```
/*Листинг класса Role*/
using System;
using Contracts;
using Microsoft.AspNet.Identity.EntityFramework;

namespace Entity.Domain.Identity
{
   public class Role : IdentityRole<int, UserRole>, ITimeStamp, IEntity
   {

      public Role()
      {
         CreatedUtc = DateTime.UtcNow;
      }

      public Role(string name)
      {
         Name = name;
      }

      public DateTime CreatedUtc { get; set; }
   }
}
/*Листинг класса User*/
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Security.Claims;
using System.Threading.Tasks;
using Contracts;
using Entity.Domain.Training;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;

namespace Entity.Domain.Identity
{
   // You can add profile data for the user by adding more properties to your User class, please
visit http://go.microsoft.com/fwlink/?LinkID=317594 to learn more.
   /// <summary>
   /// Класс пользователя
   /// </summary>
   public class User : IdentityUser<int, UserLogin, UserRole, UserClaim>, ITimeStamp,
IEntity
   {
```

```
        public User()
        {
            CreatedUtc = DateTime.UtcNow;
        }

        public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<User, int>
manager)
        {


  var userIdentity = await manager.CreateIdentityAsync(this,
DefaultAuthenticationTypes.ApplicationCookie);

            return userIdentity;
        }
        /// <summary>
        /// Время регистрации пользоваеля
        /// </summary>
        public DateTime CreatedUtc { get; set; }
        /// <summary>
        /// Настройки пользователя
        /// </summary>
        public Settings.Settings Settings { get; set; }
        /// <summary>
        /// Список тренировок пользователя
        /// </summary>
        public List<UserTraining> Trainings { get; set; }


    }
}

/*Листинг класса UserClaim */

using Microsoft.AspNet.Identity.EntityFramework;

namespace Entity.Domain.Identity
{
    public class UserClaim : IdentityUserClaim<int>
    {
    }
}

/*Листинг класса UserLogin */

using Microsoft.AspNet.Identity.EntityFramework;

namespace Entity.Domain.Identity
{
    public class UserLogin : IdentityUserLogin<int>
```

```
    {
    }
}


/*Листинг класса UserRole*/

using Microsoft.AspNet.Identity.EntityFramework;

namespace Entity.Domain.Identity
{
    public class UserRole : IdentityUserRole<int>
    {
    }
}

/* Листинг класса Settings*/
using Contracts;
using Entity.Domain.Identity;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Entity.Domain.Settings
{
    public class Settings : IEntity
    {
        public int Id { get; set; }

        public User User { get; set; }

        public Profession Profession { get; set; }

        public int? ProfessionId { get; set; }

        public List<TrainingTime> DefaultTrainingTimes { get; set; }

    }
}

/*Листинг класса TrainingTime*/
using Contracts;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Entity.Domain.Settings
```

```
{
    public class TrainingTime : IIdEntity
    {
        public int Id { get; set; }

        public TimeSpan Value { get; set; }

        public Settings Settings { get; set; }

        public int SettingsId { get; set; }
    }
}
```

/*Листинг класса UserExercise*/

```
using Contracts;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Entity.Domain.Training
{
    public class UserExercise : IEntity
    {
        public int Id { get; set; }

        public UserTraining UserTraining { get; set; }

        public int UserTrainingId { get; set; }

        public Exercise Exercise { get; set; }

        public int ExerciseId { get; set; }

        public int CountOfRepeats { get; set; }
    }
}
```

/* Листинг класса UserTraining*/

```
using Contracts;
using Entity.Domain.Identity;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace Entity.Domain.Training
{
   public class UserTraining: IEntity
   {
      public int Id { get; set; }

      public User User { get; set; }

      public int UserId { get; set; }

      public DateTime Created { get; set; }

      public bool IsPassed { get; set; }

      public List<UserExercise> Exercises { get; set; }
   }
}

/*Листинг класса Criteria*/
using Contracts;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Entity.Domain
{
   public class Criteria : IIdEntity
   {
      public int Id { get; set; }

      public string Name { get; set; }
   }
}


/*Листинг класса Exercise*/
using Contracts;
using Entity.Enums;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Entity.Domain
{
   public class Exercise : IIdEntity
   {
      public int Id { get; set; }
```

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

        public string Description { get; set; }

        public string VideoUrl { get; set; }

        public DifficultyLevel DifficultyLevel { get; set; }

        public List<ExerciseCriteria> ExerciseCriterias { get; set; }
    }
}

/*Листинг класса ExerciseCriteria*/
using Contracts;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Entity.Domain
{
    public class ExerciseCriteria : IIdEntity
    {
        public int Id { get; set; }
        public int Weight { get; set; }
        public Exercise Exercise { get; set; }
        public int ExerciseId { get; set; }
        public Criteria Criteria { get; set; }
        public int CriteriaId { get; set; }
    }
}

/*Листинг класса Profession*/
using Contracts;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Entity.Domain
{
    public class Profession : IIdEntity
    {
        public int Id { get; set; }

        public string Name { get; set; }

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

