1 - Overview

Concepts

The principle is to move the complexity to the correct place, allowing to write stupid simple business logic.

- Your services are an entry point of your business logic, the service must be agnostic.
- Hosting specific technologies like gRPC, WebServices, WebAPI's, MVC Controllers, WCF, or any other technology to expose your business logic to the world, should not require any change behind your application services. Your application services don't need to understand specific hosting technologies to work with them.
- Any technology complexity is managed by a centralized implementation of architecture/infrastructure projects as abstractions.
- All abstraction must lead the implementation around abstracted technology but never can suppress at all the raw connector or provider it can be present to perform custom implementations or access specific features without compromise the abstraction.
- All direct access to raw connector or provider via abstraction must be reviewed periodically to analyze and decide if it can be promoted as a feature of the abstraction or not.
- Maybe you are looking for Mediators to perform a request /response strategy decoupled. Maybe you are looking for a way to reduce de dependency graph on your like explained at session *What are We Trying to Solve* at Simplifying Development and Separating Concerns with MediatR. The mediation on Oragon Architecture is injected between caller and callee at a simple method call between services. Any dependent service must be hosted anywhere without compromise your consume or need changes in the caller. On .NET Framework, we were able to replace a dependent application service, providing the client an server proxies for WCF, .NET Remoting, Web Services, Enterprise Service, enabling distribute specific services however you want only changing configurations, deciding as late as possible how to distribute my business logic in remote components. Spring.Services of Spring.NET are the base of this behavior, and many extensions are created on past to perform this. New compatible implementations are coming soon.

Around these concepts, some libraries were created to address the accidental complexity, implementing the infrastructure of these concepts in real life. These libraries help in developing service applications. It implements most common non-

Table of Contents

functional requirements using abstractions to permit that you write only one agnostic service layer.

Inversion of Control, Dependency Injection and Aspect Oriented Architecture

A few years ago, I've started using Spring.NET as a principal IoC Container. Spring.NET is a most important implementation about the inversion of control and dependency injection on .NET. Providing an XML declarative way to configure factory, dependency, and control flow behaviors. Spring.NET is helpful to connect dots in a long way of disconnected services and technologies. After .NET Core and .NET Standard be launched, I've tried to update the original project to compatibility with .NET Standard, but original Spring.NET has many, useless for me, projects. Instead, the Oragon base architecture, written in 2006, and rewrote sometimes later, need only of two projects: Spring.Core and Spring.Aop. That I've trying to show that port is simples, but I had no result with the original community. And so I forked this project creating Spring.Summer (a joke) and later renaming to Oragon.Spring. Oragon.Spring has only 2 projects on requirements. and we need only 2 I've decided port this project to .NET Standard and I've created Oragon.Spring a direct fork of Spring.NET implemented to be compatible with .NET Standard 2.

Exception Handling and Complex Logs

Exception Handling

You don't need handle exception if it doesn't represent a flow change on your business code. If your unique requirements to implement a try-catch is a logging, rollback the database transaction, or switch the exception to another generic, you don't need to write any try-catch on the service layer.

Complex Logs

Provide an abstraction to write complex logs, and delivery a base implementation to publish your logs on RabbitMQ. You can implement your own output, like File, Other MQ implementation or another strategy. Complex logs as defined as a log with custom properties and values. That provides contextual information about execution contexts. You can add what you need to explain better your log, like variables, parameters, and environment variables to help on troubleshooting.

Connected Service (need rewrite)

Connected Service provide abstractions to connect technologies like AMQP, HTTP, SOAP and others technologies on your agnostic services. It's helpful to delivery most common technology scenarios with the same codebase. A unique single method of your service can be exposed as: An HTTP route. An AMQP Queue Worker. A Stream Processor. A normal Method used by dependents.

All in one, without changes, without knowing what technology is used on current execution, allowing to connect a new technology on the same old codebase, without a rebuild, only adding configuration statements.

Complex Contexts

Complex Contexts Is helpful for implement crosscutting concerns like transactions, connections and whatever you want to pass to call stack in deeply without dirty your domain/model /business logic layer or whatever you called as your Business Code with parameters that explain specific technologies and building a non-agnostic domain. We have an extensible NHibernate infrastructure with FluentNHibernate to provide a simple way delivery a configurable, cross-platform, cross-database Unit of Work.