Multi-Layered Architectures in .Net Workshop

Kristijan Horvat
Software Architect
kristijan@mono-software.com

mono.track

Table of Contents

- Layered Architecture
  - Expose data entities to outside world - Pitfall
  - Create service unit tests
  - Abstract repository layer
- Onion/Hexagonal Onion Architecture
  - Investigate project structure
  - Configure dependency injection
  - Properly expose domain models
  - Switch the service layer with new one - IoC

GitHub location
https://github.com/MonoSoftware/code-camp-multi-layered-architectures

Checkout layered branch

- git clone https://github.com/MonoSoftware/code-camp-multi-layered-architectures.git
- git checkout layered

Expose data entities to outside world - Pitfall

- UserEntity
    - add Password property to UserEntity
- CompanyEntity
    - add Balance property to CompanyEntity

Notice how private data is leaking to outside world through presentation layer

Create service unit tests

- In the Project.Service.Tests find UserService.Test.cs
- Try to run and fix unit test

Notice that you need to abstract the repository or introduce the polymorphism in order to properly test methods

Abstract repository – polymorphism

Before

```
public List<UserEntity> Get()
{
   return UserRepository.Storage;
}
```

After

```
public virtual List<UserEntity> Get()
{
   return UserRepository.Storage;
}
```

Abstract repository layer

- In the Project.Repository find UserRepository.cs
- Abstract repository (in order to do IoC)
- Inject repository contract into service

Notice that a lot of refactoring is needed in order to abstract the **repository. It's much easier to abstract things right away than afterwards**

Abstract repository layer

```
public interface IUserRepository
{
        #region Methods

        List<IUser> Get();

        #endregion Methods
}
```

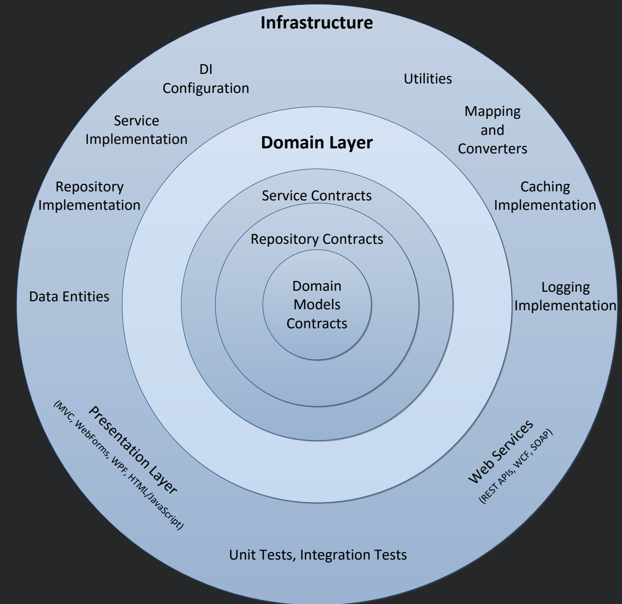
```
public class UserService : IUserService
    {
        public UserService(IUserRepository repository)
        {
            this.Repository = repository;
        }

        protected IUserRepository Repository { get; private set; }

        public List<IUser> Get()
        {
            return Repository.Get();
        }
}
```

Checkout onion branch

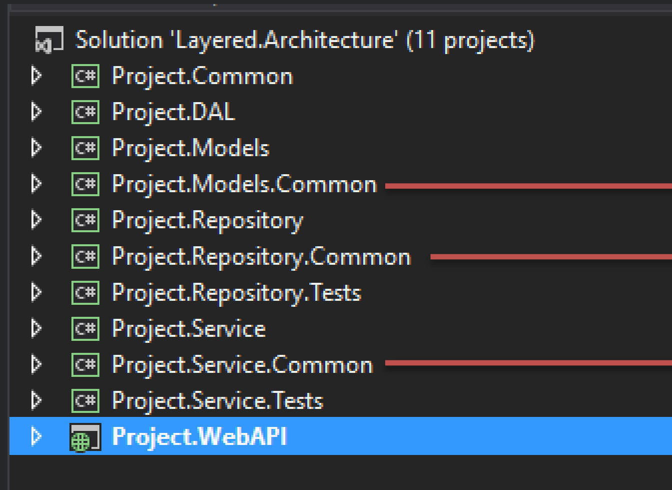
- git clone https://github.com/MonoSoftware/code-camp-multi-layered-architectures.git
- git checkout onion