```
        public List<ProfessionCriteria> ProfessionCriterias { get; set; }
    }
}

/*Листинг класса ProfessionCriteria*/

using Contracts;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Entity.Domain
{
    public class ProfessionCriteria : IIdEntity
    {
        public int Id { get; set; }
        public int Weight { get; set; }
        public Profession Profession { get; set; }
        public int ProfessionId { get; set; }
        public Criteria Criteria { get; set; }
        public int CriteriaId { get; set; }
    }
}

/*Листинг перечисления DifficultyLevel*/

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Entity.Enums
{
    public enum DifficultyLevel
    {
      [Description("Легко")]
      Easy,
      [Description("Средне")]
      Medium,
      [Description("Тяжело")]
      Hard
    }
}

/*Листинг класса Roles*/
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Entity.Identity
{
   public class Roles
   {
      public const string User = "User";
      public const string Admin = "Admin";

      public static List<string> GetAllRoles()
      {
         return new List<string>
         {
            User,
            Admin
         };
      }
   }
}


/* Листинг интерфейса IGenericRepository */
using Contracts;
using Data.Extensions.Transformers;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;
using System.Text;
using System.Threading.Tasks;
namespace Data.Interfaces.Repositories
{
   public interface IGenericRepository<TEntity> where TEntity : class, IEntity
   {
      IList<TEntity> GetFilteredList<TOrderField>(
         Expression<Func<TEntity, bool>> filter,
         Expression<Func<TEntity, TOrderField>> order = null,
         bool? orderAsc = true,
         int? skip = null,
         int? take = null,
         params Expression<Func<TEntity, object>>[] includes);

      Task<IList<TEntity>> GetFilteredListAsync<TOrderField>(
         Expression<Func<TEntity, bool>> filter,
         Expression<Func<TEntity, TOrderField>> order = null,
         bool? orderAsc = true,
         int? skip = null,
         int? take = null,
```

```
            params Expression<Func<TEntity, object>>[] includes);

        TEntity GetSingle(Expression<Func<TEntity, bool>> filter, params
Expression<Func<TEntity, object>>[] includes);
        Task<TEntity> GetSingleAsync(Expression<Func<TEntity, bool>> filter, params
Expression<Func<TEntity, object>>[] includes);

        Task<TEntity> GetAsync(Expression<Func<TEntity, bool>> filter, params
Expression<Func<TEntity, object>>[] includes);

        Task<TEntity[]> GetAllAsync(
            Expression<Func<TEntity, bool>> predicate,
            Func<IQueryable<TEntity>, IOrderedQueryable<TEntity>> orderBy,
            params Expression<Func<TEntity, object>>[] includeProperties);

        Task<PaginatedList<TEntity>> GetPaginatedAsync(
            int pageIndex,
            int pageSize,
            Expression<Func<TEntity, bool>> predicate,
            Func<IQueryable<TEntity>, IOrderedQueryable<TEntity>> orderBy,
            params Expression<Func<TEntity, object>>[] includeProperties);

        Task<ScrollableList<TEntity>> GetScrollableAsync(
            int skip,
            int take,
            Expression<Func<TEntity, bool>> predicate,
            Func<IQueryable<TEntity>, IOrderedQueryable<TEntity>> orderBy,
            params Expression<Func<TEntity, object>>[] includeProperties);

        int Count(
            Expression<Func<TEntity, bool>> filter,
            params Expression<Func<TEntity, object>>[] includes);

        T Max<T>(Expression<Func<TEntity, bool>> filter, Expression<Func<TEntity, T>>
max,
            params Expression<Func<TEntity, object>>[] includes);

        void Insert(params TEntity[] items);

        void Delete(params TEntity[] items);

        void Update(params TEntity[] items);

        IQueryable<TEntity> Collection { get; }

        IQueryable<TEntity> CollectionWithTracking { get; }
    }
}

/* Листинг интерфейса IUnitOfWork*/
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Contracts;
using Data.Interfaces.Repositories;

namespace Data.Interfaces
{
   public interface IUnitOfWork
   {
      IGenericRepository<TEntity> Repository<TEntity>() where TEntity : class, IEntity;

      int SaveChanges();

      Task<int> SaveChangesAsync();

      Task<int> SaveChangesAsync(CancellationToken cancellationToken);
   }
}

/* Листинг класса DbContext */
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Data.Implementations.Configurations;
using Data.Implementations.Migrations;
using Entity.Domain.Identity;
using Microsoft.AspNet.Identity.EntityFramework;

namespace Data.Implementations.Context
{
   public class DataContext : IdentityDbContext<User, Role, int, UserLogin, UserRole,
UserClaim>
   {
      public DataContext() : base("DefaultConnection")
      {
         Database.SetInitializer(new MigrateDatabaseToLatestVersion<DataContext,
Configuration>());
         Configuration.LazyLoadingEnabled = false;
         Configuration.ProxyCreationEnabled = true;
      }

      protected override void OnModelCreating(DbModelBuilder modelBuilder)
      {
         base.OnModelCreating(modelBuilder);
```

```
            // The Identity entities configuration canot be removed into a separate class
configuration
            // as they would have overriden the pre-defined in-house configuration
            //
            modelBuilder.Entity<User>().ToTable("Users");

modelBuilder.Entity<User>().HasRequired(s=>s.Settings).WithRequiredPrincipal(s=>s.User).
