Reusing common code via Nuget

(Demo prep)

- Remove package sources:
  dotnet nuget remove source "Package source 1"
- Reopen Postman

Refactoring common code

If we take a look at our ItemsRepository class we will notice that there's a lot of code that we will need to reuse in future microservices. Same for the MongoDb and Service Settings classes.

But before we can move anything to a new shared library we will need to do some good refactoring to keep the generic pieces separated from what is really needed in the Catalog microservice.

1. Extract IEntity (not IItem) interface from Item entity:

```
namespace Play.Catalog.Service.Entities
{
   public interface IEntity
   {
      Guid Id { get; set; }
   }
}
```

2. Move IEntity to new file

3. Generalize IItemsRepository into IRepository:

```
namespace Play.Catalog.Service.Repositories
{
   public interface IRepository<T> where T : IEntity
   {
      Task CreateAsync(T entity);
      Task<IReadOnlyCollection<T>> GetAllAsync();
      Task<T> GetAsync(Guid id);
      Task RemoveAsync(Guid id);
      Task UpdateAsync(T entity);
   }
}
```

4. Generalize ItemsRepository into MongoRepository:

```
namespace Play.Catalog.Service.Repositories
{

    public class MongoRepository<T> : IRepository<T> where T : IEntity
    {
        private readonly IMongoCollection<T> dbCollection;
        private readonly FilterDefinitionBuilder<T> filterBuilder = Builders<T>.Filter;

        public MongoRepository(IMongoDatabase database, string collectionName)
        {
            dbCollection = database.GetCollection<T>(collectionName);
        }

        public async Task<IReadOnlyCollection<T>> GetAllAsync()
        {
            return await dbCollection.Find(filterBuilder.Empty).ToListAsync();
        }

        public async Task<T> GetAsync(Guid id)
        {
            FilterDefinition<T> filter = filterBuilder.Eq(entity => entity.Id, id);
            return await dbCollection.Find(filter).FirstOrDefaultAsync();
        }

        public async Task CreateAsync(T entity)
        {
            if (entity == null)
            {
                throw new ArgumentNullException(nameof(entity));
            }

            await dbCollection.InsertOneAsync(entity);
        }

        public async Task UpdateAsync(T entity)
        {
            if (entity == null)
            {
                throw new ArgumentNullException(nameof(entity));
            }

            FilterDefinition<T> filter = filterBuilder.Eq(existingEntity => existingEntity.Id, entity.Id);
            await dbCollection.ReplaceOneAsync(filter, entity);
        }

        public async Task RemoveAsync(Guid id)
        {
            FilterDefinition<T> filter = filterBuilder.Eq(entity => entity.Id, id);
            await dbCollection.DeleteOneAsync(filter);
```

```
    }
  }
}
```

5. Fix ItemsController:

```
public class ItemsController : ControllerBase
{
  private readonly IRepository<Item> itemsRepository;

  public ItemsController(IRepository<Item> itemsRepository)
  {
    this.itemsRepository = itemsRepository;
  }
```

6. Replace repository registration in Startup:

```
services.AddSingleton<IRepository<Item>>(serviceProvider =>
{
  var database = serviceProvider.GetService<IMongoDatabase>();
  return new MongoRepository<Item>(database, "items");
});
```

At this point we have generalized our repository class so that it can be used with any type of entity. However, it feels like we are writing too much code on Startup to register the MongoDB related classes. In the next lesson we will introduce a couple of handy extension methods that will simplify things quite a bit for this and any future microservices.