# Web Services and Cloud (March 2015) – Lab – MoviesGallery

The goal of this lab is to learn how to develop web services with the ASP.NET Web API framework, Entity Framework and SQL Server. You will create database, map the database to EF code first data model and web services project which works with the database.

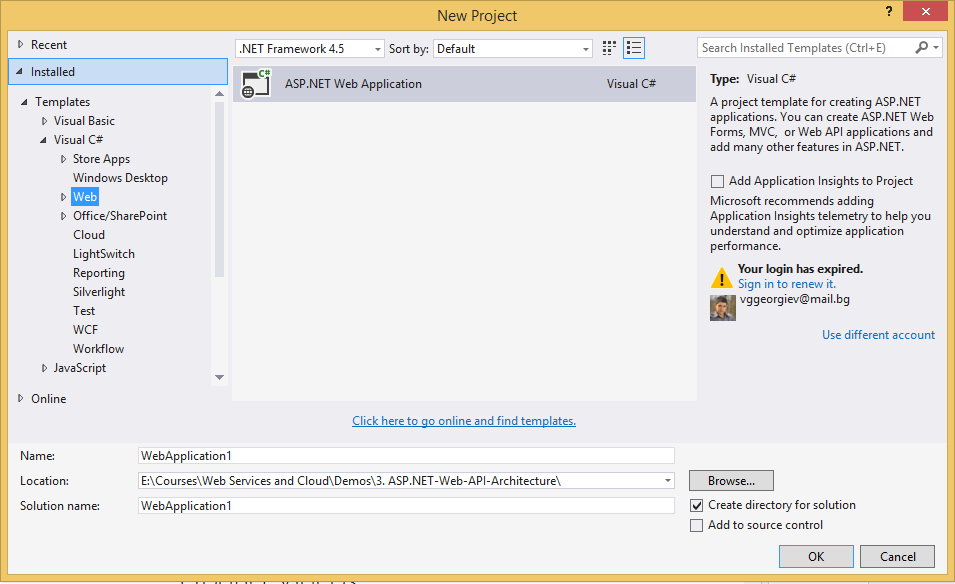
## Task 1. Create the Web API project

### Open Visual Studio and create empty solution

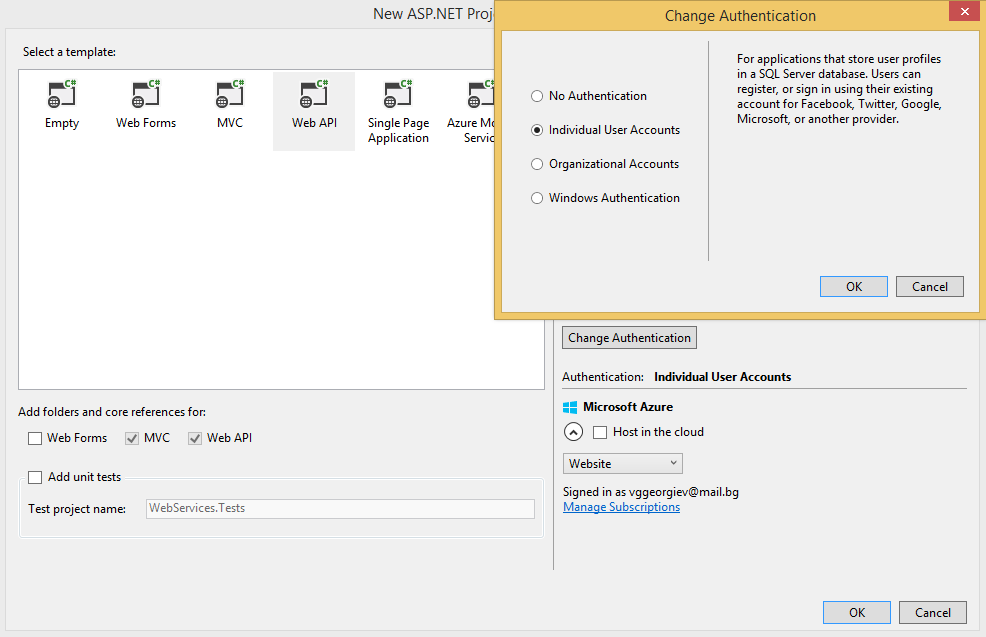
Create empty solution with the name of the project (*Example: MoviesGallery*).