WillCascadeOnDelete(true);
            modelBuilder.Entity<User>().HasMany(s => s.Trainings).WithRequired(s =>
s.User).WillCascadeOnDelete(true);
            modelBuilder.Entity<Role>().ToTable("Roles");
            modelBuilder.Entity<UserRole>().ToTable("UserRoles");
            modelBuilder.Entity<UserClaim>().ToTable("UserClaims");
            modelBuilder.Entity<UserLogin>().ToTable("UserLogins");
            modelBuilder.Configurations.Add(new CriteriaConfiguration());
            modelBuilder.Configurations.Add(new ExerciseConfiguration());
            modelBuilder.Configurations.Add(new ExerciseCriteriaConfiguration());
            modelBuilder.Configurations.Add(new ProfessionConfiguration());
            modelBuilder.Configurations.Add(new ProfessionCriteriaConfiguration());
            modelBuilder.Configurations.Add(new SettingsConfiguration());
            modelBuilder.Configurations.Add(new TrainingTimeConfiguration());
            modelBuilder.Configurations.Add(new UserTrainingConfiguration());
            modelBuilder.Configurations.Add(new UserExerciseConfiguration());

        }
    }
}

/* Листинг класса GenericRepository */
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.Migrations;
using System.Linq;
using System.Linq.Expressions;
using System.Text;
using System.Threading.Tasks;
using Contracts;
using Data.Extensions;
using Data.Extensions.Transformers;
using Data.Interfaces.Repositories;
using LinqKit;

namespace Data.Repository
{
    public class GenericRepository<TEntity> : IGenericRepository<TEntity>
        where TEntity : class, IEntity
    {
        protected readonly DbContext _context;
        protected readonly DbSet<TEntity> _dbSet;
```

```
        public GenericRepository(DbContext context)
        {
            _context = context;
            _dbSet = _context.Set<TEntity>();
        }

        public IList<TEntity> GetFilteredList<TOrderField>(
            Expression<Func<TEntity, bool>> filter,
            Expression<Func<TEntity, TOrderField>> order = null,
            bool? orderAsc = true,
            int? skip = null,
            int? take = null,
            params Expression<Func<TEntity, object>>[] includes)
        {
            var query = QueryGetFilteredList(filter, order, orderAsc, skip, take, includes);

            return query.AsNoTracking().ToList();
        }

        public async Task<IList<TEntity>> GetFilteredListAsync<TOrderField>(
            Expression<Func<TEntity, bool>> filter,
            Expression<Func<TEntity, TOrderField>> order = null,
            bool? orderAsc = true,
            int? skip = null,
            int? take = null,
            params Expression<Func<TEntity, object>>[] includes)
        {
            var query = QueryGetFilteredList(filter, order, orderAsc, skip, take, includes);

            return await query.AsNoTracking().ToListAsync();
        }

        private IQueryable<TEntity>
QueryGetFilteredList<TOrderField>(Expression<Func<TEntity, bool>> filter,
Expression<Func<TEntity, TOrderField>> order, bool? orderAsc, int? skip, int? take,
            Expression<Func<TEntity, object>>[] includes)
        {
            IQueryable<TEntity> dbQuery = _context.Set<TEntity>();
            foreach (var navigationProperty in includes)
            {
                dbQuery = dbQuery.Include(navigationProperty);
            }

            var query = dbQuery.Where(filter).AsQueryable();


            if (order != null)
            {
                if (orderAsc.HasValue && orderAsc.Value)
                {
                    query = query.OrderBy(order).AsQueryable();
```

```
            }
            else
            {
                query = query.OrderByDescending(order).AsQueryable();
            }
        }
        else
        {
            query = query.OrderBy(o => o.Id).AsQueryable();
        }

        if (skip.HasValue)
        {
            query = query.Skip(skip.Value);
        }

        if (take.HasValue)
        {
            query = query.Take(take.Value);
        }
        return query;
    }

    public TEntity GetSingle(Expression<Func<TEntity, bool>> filter, params
Expression<Func<TEntity, object>>[] includes)
    {
        return GetSingleInternal(filter, includes).FirstOrDefault();
    }

    public async Task<TEntity> GetSingleAsync(Expression<Func<TEntity, bool>> filter,
params Expression<Func<TEntity, object>>[] includes)
    {
        return await GetSingleInternal(filter, includes).FirstOrDefaultAsync();
    }

    public async Task<TEntity> GetAsync(Expression<Func<TEntity, bool>> filter, params
Expression<Func<TEntity, object>>[] includes)
    {
        return await GetSingleInternal(filter, includes).FirstOrDefaultAsync();
    }

    public int Count(Expression<Func<TEntity, bool>> filter, params
Expression<Func<TEntity, object>>[] includes)
    {
        IQueryable<TEntity> dbQuery = _context.Set<TEntity>();
        foreach (var navigationProperty in includes)
        {
            dbQuery = dbQuery.Include(navigationProperty);
        }

        var query = dbQuery.AsNoTracking().Where(filter).AsQueryable();
```

```
            return query.Count();
        }

        public T Max<T>(Expression<Func<TEntity, bool>> filter, Expression<Func<TEntity,
T>> max, params Expression<Func<TEntity, object>>[] includes)
        {
            IQueryable<TEntity> dbQuery = _context.Set<TEntity>();
            foreach (var navigationProperty in includes)
            {
                dbQuery = dbQuery.Include(navigationProperty);
            }

            var query = dbQuery.Where(filter).AsQueryable();

            var any = query.Any();
            return any ? query.Max(max) : default(T);
        }

        public void Insert(params TEntity[] items)
        {
            if (items == null)
            {
                throw new NullReferenceException("There are no items to insert");
            }

            foreach (var item in items)
            {
                _dbSet.Add(item);
            }
        }

        public virtual void Delete(params TEntity[] items)
        {
            if (items == null)
            {
                throw new NullReferenceException("There are no items to delete");
