|  |  |
| --- | --- |
| **Exercise 1** | *EF Database-First* |
| **Project** | **EFCoreMovie1\_RazorPages** |
| **Purpose** | Use the Entity Framework Core to establish a connection to a local rela­tional database, and to retrieve and alter data in the database. |
| **Description** | In this exercise, we create a local relational database, tables and insert data in the database, by using a given database script. Then, we try to connect to the database using the Entity Framework Core **Database-first approach**, and access the data in the tables. |
| **Steps** | The project contains a text file called **MovieDBScript.txt**. This script can be used to generate tables in a database, and insert some sample data into the tables.   1. In your local database server, create a new local database called **MovieDB**. 2. Once the database is created , run the given **MovieDBScript.txt** script file on the database (Right-click on the database in the **SQL Server Object Explorer** window, choose **New Query**, copy the content of the script file into the query window, and execute the query). The data­base should now contain two tables **Movie** and **Studio**, and both tables should contain some sample data (seven records in **Movie**, three records in **Studio**). 3. In the **EFCoreMovie1\_Razor Pages** project, use the Entity Framework Core to create the classes and the context class corresponding to the **MovieDB** database by using either the **DbContext Scaffolding** or the ***Entity framework Power tool*** as described in the tutorial Part 2. This process should add three new classes named **MovieDBContext**, **Movie** and **Studio** to the project. Explore these classes. 4. For the moment, we are adopting the design below to implement the user stories:  * Be able to view all **Studios** * Be able to view all **Movies**   **Design** :    We have implemented the first user story. ***Please explore carefully the code*** of the **GetStudios.cshtml** and **GetStudiosModel.cshtml.cs**. We have commented Some code (to avoid compile errors because a model is missing). Please **Uncomment this code.** In case of compile error, try to fix them.   1. Configure the MovieDBContext class in the Startup.cs 2. Run the application, you will be able to display all the studios. 3. In the same way, implement the second user story **“**to be able to display all the movies”. |

|  |  |
| --- | --- |
| **Exercise 2** | *EF Database-First - Continued* |
| **Project** | **EFCoreMovie2\_RazorPages** |
| **Purpose** | Use the Entity Framework Core to establish a connection to a local rela­tional database, and to retrieve and alter data in the database. |
| **Description** | This exercise is a continuation of the previous one. In the previous exercise, we used the EFCore **Database-First** approach to connect to an existing database. The problem with the previous design is that the Razor Pages ( Front-end) was tightly coupled to the back-end., which does not enhance maintainability and extendibility. In this exercise , we will adopt a design where the Front-end is loosely coupled to the back-end by introducing an abstraction layer in between using interfaces as shown below. |
| **Steps** | We want to change our design so that it corresponds to the one shown above. For that, the IMovieService and the IStudioService should look as follows:   |  |  | | --- | --- | | public interface ***IMovieService***  {  public IEnumerable<Movie> **GetMovies()**  } | public interface ***IStudioService***  {  public IEnumerable<Studio> ***GetStudios();***  } |     The **EFMovieService** should implement the ***IMovieService*** interface.  The ***EFStudioService*** should implement the ***IStudioService*** interface .   1. Create a folder called **Interfaces** and create the 2 interfaces mentioned above 2. Create a folder called **EFServices** and add the 2 classes EFMovieService and EFStudioService that implement the 2 interfaces defined in question 1) . ***Implement*** these 2 classes. 3. Configure/Register your services in the ***Startup.cs*** class 4. Run your application, make sure that you still can display the list of Studios and the list of Movies as well. 5. We want to implement the following ***user story***:   “I will be able to fiIter **Studios** based on the name”.   * Add the ***IEnumerable<Studio>GetStudios(string name)*** method to the ***IStudioService*** interface * Implement this method in the ***EFStudioService*** * *In the**GetStudios.cshtml.cs class* ***,*** *add code that implement the filtering*  1. We want also to implement the following ***user story***:   “I will be able to filter **Movies** based on the title”   * Add the ***IEnumerable<Movie>GetMovies(string title)*** method to the ***IMovieService*** interface * Implement this method in the ***EFMovieService.*** * *In the**GetMovies.cshtml.cs class****,*** *add code that implement the filtering* |

