1. Overall

1.1 Introduction

1.1.1 Introduction

We are implementing common and similar structures: **Authorization, Validation, Exception Handling, Logging, Localization, Database Connection Management, Setting Management, Audit Logging** are some of these common structures. Also, we are building **architectural structures** and **best practices** like **Layered** and **Modular** Architecture, **Domain Driven Design**, **Dependency Injection** and so on. And trying to develop applications based on some **conventions**.

1.1.2 A Quick Sample

Let's investigate a simple class to see StudioX's Framework benefits:

```
[StudioXAuthorize(PermissionNames.System.Administration.Users.MainMenu)]
1 reference | Long Huynh, 24 days ago | 2 authors, 3 changes
public class UserAppService :BoilerplateAppServiceBase, IUserAppService
{
    private readonly IRepository<User, long> userRepository;

    0 references | 0 changes | 0 authors, 0 changes
    public UserAppService(IRepository<User, long> userRepository)
    {
        this.userRepository = userRepository;
    }

    [StudioXAuthorize(PermissionNames.System.Administration.Users.Edit)]
    1 reference | Long Huynh, 24 days ago | 1 author, 1 change
    public async Task Update(UpdateUserInput input)
    {
        Logger.Info("Updatating a user for input: " + input);

        var user = await userRepository.GetAsync(input.Id);
        if (user == null)
        {
            throw new UserFriendlyException(L("CouldNotFoundTheUserMessage"));
        }

        var mapped = input.MapTo(user);
        await userRepository.UpdateAsync(mapped);
        await UserManager.SetRoles(user, input.RoleNames);
    }
}
```

Here, we see a sample Application Service method. An application service, in DDD, is directly used by presentation layer to perform **use cases** of the application. We can think that **Update** method is called by javascript via AJAX. Let's see StudioX's some benefits here:

- **Dependency Injection**: StudioX uses and provides a strong and conventional DI infrastructure. Since this class is an application service, it's conventionally registered to DI container as transient (created per request). It can simply inject all dependencies (as IRepository<User, long> in this sample).
- **Repository**: StudioX can create a default repository for each entity (as IRepository<User, long> in this example). Default repository has many useful methods as **Get** used in this example. We can easily extend default repository upon our needs. Repositories abstracts DBMS, ORMs and simplifies data access logic.
- **Authorization**: StudioX can check permissions. It prevents access to **Update** method if current user has no "updating user" permission or not logged in. It simplifies authorization using declarative attributes but also has additional ways of authorization.
- **Validation**: StudioX automatically checks if input is null. It also validates all properties of an input based on standard data annotation attributes and custom validation rules. If request is not valid, it throws a proper validation exception.
- **Audit Logging**: User, browser, IP address, calling service, method, parameters, calling time, execution duration and some other informations are automatically saved for each request based on conventions and configurations.
- **Unit Of Work**: In StudioX Framework, each application service method is assumed as a unit of work as default. It automatically creates a connection and begins a transaction at the beggining of the method. If the method successfully completed without exception, then the transaction is commited and connection is disposed. Even this method uses different repositories or methods, all of them will be atomic (transactional). And all changes on entities are automatically saved when transaction is commited. Thus, we don't even need to call **repository.UpdateAsync(user)** method as shown here.
- **Exception Handling**: We almost never handle exceptions in StudioX in a web application. All exceptions are automatically handled by default. If an exception occurs, StudioX automatically logs it and returns a proper result to the client. For example, if this is an AJAX request, the it returns a JSON to client

indicates that an error occurred. If hides actual exception from client unless the exception is a UserFriendlyException as used in this sample. It also understands and handles errors on client side and show appropriate messages to users.

- **Logging**: As you see, we can write logs using the Logger object defined in base class. Log4Net is used as default but it's changeable or configurable.
- **Localization**: Notice that we used **L** method while throwing exception. Thus, it's automatically localized based on current user's culture. Surely, we're defining CouldNotFoundTheTaskMessage in somewhere (see localization document for more).
- **Auto Mapping**: In the last line, we're using StudioX's MapTo extension method to map input properties to entity properties. It uses AutoMapper library to perform mapping. Thus, we can easily map properties from one object to another based on naming conventions.
- **Dynamic Web API Layer**: TaskAppService is a simple class actually (even no need to deliver from ApplicationService). We generally write a wrapper Web API Controller to expose methods to javascript clients. StudioX automatically does that on runtime. Thus, we can use application service methods directly from clients.
- **Dynamic Javascript AJAX Proxy**: StudioX creates javascript proxy methods those make calling application service methods just as simple as calling javascript methods on the client.

We can see benefit of StudioX in such a simple class. All these tasks normally take significiant time, but all they are automatically handled by StudioX.

1.1.3 What Else

Beside this simple example, StudioX provides a strong infrastructure and application model. Here, some other features of StudioX:

- **Modularity**: Provides a strong infrastructure to build reusable modules.
- **Data Filters**: Provides automatic data filtering to implement some patterns like soft-delete and multi-tenancy.
- **Multi Tenancy**: It fully supports multi-tenancy, including single database or database per tenant architectures.

- **Setting Management**: Provides a strong infrastructure to get/change application, tenant and user level settings.
- **Unit & Integration Testing**: It's built testability in mind. Also provides base classes to simplify unit & integration tests. See this article for more information.

For all features, see documentation.