            }

            if (items.Any(x => x is IDeleted))
            {
                throw new NotSupportedException("Entity of this type only supports logical
deletion");
            }

            _dbSet.RemoveRange(items);
        }

        public void Update(params TEntity[] items)
        {
            if (items == null)
```

```
    {
        throw new NullReferenceException("There are no items to update");
    }
    foreach (var item in items)
    {
        //var entity = _dbSet.FirstOrDefault(f => f.Id == item.Id);
        //if (entity != null)
        //{
        //    _context.Entry(entity).State = EntityState.Detached;
        //}

        //_dbSet.Attach(item);
        //_context.Entry(item).State = EntityState.Modified;

        _dbSet.AddOrUpdate(item);
    }
}

public async Task<TEntity[]> GetAllAsync(
    Expression<Func<TEntity, bool>> predicate,
    Func<IQueryable<TEntity>, IOrderedQueryable<TEntity>> orderBy,
    params Expression<Func<TEntity, object>>[] includeProperties)
{
    var entities = FilterQuery(orderBy, predicate, includeProperties);
    return await entities.ToArrayAsync();
}

public async Task<PaginatedList<TEntity>> GetPaginatedAsync(
    int pageIndex,
    int pageSize,
    Expression<Func<TEntity, bool>> predicate,
    Func<IQueryable<TEntity>, IOrderedQueryable<TEntity>> orderBy,
    params Expression<Func<TEntity, object>>[] includeProperties)
{
    var entities = FilterQuery(orderBy, predicate, includeProperties);
    var total = await entities.CountAsync();
    entities = entities.Paginate(pageIndex, pageSize);
    var list = await entities.ToListAsync();
    return list.ToPaginatedList(pageIndex, pageSize, total);
}
public async Task<ScrollableList<TEntity>> GetScrollableAsync(
    int skip,
    int take,
    Expression<Func<TEntity, bool>> predicate,
    Func<IQueryable<TEntity>, IOrderedQueryable<TEntity>> orderBy,
    params Expression<Func<TEntity, object>>[] includeProperties)
{
    var entities = FilterQuery(orderBy, predicate, includeProperties);
    var total = await entities.CountAsync();
    entities = entities.Scrollable(skip, take);
    var list = await entities.ToListAsync();
```

```
            return list.ToScrollableList(skip, take, total);
        }

        private IQueryable<TEntity> FilterQuery(
            Func<IQueryable<TEntity>, IOrderedQueryable<TEntity>> orderBy,
            Expression<Func<TEntity, bool>> predicate,
            Expression<Func<TEntity, object>>[] includeProperties)
        {
            var entities = IncludeProperties(includeProperties);
            entities = (predicate != null) ? entities.AsExpandable().Where(predicate) : entities;
            if (orderBy != null)
            {
                entities = orderBy(entities);
            }
            return entities;
        }

        private IQueryable<TEntity> IncludeProperties(params Expression<Func<TEntity,
object>>[] includeProperties)
        {
            IQueryable<TEntity> entities = _dbSet.AsNoTracking();

            foreach (var includeProperty in includeProperties)
            {
                entities = entities.Include(includeProperty);
            }

            return entities;
        }

        private IQueryable<TEntity> GetSingleInternal(Expression<Func<TEntity, bool>> filter,
            params Expression<Func<TEntity, object>>[] includes)
        {
            var dbQuery = IncludeProperties(includes);

            return dbQuery.Where(filter).AsNoTracking();
        }

        public IQueryable<TEntity> Collection => _dbSet.AsNoTracking();

        public IQueryable<TEntity> CollectionWithTracking => _dbSet;
    }
}

/* Листинг класса GenericRepository */
using System;
using System.Collections;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Text;
```

```
using System.Threading;
using System.Threading.Tasks;
using Contracts;
using Data.Implementations.Context;
using Data.Interfaces;
using Data.Interfaces.Repositories;
using Data.Repository;

namespace Data
{
   public class UnitOfWork : IUnitOfWork
   {
      private readonly DataContext _context;
      private Hashtable _repositories;

      public UnitOfWork(DataContext context)
      {
         _context = context;
      }


      public IGenericRepository<TEntity> Repository<TEntity>() where TEntity : class, IEntity
      {
         if (_repositories == null)
         {
            _repositories = new Hashtable();
         }
         var type = typeof(TEntity);
         var typeName = type.Name;

         if (!_repositories.ContainsKey(typeName))
         {
            var repositoryType = typeof(GenericRepository<>);
            _repositories.Add(typeName,
Activator.CreateInstance(repositoryType.MakeGenericType(typeof(TEntity)), _context));
         }

         return (IGenericRepository<TEntity>)_repositories[typeName];
      }

      public int SaveChanges()
      {
         return _context.SaveChanges();
      }

      public async Task<int> SaveChangesAsync()
      {
         return await _context.SaveChangesAsync();
      }

      public async Task<int> SaveChangesAsync(CancellationToken cancellationToken)
```

```
        {
            return await _context.SaveChangesAsync(cancellationToken);
        }
    }
}

/* Листинг интерфейса ICriteriaService */

using Services.DTO.Criteria;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Services.Interfaces
{
    public interface ICriteriaService
    {
        Task<int> AddOrUpdateCriteriaAsync(CriteriaDTO dto);
        Task DeleteAsync(params int[] ids);
        Task<List<CriteriaDTO>> GetAllAsync();
        Task<CriteriaDTO> GetByIdAsync(int id);
    }
}


/* Листинг интерфейса IExerciseService */
using Services.DTO.Exercise;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Services.Interfaces
{
    public interface IExerciseService
    {
        Task<int> AddOrUpdateExerciseAsync(ExerciseDetailsDTO dto);
        Task DeleteAsync(params int[] ids);
        Task<List<ExerciseDTO>> GetAllAsync();
        Task<ExerciseDetailsDTO> GetByIdAsync(int id);
    }
}

/* Листинг интерфейса IProfessionService */

using Services.DTO.Profession;
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Services.Interfaces
{
   public interface IProfessionService