|  |  |
| --- | --- |
| **Exercise 3** | *EF Database-First \_continued* |
| **Project** | **EFCoreMovie3\_RazorPages** |
| **Purpose** | Use the Entity Framework Core to establish a connection to a local rela­tional database, and to retrieve and alter data in the database. |
| **Description** | In this exercise, we continue working with the solution of the exercise 2. In the exercise 2 , we used the ***EFCore database-First approach*** to access the data into the database. We implemented ***SPRINT1*** that contains the following user stories:   * Be able to view **all studios** /**all movies** * Be able **to filter Studios** based on the name * Be able to **filter Movies** based on the title.   In this exercise, we are going to implement ***SPRINT 2*** having the following user stories   1. Be able to create a studio 2. Be able to add movies to a specific studio 3. Be able to delete a studio, but when deleting a studio all its movies should automatically be deleted. |
| **Steps** | The project **EFCoreMovie3\_RazorPages** is a possible solution for the **EFCoreMovie2\_RazorPages.**  **Step1**   * In the Studios folder :   + Add a razor page , name it **Create. It is** the page for creating a new Studio.   + Add a razor page, name it **Delete. It is** the page for deleting a Studio. * In the Movies folder , add a razor page , name it **Create. It is** the page for creating a new Movie.   **Step 2:**   * In the **IStudioService** interface :   + Add the ***void AddStudio(Studio studio)*** method**.**   + Add the ***Studio GetStudioById(int id)*** method**.**   + Add the ***void DeleteStudio(Studio studio)*** method**.** * In the **IMovieService** interface :   + Add the  ***void AddMovie(Movie movie)*** method,**.**   **Step 3**   * In the **EFStudioService** class:   + Implement the **void AddStudio(Studio studio*)*** method to add a new studio   + Implement the **void DeleteStudio(Studio studio)** method to delete a studio   + Implement the **Studio GetStudioById(int id)**method to geta studio based on its id. * In the **EFMovieService** class:   + Implement the **void** **AddMovie(Movie movie)** method to add a new movie.   **HINT:**  For the user story 2, you should select a Studio. Once selected, its **Id** is passed to the **Create** Movie page. Then we assign the id to the new Movie. The new Movie is then added to the database. ***VERY SIMPLE***.  **Step 4**   * In the **Studios Razor Pages**:   + Implement the code for adding a new Studio   + Implement the code for deleting a Studio   **Rememeber:** To allow the database delete all movies(children) when deleting a studio(parent) , add the **ON DELETE CASCADE** rule to the Foreign key constraint in the Movie table definition.   * In the ***Movies Razor Pages*** :   + Implement the code for adding a new Movie   **Step 5**   * Run the application * Create some studios and add some movies to studios * Delete a studio and check that its movies are deleted from the database as well   **Step 6**   * Implement the following user stories: * ***I will be able to display the movies that take place in a specific studio***. * ***I will be able to delete a movie***. * Test these user stories. |

