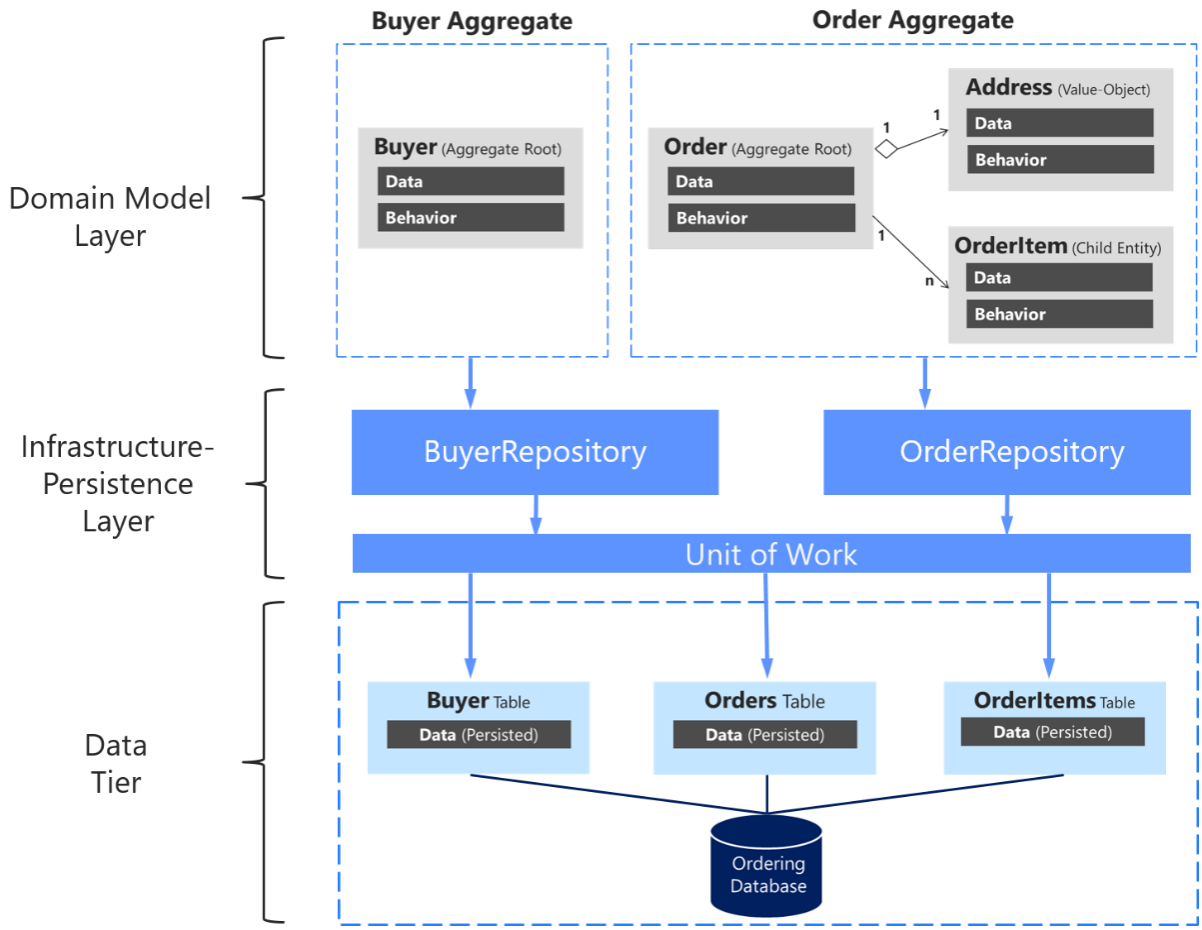
Tutorial 10

Objectives

- Repository pattern
- Dependency injection

To follow this tutorial, you first need to complete the previous one, Tutorial 09. Go back and complete that one first if you have not done it already.

In this tutorial we will see how we can implement a repository pattern. The repository pattern works as an abstraction layer on top of the database access code (ORM).



Base repository

We start by creating IBaseRepository interface which should contain operations which are going to be shared by all entities. This should be a generic interface so that we can support any entity. Those operations are Add, Update, Delete, Find, Get all, Save changes.

```
2 references
public interface IBaseRepository<T>
{
    2 references
    void Add(T entity);
    2 references
    void Update(T entity);
    2 references
    void DeleteAsync(T entity);
    3 references
    Task<T> FindAsync(Guid id);
    2 references
    Task<IEnumerable<T>> GetAllAsync();
    4 references
    Task<bool> SaveChangesAsync();
}
```

IBaseRepository interface with all the operations.

So far we only have an abstraction (interface) which we can think of as a contract. Next we need to create BaseRepository class which will implement the interface and provide an implementation.