   {
      Task<int> AddOrUpdateProfessionAsync(ProfessionDetailsDTO dto);
      Task DeleteAsync(params int[] ids);
      Task<List<ProfessionDTO>> GetAllAsync();
      Task<ProfessionDetailsDTO> GetByIdAsync(int id);
      Task<List<IGrouping<string, ProfessionDTO>>> GetAllGrouped();
   }
}

/* Листинг интерфейса ISettingsService */
using Services.DTO.Settings;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Services.Interfaces
{
   public interface ISettingsService
   {
      Task UpdateProfessionAsync(int userId, int professionId);
      Task UpdateDefaultTrainingTimesAsync(int userId, List<TimeSpan> times);
      Task<SettingsDTO> GetSettingsAsync(int userId);
   }
}

/* Листинг интерфейса IUserTrainingService */
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Services.Interfaces
{
   public interface IUserTrainingService
   {
      Task<UserTraining> GetUserTraining(int userId);
      Task CompleteTraining(int userId, int trainingId);
   }
}

/* Листинг интерфейса CriteriaService */
```

```
using Data.Interfaces;
using Data.Interfaces.Repositories;
using Entity.Domain;
using Services.DTO.Criteria;
using Services.Interfaces;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data.Entity;
using AutoMapper;
using Z.EntityFramework.Plus;

namespace Services
{
   public class CriteriaService : ICriteriaService
   {
      private readonly IUnitOfWork _unitOfWork;
      private readonly IGenericRepository<Criteria> _criteriaRepository;
      public CriteriaService(IUnitOfWork unitOfWork)
      {
         _unitOfWork = unitOfWork;
         _criteriaRepository = _unitOfWork.Repository<Criteria>();
      }
      public async Task<int> AddOrUpdateCriteriaAsync(CriteriaDTO dto)
      {
         if (dto == null)
         {
            throw new ArgumentNullException(nameof(dto));
         }
         var criteria = Mapper.Map<CriteriaDTO, Criteria>(dto);
         var criteriaInDb = await
_criteriaRepository.CollectionWithTracking.FirstOrDefaultAsync(f => dto.Id == f.Id);
         if (criteriaInDb == null)
         {
            _criteriaRepository.Insert(criteria);
         }
         else
         {
            Mapper.Map(criteria, criteriaInDb);
            _criteriaRepository.Update(criteriaInDb);
         }
         await _unitOfWork.SaveChangesAsync();
         return criteria.Id;
      }

      public async Task DeleteAsync(params int[] ids)
      {
         await _criteriaRepository.Collection.Where(f => ids.Any(z => z ==
f.Id)).DeleteAsync();
```

```
        }

        public async Task<List<CriteriaDTO>> GetAllAsync()
        {
            return await _criteriaRepository.Collection.Select(f => new CriteriaDTO
            {
                Id = f.Id,
                Name = f.Name
            }).ToListAsync();
        }

        public async Task<CriteriaDTO> GetByIdAsync(int id)
        {
            return await _criteriaRepository.Collection.Where(f => f.Id == id).Select(f => new
CriteriaDTO
            {
                Id = f.Id,
                Name = f.Name
            }).FirstOrDefaultAsync();
        }
    }
}


/* Листинг интерфейса ExerciseService */
using AutoMapper;
using Data.Interfaces;
using Data.Interfaces.Repositories;
using Entity.Domain;
using Services.DTO.Exercise;
using Services.DTO.ExerciseCriteria;
using Services.Interfaces;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Z.EntityFramework.Plus;

namespace Services
{
    public class ExerciseService : IExerciseService
    {
        private readonly IUnitOfWork _unitOfWork;
        private readonly IGenericRepository<Exercise> _exerciseRepository;
        private readonly IGenericRepository<ExerciseCriteria> _exerciseCriteriaRepository;
        public ExerciseService(IUnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
            _exerciseRepository = _unitOfWork.Repository<Exercise>();
```

```
            _exerciseCriteriaRepository = _unitOfWork.Repository<ExerciseCriteria>();
        }
        public async Task<int> AddOrUpdateExerciseAsync(ExerciseDetailsDTO dto)
        {
            var exercise = Mapper.Map<ExerciseDetailsDTO, Exercise>(dto);
            var exerciseInDb = await _exerciseRepository.CollectionWithTracking.Include(z =>
z.ExerciseCriterias).FirstOrDefaultAsync(f => f.Id == exercise.Id);
            if (exerciseInDb == null)
            {
                _exerciseRepository.Insert(exercise);
            }
            else
            {
                Mapper.Map(exercise, exerciseInDb);
                _exerciseCriteriaRepository.Delete(exerciseInDb.ExerciseCriterias.ToArray());
                exerciseInDb.ExerciseCriterias.AddRange(exercise.ExerciseCriterias);
                _exerciseRepository.Update(exerciseInDb);
            }
            await _unitOfWork.SaveChangesAsync();
            return exercise.Id;
        }

        public async Task DeleteAsync(params int[] ids)
        {
            await _exerciseRepository.Collection.Where(f => ids.Any(z => z ==
f.Id)).DeleteAsync();
        }

        public async Task<List<ExerciseDTO>> GetAllAsync()
        {
            return await _exerciseRepository.Collection.Select(f => new ExerciseDTO
            {
                Id = f.Id,
                Name = f.Name,
                VideoUrl = f.VideoUrl
            }).ToListAsync();
        }

        public async Task<ExerciseDetailsDTO> GetByIdAsync(int id)
        {
            var exercise = await _exerciseRepository.Collection.Include(f =>
f.ExerciseCriterias).FirstOrDefaultAsync(f => f.Id == id);
            return Mapper.Map<Exercise, ExerciseDetailsDTO>(exercise);
        }
    }
}
```

/* Листинг интерфейса ProfessionService */

```
using AutoMapper;
using Data.Interfaces;
using Data.Interfaces.Repositories;
```

```
using Entity.Domain;
using Services.DTO.Profession;
using Services.DTO.ProfessionCriteria;
using Services.Interfaces;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Z.EntityFramework.Plus;

namespace Services
{
   public class ProfessionService : IProfessionService
   {
      private readonly IUnitOfWork _unitOfWork;
      private readonly IGenericRepository<Profession> _professionRepository;
      private readonly IGenericRepository<ProfessionCriteria> _professionCriteriaRepository;
