loop
modelBuilder.Entity<MoviePlanet>().HasData(new MoviePlanet()
                        {
                            MovieUri = movie.Uri,
                            PlanetUri = planet.Uri
                        });
```

Step 3 : Add Migration and Update Database

Time to add another migration. Call this one movieplanets and update your database. If everything went well there should be a MoviePlanets with data.

Step 4 : Elaborate unit test

Now you could elaborate on MovieDbRepositoryTests to check if the GetAllMovies method also loads the planets of each movie. Put one or more planets and movieplanets in the in-memory database and write a test that checks if the relations are loaded.

Step 5 : Modify Program.cs

Now modify the program that with every movie, also the planets are listed. So you will have to tweak MovieDbRepository (to make the previous test green) for that:

```
C#
public IList<Movie> GetAllMovies()
{
    return _context.Movies
                .Include(m => m.MoviePlanets)
                .ThenInclude(mp => mp.Planet)
                .OrderBy(m => m.EpisodeId).ToList();
}
```

Include specifies to load related entities in the result query, accordingly will ThenInclude add further related entities in the relationship.

Example output:

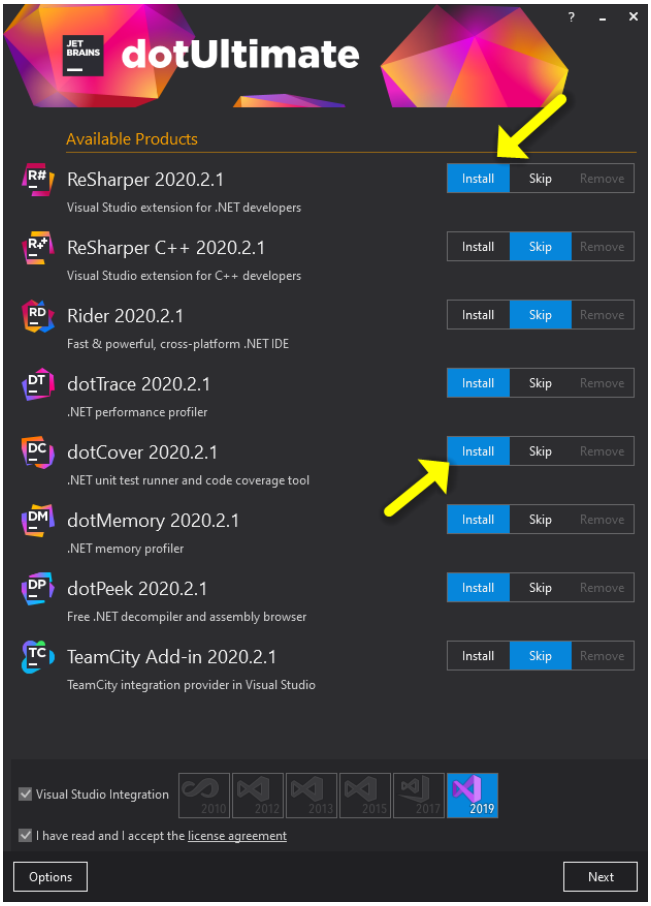
```
=== Star Wars Movies ===
Episode 1 - The Phantom Menace
   Released: 19/05/1999
   [( Tatooine ) ( Naboo ) ( Coruscant ) ]
Episode 2 - Attack of the Clones
   Released: 16/05/2002
   [( Tatooine ) ( Kamino ) ( Geonosis ) ( Naboo ) ( Coruscant ) ]
Episode 3 - Revenge of the Sith
   Released: 19/05/2005
   [( Tatooine ) ( Utapau ) ( Mustafar ) ( Kashyyyk ) ( Polis Massa ) (
Mygeeto ) ( Felucia ) ( Cato Neimoidia ) ( Saleucami ) ( Alderaan ) ( Dagobah
) ( Naboo ) ( Coruscant ) ]
Episode 4 - A New Hope
   Released: 25/05/1977
   [( Tatooine ) ( Alderaan ) ( Yavin IV ) ]
Episode 5 - The Empire Strikes Back
   Released: 17/05/1980
   [( Ord Mantell ) ( Hoth ) ( Dagobah ) ( Bespin ) ]
Episode 6 - Return of the Jedi
   Released: 25/05/1983
   [( Tatooine ) ( Dagobah ) ( Endor ) ( Naboo ) ( Coruscant ) ]