|  |  |
| --- | --- |
| **Exercise 4** | EFCore Code-First |
| **Project** | **EFCoreTeaching\_RazorPages** |
| **Purpose** | Use the Entity Framework Core Code-First approach to establish a connection to a local rela­tional database, and to retrieve data in the database. |
| **Description** | In the tutorial part2, we have seen how to configure the one-to-many relationship in EF Core when using the Code First approach.  In this exercise, we are going to look at how to configure the **many-to-many** relationship. As you will see, it is necessary to include an entity in the model to represent the join table, and then add navigation properties to either side of the many-to-many relations that point to the join entity.  We are going to work on a simple **Student Registration System**. We will implement the following model: |
| **Steps** | The project contains a folder called **Models**. This folder contains 3 model classes and a context class as well. **Please** **Explore** deeply the classes in this folder and look how we configure the many-to many relationship and how we added validations (using data annotations) that will be applied to the database.   1. In the **EFCoreTeaching\_RazorPages** project, use the EFCore Migration to create the **RegistrationDB** database, as described in the tutorial Part 2. This process should create the database schema. Take a look at the Enrollment table definition in the T-SQL window, what is the meaning of the rule ***ON DELETE CASCADE*** *that is setup by default?* 2. This time, you ***are going to manually insert some data*** in the tables ( Right click on the table and select View data , then insert data into the table). Remember that the id is auto-incremented so no need to enter the id.  * Insert 15 students * Insert 4 courses. * Enroll each student in at least 2 courses   **NB**: when inserting enrollments, do not forget to comply to the reference integrity rule. By the way, what is a reference integrity?   1. Explore carefully the design and the code (implementation). As you can see, we have already implemented the 2 following user stories (***SPRINT 1***) :   User Story 1: I will be able to create a new Student.  User **story 2**: “*I will be able to view all students*”.  User **story 3**: “*I will be able to view all enrollments for a specific student*”  **NB:** The most important code is the use of the LINQ **include** method in the **EFEnrollementService** class to include the Enrollments navigation property of the student entity.   1. Configure the different service in the Startup.cs (in case they are not configured yet). Run the application and test the implemented user stories. 2. We want to expand our application by implementing the following user stories (***SPRINT 2***):  * ***User story 4****: I will be able to view all courses* * ***User story 5****: I will be able to display all enrollments (the name of the student and the student´s grade) in a* ***specific*** *course*     *In the ICourseService, define the following 2 methods*   * **IEnumerable<Course> GetCourses()**; ( for user story 4) * **Course GetCourse(int cid);** ( for user story 5)     *In the EFCourseService ,* ***implement*** *these 2 methods.*  **Hint:***For the implementation of the user story 5 , implement the* ***GetCourse(int cid)***  *by using the LINQ* ***include*** *method to include the enrollments navigation property of the course entity* ***Then include*** *the student having this enrollment. Take inspiration from the implementation of the user story 2.*   1. We want to expand more our application by implementing the following user story (***SPRINT 3***):  * ***User story 6****: I will be able to enroll a student into a course.*   **HINT**: You select a student. Once selected its **Id** is passed to the GetCourses() page( the page that display all courses). Then you select a course from the list, its id along with the student id are both passed to the **Create** Enrollment page. Once the student id and the course id are assigned to the new enrollment, the enrollment is created with eventually a grade.  ***Good Luck mohammed*** |

|  |  |
| --- | --- |
| Ex**ercise 5** |  |
| **Project** | **EFCoreHotel** |
| **Purpose** | Use the Entity Framework Core to establish a connection to a local rela­tional database, and to retrieve data in the database. |
| **Description** | In this exercise, we create a local relational database, and create tables and data in the database, by using a given database script. We then try to connect to the database through the Entity Framework Core.  We did use EFCore database-First approach to get the model classes and the context class, the same way we did previously. A script to create the database is provided with the project. |
| **Steps** | The project contains a text file called **HotelDBScript.txt**. This script can be used to generate tables in a database, and insert some sample data into the tables.   1. Create a new local database called **HotelDB**. 2. Run the given script file on the database (Right-click on the database in the **SQL Server Object Explorer** window, choose **New Query**, copy the content of the script file into the query window, and execute the query). The data­base should now contain four tables **Booking, Guest, Hotel** and **Room**, and all tables should contain some sample data. 3. In the **EFCoreHotel**  **RazorPages** project, we have already implemented **SPRINT1** , which contains the following user stories :    * I will be able to display all hotels    * I will be able to display all rooms    * I will be able to display all bookings    * I will be able to display all Guests   Take a closer look at the model classes. Some of them contain proper­ties of class types and collection types, like e.g. the **Rooms** property in the **Hotel** class, of type **ICollection<Room>**. What do you suppose such a pro­perty represents?  Run the application and Test all the user stories in SPRINT 1   1. We want to expand our application by implementing **SPRINT 2,** having the following user stories:  * I will be able to List price and type in all rooms of every hotel * I will be able to list booking for every guest.  1. We want to expand our application by implementing **SPRINT 3**  * For each room in a specific hotel , I want to display all bookings and the guest making this booking. |