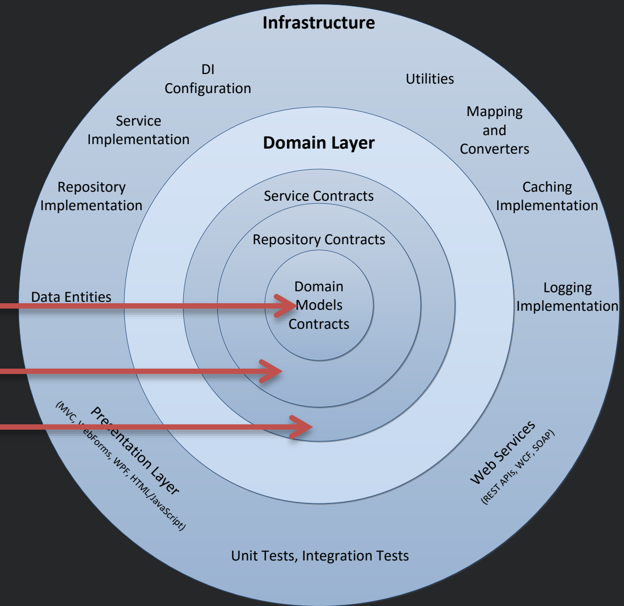
Investigate project structure

Solution 'Layered.Architecture' (11 projects)
- Project.Common
- Project.DAL
- Project.Models
- Project.Models.Common
- Project.Repository
- Project.Repository.Common
- Project.Repository.Tests
- Project.Service
- Project.Service.Common
- Project.Service.Tests
- Project.WebAPI