### Add new “ASP.NET Web Application” project in the solution

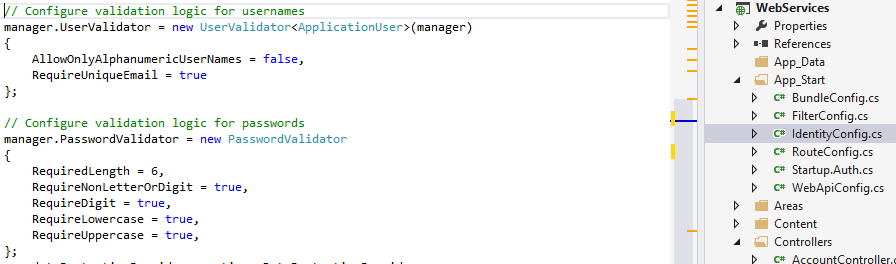
Set the name of the project with the solution as prefix (*Examples: MoviesGallery.WebServices, MoviesGallery.Services*)



### Create “Web API” project with Individual User Accounts



### Configure user validator and password validator.

Go to the IdentityConfig.cs in AppStart folder and set the user and password validators of your choice. You can allow only alphanumeric usernames. You can set validations for password like: required minimum length, required uppercase letter, required lowercase letter, required digit and required symbol. ****

### Change connection string

Your task is to change the connection string to work with SQL Server instance on your machine. Make the database name with suitable name.

### Start the project and try to register and login user using postman

* **Registration –** You should send **Email**, **Password** and **ConfirmPassword** to the Register action.
* **Get Token –** You should send **Username**, **Password** and **grant\_type** to the /Token url.

Test the registration and login. Ensure that the user is in the right database.

## Task 2. Create the EF Code First Models

### Add new project for the models

Add new project for the code first models (*Name example: MoviesGallery.Models*).

You should add the packages: **EntityFramework** and **Microsoft ASP.NET Identity EntityFramework**

### Move the ApplicationUser class

Your task is to move the ApplicationUser class from the IndentityModels.cs to the created project.

### Add the reference to the Models project

Your task is to add reference to the models project in the Web API project.

Build the solution and add the missing usings.

### Rename the ApplicationUser to User.

Your task is to rename the ApplicationUser class to User. Ensure that the build succeeded.

### Add code first models

* User – Extend user class. Every user should have personal page, gender, birth date. Every user should have collection of favourite movies, collection of favourite actors, collection of reviews
* Actor – Actors should have id, name, born date, biography, home town and collection of movies.
* Genre – Genres should have id, name and collection of movies.
* Movie – Movies should have id, title, length (in minutes), ration (from 1 to 10), country, collection of actors and collection of reviews.
* Review – Reviews should have id, User, Movie, content and date of creation and

Add navigational properties in all models.

## Task 3. Create the Data layer

### Add new project for the data layer.

Your task is to add new project for the data layer (*Name example: MoviesGallery.Data*).

You should add the reference to the Models project.

You should add the packages: EntityFramework and Microsoft ASP.NET Identity EntityFramework**.**

### Move the ApplicationDbContext

Your task is to move the ApplicationDbContext from the IndentityModels.cs to the data project.

### Add the reference to the Data project

Your task is to add reference to the Data project in the Web API project.

Build the solution and add the missing references in the Web API project.

### Add the models in the context

Add created code first models in the DbContext class as properties of type IDbSet<T>.