|  |  |
| --- | --- |
| ***Exercise 6*** |  |
| **Project** | **ADONetCoreSale** |
| **Purpose** | The purpose is to establish a connection to a local rela­tional database using ADO.Net and perform CRUD operations on the data in the database that is based on the following model: |
| **Description** | The ***ADOSelling\_RazorPages*** project contains all the Front-end razor pages for Creating , deleting, updating , getting a specific item and getting all items ( item could be either a seller or a customer entity.  In this exercise, we will implement a ADO.Net data access layer to access the ***SellingDB*** database. You will write queries to perform the following user stories:   * Create a Seller * Create a Customer for a specific Seller * Delete a Seller ( all Customer should automatically be removed) * View all Sellers and View All Customers * View every seller and her/his customers * View all customers who do not work through a seller ( customers working by their own) |
| **Steps** | 1. In the local SQL Server, create a database , name it ***SellingDB*** 2. Create the **Seller** and the **Customer** tables 3. The tables should contain the following data :     The design of the application is illustrated as shown below:      Now that you get the hang of how to implement the Front-End ( Razor pages part ), the interface services ( ***ISellerService*** and ***ICustomerService***) and the implementation services (***ADOSellerService*** and ***ADOSellerService***), we have implemented everything for you. The only part missing is the ADO.Net level shown on the design above.   1. Implement ***SPRINT 1*** having the following user stories :  * I will be able to get all Sellers * I will be able to get all Customers * I will be able to create a new Seller * I will be able to create a new Customer ***that may work*** through a specific seller.  1. Implement ***SPRINT 2*** having the following user stories :  * I will be able to update a seller * I will be able to update a customer * I will be able to delete a seller, its eventual associated customers should automatically be removed.  1. Implement ***SPRINT 3*** having the following user stories :  * I will be able to display a list of Seller name , customer name and their cities for the seller and customer who belongs to the same city. * I will be able to view customers working on their own ( without any seller) * I will be able to display a list in ascending order for the salesmen who works either for one or more customer or not yet join under any of the customers.   **Good Luck** |

|  |  |
| --- | --- |
| ***Exercise 7*** |  |
| **Project** | **ADONetCoreMovie** |
| **Purpose** | The purpose is to establish a connection to a local rela­tional database using ADO.Net and perform CRUD operations on the data in the database that is based on the following model:  ??? |
| **Description** | The ***ADONetMovie\_RazorPages*** project contains all the Front-end razor pages for Creating , deleting, updating , getting a specific item and getting all items ( item could be either a seller or a customer entity.  In this exercise, we will implement a ADO.Net data access layer to access the **Cinema*DB*** database. You will write queries to perform the following user stories:   * **View every actor and her/his movies** * **View every studio and its movies** * Create a Movie for a specific actor and studio * View all movies whose studio and actor are in the same city ? * View all movies along with the name of its actor and the name of the studio the movie took place? |
| **Steps** | 1. In the local SQL Server, create a database , name it ***CinemaDB*** 2. Run the script CinemaDBCreate.txt to create the tables : Movie , Actor and Studio. 3. Run the CinemaDBInsert.txt to insert data in different tables   The design of the application is illustrated as shown below:  **Design**    Now that you get the hang of how to implement the Front-End ( Razor pages part ), the interface services ( ***ISellerService*** and ***ICustomerService***) and the implementation services (***ADOSellerService*** and ***ADOSellerService***), we have implemented everything for you. The only part missing is the ADO.Net level shown on the design above.   1. Implement ***SPRINT 1*** having the following user stories :  * I will be able to get all Sellers * I will be able to get all Customers * I will be able to create a new Seller * I will be able to create a new Customer ***that may work*** through a specific seller.  1. Implement ***SPRINT 2*** having the following user stories :  * I will be able to update a seller * I will be able to update a customer * I will be able to delete a seller, its eventual associated customers should automatically be removed.  1. Implement ***SPRINT 3*** having the following user stories :  * I will be able to display a list of Seller name , customer name and their cities for the seller and customer who belongs to the same city. * I will be able to view customers working on their own ( without any seller) * I will be able to display a list in ascending order for the salesmen who works either for one or more customer or not yet join under any of the customers.   **Good Luck** |

|  |  |
| --- | --- |
| **Exercise 8** |  |
| **Project** | EFCoreMovie and EFCoreHotel |
| **Purpose** | Use the Entity Framework Core to establish a connection to a data­base deployed to the Azure cloud service. |
| **Description** | In this exercise, we create a relational database **on Azure**, and create tables and data in the database, by using a given database script. We then try to connect to the database through the Entity Framework Core, and access the data in the tables. |
| **Steps** | Perform the same steps as in **Exercise 3** and **Exercise 5**, with a couple of changes:   1. The database should be created on **Azure** 2. You may need to create tables from both exercises in the same database, since you might not be allowed to create more than a single database on **Azure**. However, the same database on Azure can contain tables from many “logical” databases without problems. |