The class takes DbContext through dependency injection.

```
public class BaseRepository<T> : IBaseRepository<T>, IDisposable where T : class
{
    protected ApplicationDbContext _context;

    1 reference
    public BaseRepository(ApplicationDbContext dbContext)
    {
        _context = dbContext;
    }

    2 references
    public void Add(T entity)
    {
        _context.Add(entity);
    }

    2 references
    public void Update(T entity)
    {
        _context.Update(entity);
    }

    2 references
    public void DeleteAsync(T entity)
    {
        _context.Remove(entity);
    }

    3 references
    public virtual async Task<T> FindAsync(Guid id)
    {
        return await _context.Set<T>().FindAsync(id);
    }

    2 references
    public virtual async Task<IEnumerable<T>> GetAllAsync()
    {
        return await _context.Set<T>().ToListAsync();
    }
```

```
2 references
public virtual async Task<IEnumerable<T>> GetAllAsync()
{
    return await _context.Set<T>().ToListAsync();
}

4 references
public async Task<bool> SaveChangesAsync()
{
    // return true if 1 or more entities were changed
    var saveResult = await _context.SaveChangesAsync() > 0;
    return saveResult;
}

// Free up the memory used by DbContext
0 references
public void Dispose()
{
    if (_context != null)
    {
        _context.Dispose();
        _context = null;
    }
    GC.SuppressFinalize(this);
}
```

Implementation of the BaseRepository class.

Todo repository

If we want to extend the functionality of the base repository and perform queries which involve properties specific to an entity, then for such an entity we have to create a separate repository. That repository class should inherit from BaseRepository so that we can get the shared method and functionalities.

We will use Todos model as an example and for this entity we will create a separate repository. First, we create an interface that contains the methods which will be implemented in our repository. This interface should also inherit from IBaseRepository.

```
4 references
public interface ITodosRepository : IBaseRepository<Todo>
{
    2 references
    public bool Any(Guid id);
}
```

Next, we create TodosRepository class which should inherit from BaseRepository and implement ITodosRepository.

```
2 references
public class TodosRepository : BaseRepository<Todo>, ITodosRepository
{
    0 references
    public TodosRepository(ApplicationDbContext context) : base(context)
    {
    }

    2 references
    public bool Any(Guid id)
    {
        return _context.Todos.Any(t => t.Id == id);
    }
}
```

This is what our file structure should look like.

```
Solution 'HostedWithIdentity' (3 of 3 projects)
  HostedWithIdentity.Client
  HostedWithIdentity.Server
    Connected Services
    Dependencies
    Properties
    Areas
    Controllers
      OidcConfigurationController.cs
      TodosController.cs
      WeatherForecastController.cs
    Data
      Migrations
      Repositories
        BaseRepository.cs
        IBaseRepository.cs
        ITodosRepository.cs
        TodosRepository.cs
      ApplicationDbContext.cs
    Models
    Pages
    appsettings.Development.json
    appsettings.json
    Program.cs
    Startup.cs
  HostedWithIdentity.Shared
    Dependencies
    Todo.cs
    WeatherForecast.cs
```

Dependency Injection

The benefit of using interfaces is that we can register an interface and it's implementation in dependency injection and provide the interface to the classes which need to get an instance of the implementation.

Registering Todo Service in ConfigureServices method of the Startup.cs file.

Add the following line at the end of the method.

services.AddScoped<ITodosRepository, TodosRepository>();

```
Startup.cs   BaseRepository.cs   WeatherForecast.cs   TodosRepository.cs   ITodosRepository.cs   IBaseRepository.cs   Index.razor   Todo
HostedWithIdentity.Server   HostedWithIdentity.Server.Startup   ConfigureSer

27          // For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?LinkID=
            0 references
28          public void ConfigureServices(IServiceCollection services)
29          {
30              services.AddDbContext<ApplicationDbContext>(options =>
31                  options.UseSqlServer(
32                      Configuration.GetConnectionString("DefaultConnection")));
33
34
35              services.AddDatabaseDeveloperPageExceptionFilter();
36
37              services.AddDefaultIdentity<ApplicationUser>(options => options.SignIn.RequireConfirmedAccount = true)
38                  .AddEntityFrameworkStores<ApplicationDbContext>();
39
40              services.AddIdentityServer()
41                  .AddApiAuthorization<ApplicationUser, ApplicationDbContext>();
42
43              services.AddAuthentication()
44                  .AddIdentityServerJwt();
45
46              services.AddControllersWithViews();
47              services.AddRazorPages();
48
49              // Register the Swagger generator, defining 1 or more Swagger documents
50              services.AddSwaggerGen();
51
52              services.AddScoped<ITodosRepository, TodosRepository>();
53          }
```

Inject dependencies

Using TodoRepository through dependency injection in TodosController API as shown below. Please note, we pass an interface rather than the implementation (TodosRepository) which allows us to easily swap implementations. If we want to work with a different DbContext or a different database, this would require us to make just a single code change. That change would be made in the ConfigureServices method of Startup class.

```
[Authorize]
[ApiController]
[Route("api/[controller]")]
1 reference
public class TodosController : ControllerBase
{
    private readonly ApplicationDbContext _context;
    private readonly ITodosRepository _todosRepository;

    0 references
    public TodosController(ApplicationDbContext context, ITodosRepository todosRepository)
    {
        _context = context;
        _todosRepository = todosRepository;
    }

    // GET: api/Todos
    [HttpGet]
    0 references
    public async Task<ActionResult<IEnumerable<Todo>>> GetTodos()
    {
        var items = await _todosRepository.GetAllAsync(); //_context.Todos.ToListAsync();
        return items.ToList();
    }

    // GET: api/Todos/5
    [HttpGet("{id}")]
    0 references
    public async Task<ActionResult<Todo>> GetTodo(Guid id)
    {
        var todo = await _todosRepository.FindAsync(id); // _context.Todos.FindAsync(id);
```

Learn more

https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design