### Add migrations in the Data project

You should add migrations. You have two options

* You can use the Enable-Migrations command in the console
* You can create the Configurations class that extends DbMigrationsConfiguration<[your DbContext]>

In the constructor you should set properties AutomaticMigrationDataLossAllowed to false and AutomaticMigrationsEnabled to true

### Set migration strategy in the context constructor

Your task is to set migration strategy for your choice in your context’s constructor. We suggest to use MigrateDatabaseToLatestVersion.

### Add interface IRepository

Make generic IRepository. The interface should have methods for:

* Getting all data and returns IQueryable collection.
* Get entity by id. The parameter is of type object, because the id can be int, string, Guid… and returns the entity.
* Method for adding entity.
* Method for updating entity by passed entity as parameter.
* Method for deleting entity by passed entity as parameter.
* Method for deleting entity by passed id as parameter.
* Method for detaching entity by passed entity as parameter.
* SaveChanges method that returns int.

### Add GenericRepository implementation

The GenericRepository should implements the IRepository interface. It should be generic also.

The GenericRepository should have DbContext and IDbSet<T> properties.

The constructor should have DbContext parameter. Set the context property to the passed parameter and set property to the context.Set<T>().

The Set<T> represents the collection of entities from the database table. You can work over the set.

* Implement ChangeState private method – The method have two parameters: T entity & EntityState state. The method should get the entry from the context by passed entity. If the state of entry is detached you should attach the entity to the Set.
* Implement SaveChanges method that returns SaveChanges of the context.
* Add, Update and Delete methods should only change the state of the passed entities to Added, Modified or Deleted. Changing state of the entity to added, adds entity to the database. Changing state of the entity to modified, updates entity to the database. Changing state of the entity to deleted, deletes entity from the database.
* All method should returns the Set.
* GetById – Call the Find method of the Set and return the result.

### Add Unit of Work interface

Make the interface IData. The interface should have properties for all code first models.

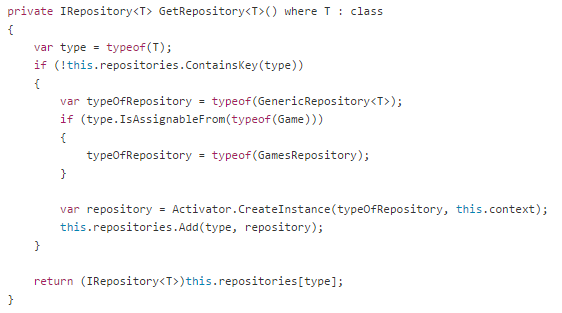
* IRepository<User> Users { get; }
* IRepository<Movie> Movies { get; }
* IRepository<Genre> Genres { get; }
* IRepository<Actor> Actors { get; }
* IRepository<Review> Reviews { get; }
* int SaveChanges();

### Add Unit of Work implementation

Add implementation for the interface. It should have two properties.

* DbContext context
* IDictionary<Type, object> repositories

Pass the context as parameter. Initialize the dictionary in the constructor. Use the GetRepository method and implement the interface properties. If you want try to implement it.



### Add the test data in the database

Create several users with postman and then use the seed method to put test data in all tables in the database. Make relationships between test data and the created users with the postman in the database.

## Task 4. Write the controllers

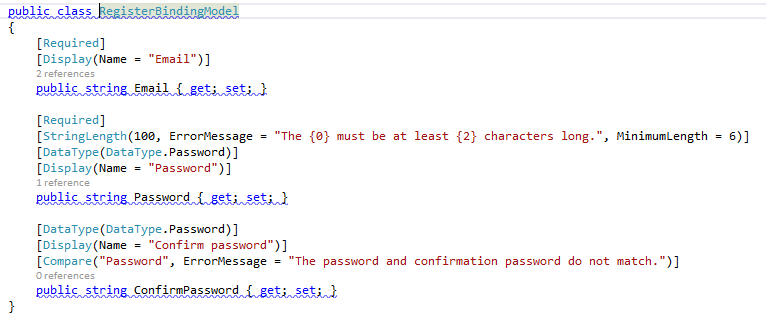
### BaseApiController

Implement the ApiController. Add private field IMoviesGalleryData data. Make protected property for the data only with getter. Make constructor with parameter IMoviesGalleryData data and default constructor that calls the constructor bellow.