========================
```

Exercise 6: Unit Test Code Coverage

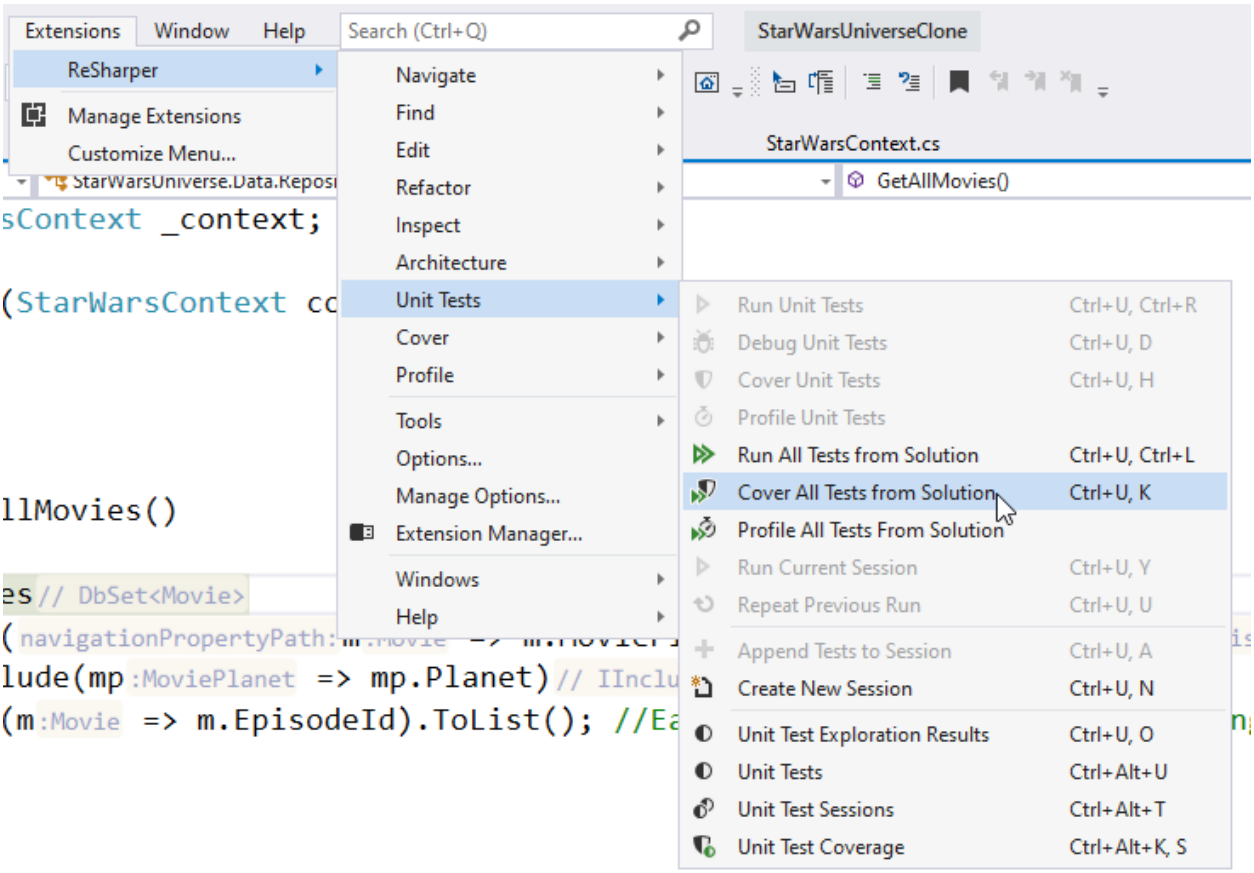
An important measure while developing your program is “Code coverage” (https://en.wikipedia.org/wiki/Code_coverage). It is expressed as a *percentage*, indicating *how much of your code is executed by running the unit tests.*

Visual Studio doesn't offer any means to calculate this number. Luckily there are some plugins who do. The easiest is to apply for a student license via (use your PXL mail address!):
https://www.jetbrains.com/community/education/#students

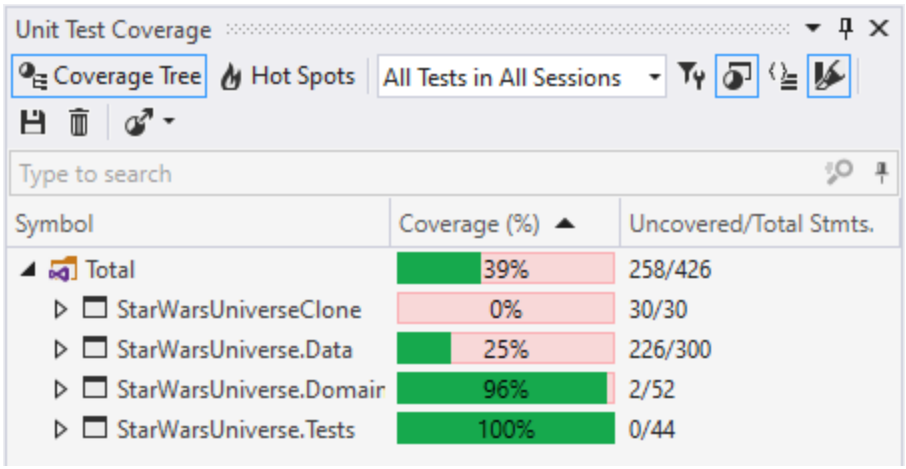
Next install at least Resharper and dotCover:



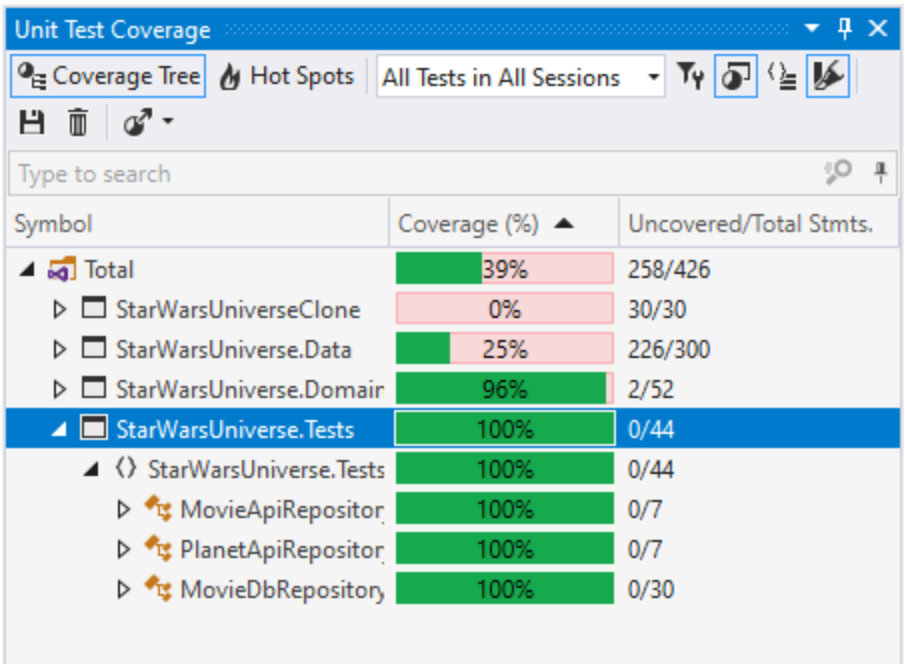
After the installation process, you calculate the code coverage by running your unit test like this:



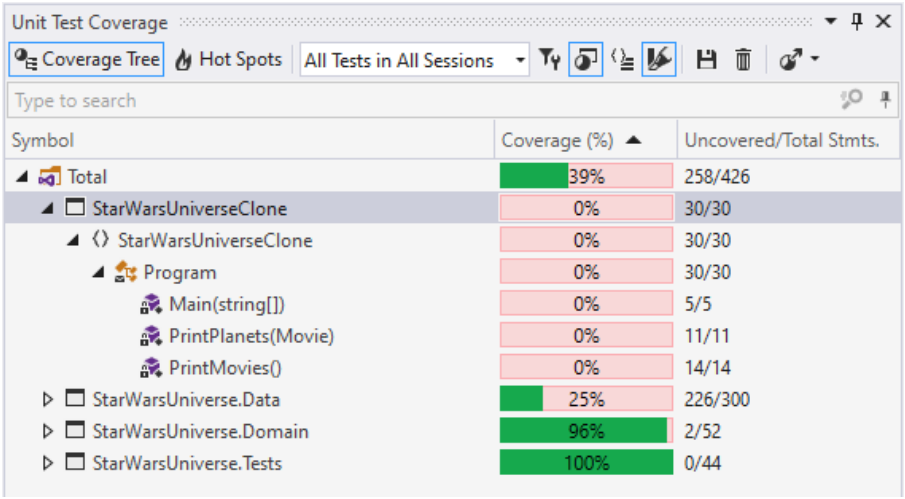
When the unit test runner has finished, you will get a report similar like this:



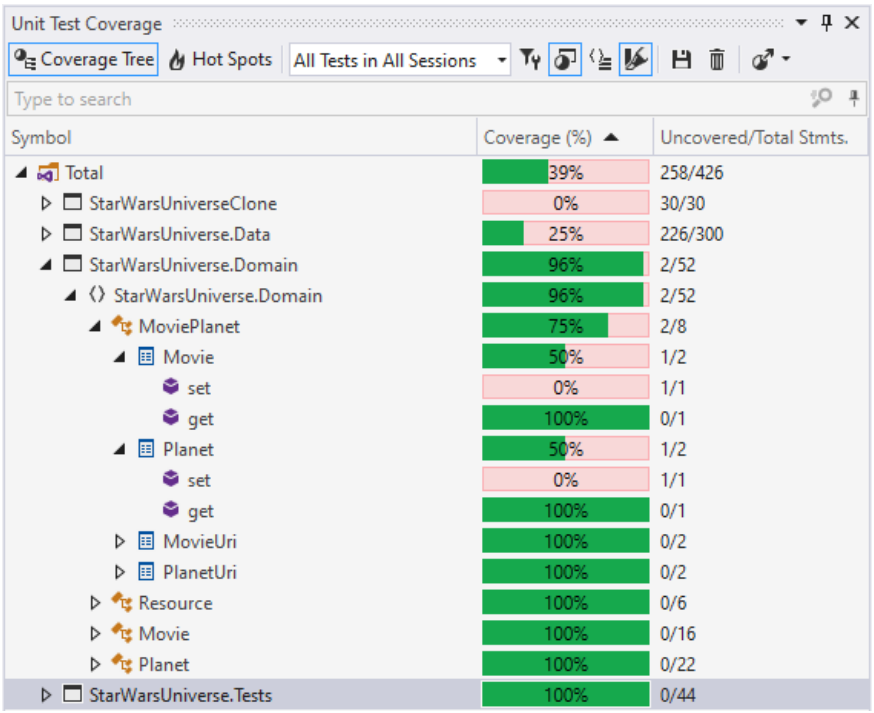
In our codebase we got a total coverage of 39% (which seem rather low), but we can drill down and try to explain the numbers further.



StarWarsUniverse.Tests have a coverage of 100%, which is obvious, because we actually run all the unit tests for calculating total coverage. So each and every statement of the unit test have run. This does *not* mean there could be more tests (because coverage for the other projects are lower).



Project StarWarseUniverseClone contains the main Program, which is always quite difficult to unit test. However, methods PrintPlanets and PrintMovies are not executed while running the tests, so there is still room for improvement.



StarWarsUniverse.Domain is covered good (96%), with only two setter properties who are not covered by the tests.

To do:

- Try to increase the coverage of your tests so that StarWarsUniverse.Domain is covered 100%.
- Which methods from class ResultsPage<T> are not covered? Can you fix this?

Exercise 7 (optional): Add all Star Wars entities to the database

Now add all the remaining entities to the database and add a migration per entity:

- People
- StarShips
- Vehicles
- Species