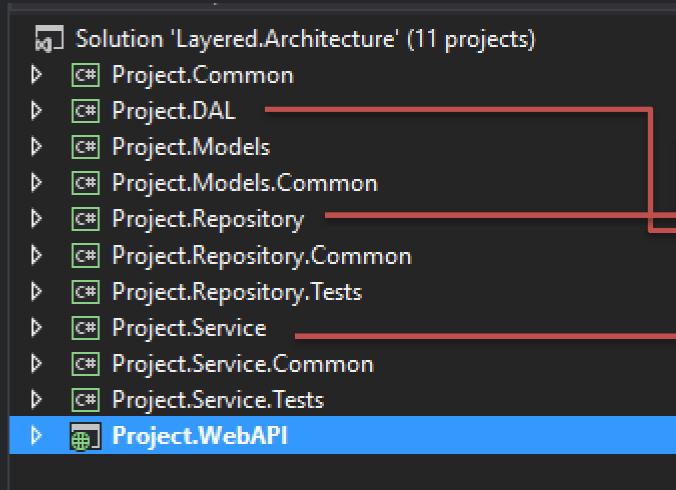


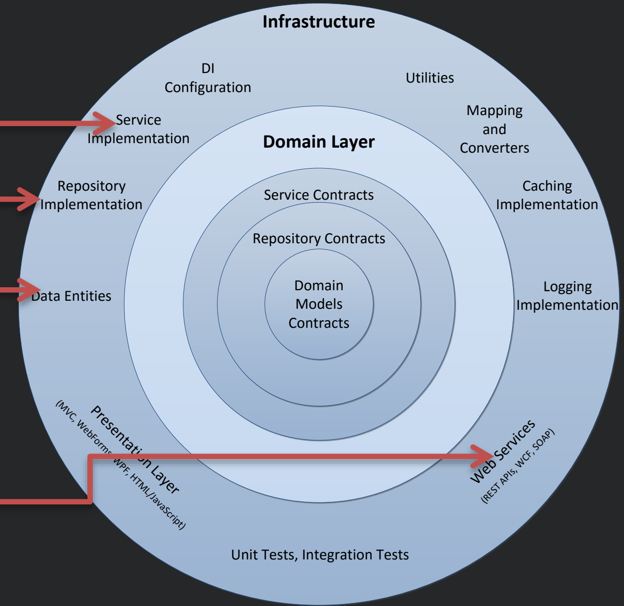
Investigate project structure





Investigate project structure





Investigate project structure

```
Solution 'Layered.Architecture' (11 projects)
  ▷ Project.Common
  ◢ Project.DAL
      ▷ Properties
      ▷ References
      ▷ CompanyEntity.cs
      ▷ UserEntity.cs
  ◢ Project.Models
      ▷ Properties
      ▷ References
      ▷ Company.cs
      ▷ User.cs
  ◢ Project.Models.Common
      ▷ Properties
      ▷ References
      ▷ ICompany.cs
      ▷ IUser.cs
  ◢ Project.Repository
      ▷ Properties
      ▷ References
      ▷ CompanyRepository.cs
      ▷ DIModule.cs
        packages.config
      ▷ UserRepository.cs
  ◢ Project.Repository.Common
      ▷ Properties
      ▷ References
      ▷ ICompanyRepository.cs
      ▷ IUserRepository.cs
  ◢ Project.Service
      ▷ Properties
      ▷ References
      ▷ CompanyService.cs
      ▷ DIModule.cs
        packages.config
      ▷ UserService.cs
  ◢ Project.Service.Common
      ▷ Properties
      ▷ References
      ▷ ICompanyService.cs
      ▷ IUserService.cs
  ▷ Project.Service.Tests
  ◢ Project.WebAPI
      ▷ Properties
      ▷ References
      ▷ App_Start
      ▷ CompanyController.cs
      ▷ Global.asax
        packages.config
      ▷ UserController.cs
      ▷ Web.config
```

Configure dependency injection

- Reference everything in Composition Root (WebAPI) and add bindings
- Reference everything in Composition Root (WebAPI) and add DI modules
- Dynamically load DI modules from all layers

Configure DI – reference everything

Composition Root -> NinjectWebCommon.cs

```
kernel.Bind<IUserRepository>().To<UserRepository>();
kernel.Bind<ICompanyRepository>().To<CompanyRepository>();
kernel.Bind<IUserService>().To<UserService>();
kernel.Bind<ICompanyService>().To<CompanyService>();
```

Note: If you register everything using this approach then references to all dependent projects are needed.

Configure DI – reference using DI Module

```
public class DIModule : Ninject.Modules.NinjectModule
{
   public override void Load()
   {
      kernel.Bind<IUserRepository>().To<UserRepository>();
      kernel.Bind<ICompanyRepository>().To<CompanyRepository>()
      }
}

public class DIModule : Ninject.Modules.NinjectModule
{
   public override void Load()
   {
      kernel.Bind<IUserService>().To<UserService>();
      kernel.Bind<ICompanyService>().To<CompanyService>();
   }
}
```

Composition Root -> NinjectWebCommon.cs

```
var kernel = new StandardKernel(
            new Project.Repository.DIModule(),
            new Project.Service.DIModule()
            );
```

Note: If you register everything using this approach you still need references to all dependent projects.

Configure DI – dynamically load DI Module

```
public class DIModule : Ninject.Modules.NinjectModule
{
    public override void Load()
    {
        kernel.Bind<IUserRepository>().To<UserRepository>();
        kernel.Bind<ICompanyRepository>().To<CompanyRepository>()
        }
}

public class DIModule : Ninject.Modules.NinjectModule
{
    public override void Load()
    {
        kernel.Bind<IUserService>().To<UserService>();
        kernel.Bind<ICompanyService>().To<CompanyService>();
    }
}
```

Composition Root -> NinjectWebCommon.cs

```
 var settings = new NinjectSettings();
settings.LoadExtensions = true;

settings.ExtensionSearchPatterns =
settings.ExtensionSearchPatterns.Union(new string[]
{ “Project.*.dll" }).ToArray();

var kernel = new StandardKernel(settings);
```

Note: This approach doesn’t need references to all dependent projects.

Properly expose domain models

- UserEntity, IUser & User
    - have Password property
- CompanyEntity, ICompany & Company
    - have Balance property

Notice that you can transform your domain models to hide private data and stop leaking data to outside world through presentation layer

Switch the service layer with new one - IoC

- Create Project.Service.New
- Implement
    - IUserService
    - ICompanyService
- Register new implementation with DI

If you have setup DI configuration properly you can switch the service layer very easily

Readings

- Architectural Patterns and Styles
  - https://msdn.microsoft.com/en-us/library/ee658117.aspx
- Layered Application Guidelines
  - https://msdn.microsoft.com/en-us/library/ee658109.aspx
- The Onion Architecture
  - http://jeffreypalermo.com/blog/the-onion-architecture-part-1/
- Hexagonal architecture
  - http://alistair.cockburn.us/Hexagonal+architecture
- The Clean Architecture
  - http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html
- Layers, Onions, Ports, Adapters: it's all the same
  - http://blog.ploeh.dk/2013/12/03/layers-onions-ports-adapters-its-all-the-same/
- Onion-izing your multi tier architecture
  - http://www.incredible-web.com/blog/the-onion-architecture/

Next Mono.Tracks on CodeCamp

- 16.12. - NodeJS Introduction & Workshop

We are hiring

Facebook: mono.software

Twitter: @monosoftware

E-mail: careers@mono.software

Thank you!

Questions?

mono.track