      public ProfessionService(IUnitOfWork unitOfWork)
      {
         _unitOfWork = unitOfWork;
         _professionRepository = _unitOfWork.Repository<Profession>();
         _professionCriteriaRepository = _unitOfWork.Repository<ProfessionCriteria>();
      }
      public async Task<int> AddOrUpdateProfessionAsync(ProfessionDetailsDTO dto)
      {
         var profession = Mapper.Map<ProfessionDetailsDTO, Profession>(dto);
         var professionInDb = await _professionRepository.CollectionWithTracking.Include(z
=> z.ProfessionCriterias).FirstOrDefaultAsync(f => f.Id == profession.Id);
         if (professionInDb == null)
         {
            _professionRepository.Insert(profession);
         }
         else
         {
            Mapper.Map(profession, professionInDb);

_professionCriteriaRepository.Delete(professionInDb.ProfessionCriterias.ToArray());
            professionInDb.ProfessionCriterias.AddRange(profession.ProfessionCriterias);
            _professionRepository.Update(professionInDb);
         }
         await _unitOfWork.SaveChangesAsync();
         return profession.Id;
      }

      public async Task DeleteAsync(params int[] ids)
      {
         await _professionRepository.Collection.Where(f => ids.Any(z => z ==
f.Id)).DeleteAsync();
      }
```

```
        public async Task<List<ProfessionDTO>> GetAllAsync()
        {
            return await _professionRepository.Collection.Select(f => new ProfessionDTO
            {
                Id = f.Id,
                Name = f.Name
            }).ToListAsync();
        }

        public async Task<List<IGrouping<string, ProfessionDTO>>> GetAllGrouped()
        {
            return await _professionRepository.Collection.Select(f => new ProfessionDTO
            {
                Id = f.Id,
                Name = f.Name
            }).GroupBy(s=>s.Name.ToUpper().Substring(0,1)).ToListAsync();

        }

        public async Task<ProfessionDetailsDTO> GetByIdAsync(int id)
        {
            var profession = await _professionRepository.Collection.Include(f =>
f.ProfessionCriterias).FirstOrDefaultAsync(f => f.Id == id);
            return Mapper.Map<Profession, ProfessionDetailsDTO>(profession);
        }
    }
}
/* Листинг интерфейса SettingsService */
using Data.Interfaces;
using Data.Interfaces.Repositories;
using Entity.Domain.Settings;
using Services.DTO.Settings;
using Services.Interfaces;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Z.EntityFramework.Plus;

namespace Services
{
    public class SettingsService : ISettingsService
    {
        private readonly IUnitOfWork _unitOfWork;
        private readonly IGenericRepository<Settings> _settingsRepository;
        public SettingsService(IUnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
```

```
            _settingsRepository = _unitOfWork.Repository<Settings>();
        }

        public async Task<SettingsDTO> GetSettingsAsync(int userId)
        {
            var result =  await
_settingsRepository.Collection.Include(s=>s.Profession).Include(s=>s.DefaultTrainingTimes).
FirstOrDefaultAsync(s => s.Id == userId);
            if(result != null)
            {
                return new SettingsDTO
                {
                    Profession = result.Profession != null ? new SettingsProfessionDTO
                    {
                        Id = result.Profession.Id,
                        Name = result.Profession.Name
                    } : null,
                    PreferredTrainingTime = result.DefaultTrainingTimes.Select(s =>
s.Value).ToList()
                };
            }
            return null;
        }

        public async Task UpdateDefaultTrainingTimesAsync(int userId, List<TimeSpan> times)
        {
            var settings = await _settingsRepository.CollectionWithTracking.Include(s =>
s.DefaultTrainingTimes).FirstOrDefaultAsync(s => s.Id == userId);
            if (settings == null)
            {
                return;
            }
            settings.DefaultTrainingTimes.Clear();
            var timesToAdd = times.Select(z => new TrainingTime
            {
                Value = z
            }).ToList();
            settings.DefaultTrainingTimes.AddRange(timesToAdd);
            await _unitOfWork.SaveChangesAsync();
        }

        public async Task UpdateProfessionAsync(int userId, int professionId)
        {
            await _settingsRepository.Collection.Where(s => s.Id == userId).UpdateAsync(z =>
new Settings
            {
                ProfessionId = professionId
            });
        }
    }
}
```

```
/*Контроллеры*/
using Entity.Domain.Identity;
using Entity.Domain.Settings;
using Entity.Identity;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.OAuth;
using Services.DTO.Settings;
using Services.Interfaces;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Security.Claims;
using System.Threading.Tasks;
using System.Web.Http;
using WebUI.Identity;
using WebUI.Models.Identity;

namespace WebUI.Controllers.API
{
    [RoutePrefix("api/account")]
    public class AccountController : ApiController
    {
        private ApplicationSignInManager _signInManager;
        private ApplicationUserManager _userManager;
        private readonly IIdentityMessageService _messageService;
        private readonly IAuthenticationManager _authenticationManager;
        private readonly ISettingsService _settingsService;
        public AccountController(
            ApplicationUserManager userManager,
            ApplicationSignInManager signInManager,
            IIdentityMessageService messageService,
            IAuthenticationManager authenticationManager,
            ISettingsService settingsService)
        {
            _signInManager = signInManager;
            _userManager = userManager;
            _messageService = messageService;
            _authenticationManager = authenticationManager;
            _settingsService = settingsService;
        }

        // POST: /Account/Login
        [HttpPost]
        [Route("sign-in")]
        [AllowAnonymous]
        public async Task<IHttpActionResult> Login(LoginViewModel model)
```

```
        {
            if (!ModelState.IsValid)
            {