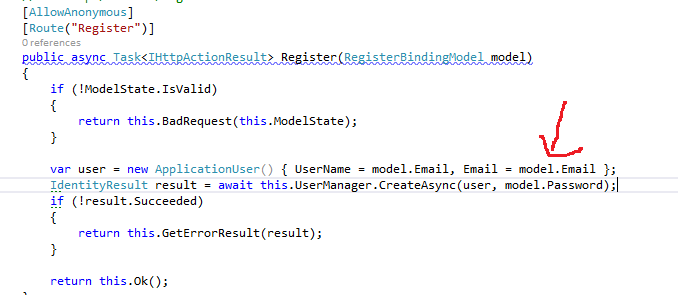
All controllers should inherit the BaseApiController.

### AccountController

Extend the RegisterBindingModel in the AccountBindingModels file. You should add **personal page**, **gender**, **birth date**. Add the needed validations. The personal page is optional, gender and birth date are required.



You should add the new properties from the model in the register method:



### UsersController

UsersController should have the following actions:

* GetAllUsers – Returns all users. Return only **Id**, **Username**, **BirthData**, **PersonalPage** and Gender.
* GetUserById – Returns detailed information about the user with his favorite movies and actors, and all reviews.
* GetUsersByGender – Returns all users by gender. Return the data model from GetAllUsers action.

Make the following DataModels:

* **DetailedUserDataModel**
* **ShortUserDataModel**
* **ShortMovieDataModel**
* **ShortActorDataModel**
* **ShortReviewDataModel**

### ValuesController

You should remove the unused ValuesController.

### MoviesController

MoviesController should have the following actions:

* GetAllMovies(); -
* GetMovieById(int id); -
* GetMoviesByGenre(int genreId); -
* AddMovie(MovieBindingModel model); -
* UpdateMovie(int id, MovieBindingModel model); -
* DeleteMovie(int id); -

### GenresController

* GetAllGenres(); -
* AddGenre(GenreBindingModel model); -
* UpdateGenre(int id, GenreBindingModel model); -
* DeteleGenre(int id); -

### ActorsController

* GetActorsByMovie(int movieId); -
* GetActorById(int id); -
* AddActor(int movieId, ActorBindingModel model); -
* UpdateActor(int id, ActorBindingModel model); -
* DeleteActor(int id); -

### ReviewsController

* GetReviewsByMovie(int movieId); -
* AddReview(int movieId, ReviewBindingModel model); -
* DeleteReview(int id); -

## Task 5. Test the application

### Test the data layer

Make unit tests for the MoviesRepository. Use the transactions over the original database.

1. Create Test project.
2. Add references to the Data project.
3. Create MoviesRepository and pass it the context.
4. Test all repository methods with different cases.

### Test the controllers

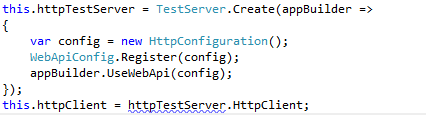
Test the ActorsController. Make FakeActorsRepository and inject it in to the controller. The fake repository should works with List<Actor>. Put some fake data in the list and test the ActorsController with this fake data.

### Make integration tests

Make integration test for the UsersController.

Use the TestServer from the Microsoft.Owin.Testing. Use HttpClient from System.Net.Http

Start OWIN testing HTTP server in TestInitialize method and the client to server’s http client:



Add fake data in the database. You can use test database. Just change the connection string database name in the test project.

Make the request with http client and read the response. Assert the input data with the data from the response.