                return BadRequest("Invalid model");
            }

            // This doesn't count login failures towards account lockout
            // To enable password failures to trigger account lockout, change to shouldLockout: true
            var user = await _userManager.FindAsync(model.Email, model.Password);
            if (user == null)
            {
                return BadRequest("No such user!");
            }
            var token = await GenerateTokenAsync(user);
            var roles = await _userManager.GetRolesAsync(user.Id);
            var result = new
            {
                Id = user.Id,
                Name = user.UserName,
                Roles = await _userManager.GetRolesAsync(user.Id),
                Token = token,
                Settings = roles.Any(z => z == Roles.User) ? await
_settingsService.GetSettingsAsync(user.Id) : default(SettingsDTO)
            };
            return Ok(result);
        }
        // POST: /Account/Register
        [HttpPost]
        [Route("sign-up")]
        [AllowAnonymous]
        public async Task<IHttpActionResult> Register(RegisterViewModel model)
        {
            if (ModelState.IsValid)
            {
                var user = new User() { UserName = model.Email, Email = model.Email, Settings =
new Settings() };
                var result = await _userManager.CreateAsync(user, model.Password);
                if (result.Succeeded)
                {
                    var userInDb = await _userManager.FindByEmailAsync(user.Email);
                    result = await _userManager.AddToRoleAsync(userInDb.Id, Roles.User);
                    if (result.Succeeded)
                    {
                        var token = await GenerateTokenAsync(userInDb);
                        return Ok(new
                        {
                            Id = user.Id,
                            Name = user.UserName,
                            Roles = await _userManager.GetRolesAsync(user.Id),
                            Token = token
                        });
```

```
                }
            }
        }
        return BadRequest("Error");
    }

    // GET: /Account/Register
    [HttpGet]
    [Route("check-login")]
    [AllowAnonymous]
    public async Task<IHttpActionResult> CheckLogin()
    {
        var userId = User.Identity.GetUserId<int>();
        if (ModelState.IsValid)
        {
            var user = await _userManager.FindByIdAsync(userId);
            if (user != null)
            {
                return Ok(new
                {
                    Email = user.Email,
                    Id = user.Id,
                    Roles = user.Roles
                });
            }
        }

        // If we got this far, something failed, redisplay form
        return BadRequest("Error");
    }

    private async Task<string> GenerateTokenAsync(User user)
    {
        var tokenExpiration = Startup.OAuthServerOptions.AccessTokenExpireTimeSpan;
        var identity = new ClaimsIdentity(OAuthDefaults.AuthenticationType);
        identity.AddClaim(new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()));
        identity.AddClaim(new Claim(ClaimTypes.Role, string.Join(",", await
_userManager.GetRolesAsync(user.Id))));

        var props = new AuthenticationProperties()
        {
            IssuedUtc = DateTime.UtcNow,
            ExpiresUtc = DateTime.UtcNow.Add(tokenExpiration),
        };
        var ticket = new AuthenticationTicket(identity, props);
        var accessToken = Startup.OAuthServerOptions.AccessTokenFormat.Protect(ticket);
        return accessToken;
    }
  }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace WebUI.Controllers.API.Features
{
   public class UserTrainingController : ApiController
   {
   }
}


using Services.DTO.Exercise;
using Services.DTO.ExerciseCriteria;
using Services.Interfaces;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using System.Web.Http;

namespace WebUI.Controllers.API.Admin
{
   [RoutePrefix("api/admin/exercise")]
   public class AdminExerciseController : ApiController
   {
      private readonly IExerciseService _exerciseService;
      private readonly ICriteriaService _criteriaService;
      public AdminExerciseController(IExerciseService service, ICriteriaService
criteriaService)
      {
         _exerciseService = service;
         _criteriaService = criteriaService;
      }
      // GET: api/ApiExercise
      [HttpGet]
      [Route("getAll")]
      public async Task<IHttpActionResult> GetAllAsync()
      {
         return Ok(await _exerciseService.GetAllAsync());
      }

      // GET: api/ApiExercise/5
      [HttpGet]
      [Route("get")]
      public async Task<IHttpActionResult> GetAsync(int? id = null)
```

```
        {
            var criterias = await _criteriaService.GetAllAsync();
            if (id.HasValue)
            {
                return Ok(new {
                    exercise = await _exerciseService.GetByIdAsync(id.Value),
                    criterias = criterias
                });
            }
            else
            {
                return Ok(new {
                    exercise = new ExerciseDetailsDTO
                    {
                        Criterias = new List<ExerciseCriteriaDTO>()
                    },
                    criterias = criterias
                });
            }
        }


        [HttpPost]
        [Route("save")]
        public async Task<IHttpActionResult> Save(ExerciseDetailsDTO model)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest("Invalid Model");
            }
            return Ok(await _exerciseService.AddOrUpdateExerciseAsync(model));
        }

        [HttpDelete]
        [Route("delete")]
        public async Task<IHttpActionResult> Delete([FromUri]int[] ids)
        {
            await _exerciseService.DeleteAsync(ids);
            return Ok();
        }
    }
}

using Services.DTO.Profession;
using Services.DTO.ProfessionCriteria;
using Services.Interfaces;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
```

```
using System.Threading.Tasks;
using System.Web.Http;

namespace WebUI.Controllers.API.Admin
{
   [RoutePrefix("api/admin/profession")]
   public class AdminProfessionController : ApiController
   {
      private readonly IProfessionService _professionService;
      private readonly ICriteriaService _criteriaService;
      public AdminProfessionController(IProfessionService service, ICriteriaService
criteriaService)
      {
         _professionService = service;
         _criteriaService = criteriaService;
      }
      // GET: api/ApiProfession
      [HttpGet]
      [Route("getAll")]
      public async Task<IHttpActionResult> GetAllAsync()
      {
         return Ok(await _professionService.GetAllAsync());
      }

      // GET: api/ApiProfession/5
      [HttpGet]
      [Route("get")]
      public async Task<IHttpActionResult> GetAsync(int? id = null)
      {
         var criterias = await _criteriaService.GetAllAsync();
         if (id.HasValue)
         {
            return Ok(new
            {
               profession = await _professionService.GetByIdAsync(id.Value),
               criterias = criterias
            });
         }
         else
         {
            return Ok(new
            {
               profession = new ProfessionDetailsDTO
               {
                  Criterias = new List<ProfessionCriteriaDTO>()
               },
               criterias = criterias
            });
         }
      }
```

```
        [HttpPost]
        [Route("save")]
        public async Task<IHttpActionResult> Save(ProfessionDetailsDTO model)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest("Invalid Model");
            }
            return Ok(await _professionService.AddOrUpdateProfessionAsync(model));
        }

        [HttpDelete]
        [Route("delete")]
        public async Task<IHttpActionResult> Delete([FromUri]int[] ids)
        {
            await _professionService.DeleteAsync(ids);
            return Ok();
        }
    }
}

using Services.DTO.Profession;
using Services.Interfaces;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using System.Web.Http;

namespace WebUI.Controllers.API.Features
{
    //[Authorize]
    [RoutePrefix("api/professions")]
    public class ProfessionController : ApiController
    {
        private readonly IProfessionService _professionService;
        public ProfessionController(IProfessionService professionService)
        {
            _professionService = professionService;
        }
        [HttpGet]
        [Route("getAll")]
        public async Task<IHttpActionResult> GetAll()
        {
            var result = await _professionService.GetAllGrouped();
            return Ok(result.Select(f => {
                var dictionary = new Dictionary<string, List<ProfessionDTO>>();
                dictionary.Add(f.Key.ToUpper(), f.ToList());
```

```
                return dictionary;
            }));
        }
    }
}

using Microsoft.AspNet.Identity;
using Services.Interfaces;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using System.Web.Http;

namespace WebUI.Controllers.API.Mobile
{
    [RoutePrefix("api/settings")]
    [Authorize]
    public class SettingsController : ApiController
    {
        private readonly ISettingsService _settingsService;
        public SettingsController(ISettingsService settingsService)
        {
            _settingsService = settingsService;
        }

        [HttpGet]
        public async Task<IHttpActionResult> GetSettings()
        {
            var userId = User.Identity.GetUserId<int>();
            var settings = await _settingsService.GetSettingsAsync(userId);
            return Ok(settings);
        }

        [HttpPost]
        [Route("profession")]
        public async Task<IHttpActionResult> ChangeProfession(int professionId)
        {
            var userId = User.Identity.GetUserId<int>();
            await _settingsService.UpdateProfessionAsync(userId, professionId);
            return Ok();
        }

        [HttpPost]
        [Route("preferredTimes")]
        public async Task<IHttpActionResult> ChangePreferredTime(List<TimeSpan>
preferredTimes)
        {
            var userId = User.Identity.GetUserId<int>();
```

```
            await _settingsService.UpdateDefaultTrainingTimesAsync(userId, preferredTimes);
            return Ok();
        }
    }
}
/*Главный компонент приложения для ПК*/
<template>
  <v-app>
    <app-alerts></app-alerts>
    <app-loading></app-loading>
    <app-sidebar></app-sidebar>
    <app-toolbar></app-toolbar>
    <v-content>
      <router-view/>
    </v-content>
  </v-app>
</template>
<script>
import AppToolbar from "@/components/Shared/Layout/Toolbar";
import AppSidebar from "@/components/Shared/Layout/Sidebar";
import AuthGuard from "./router/auth-guard.js";
export default {
  data() {
    return {};
  },
  components: {
    AppToolbar,
    AppSidebar
  },
  name: "App"
};
</script>
<style>
@importurl(https://fonts.googleapis.com/css?family=Roboto:300,400,500,700|Material+Icons);
/* Global CSS */
.media-holder {
  position: relative;
  height: 0;
  padding-bottom: 56.25%;
  width: 100%;
}
.media-holder iframe {
  position: absolute;
  height: 100%;
  width: 100%;
  left: 0;
  top: 0;
}
</style>
/*Main.js*/
import Vue from 'vue'
```

```
import axios from 'axios'
import Vuetify from 'vuetify'
import 'vuetify/dist/vuetify.css'

import App from './App'
import router from './router'
import {
  store
} from './store'

import AlertsCmp from "./components/Shared/Alerts"
import LoadingCmp from "./components/Shared/Loading"

Vue.use(Vuetify, {
  theme: {
    primary: "#1565c0",
    secondary: "#424242",
    accent: "#82B1FF",
    error: "#FF5252",
    info: "#2196F3",
    success: "#4CAF50",
    warning: "#FFC107"
  }
});
Vue.component("app-alerts", AlertsCmp);
Vue.component("app-loading", LoadingCmp);
if (!process.env.IS_WEB) Vue.use(require('vue-electron'))
Vue.http = Vue.prototype.$http = axios
Vue.config.productionTip = false;

if (process.env.NODE_ENV === 'production') {
  axios.defaults.baseURL = "http://localhost:57327/";
}
/* eslint-disable no-new */
new Vue({
  components: {
    App
  },
  router,
  store,
  template: '<App/>'
}).$mount('#app');

Storage.prototype.setObject = function(key, value) {
  this.setItem(key, JSON.stringify(value));
}

Storage.prototype.getObject = function(key) {
  var value = this.getItem(key);
  return value && JSON.parse(value);
}
```