1.4 Startup Configuration

StudioX provides an infrastructure and a model to configure it and modules on startup.

1.4.1 Configuring StudioX

Configuring StudioX is made on **PreInitialize** event of your module. Example configuration:

```
public class SimpleTaskSystemModule : StudioXModule
{
    public override void PreInitialize()
    {
        //Add languages for your application
        Configuration.Localization.Languages.Add(
                                            new LanguageInfo("en", "English", "flag-en", true));
        Configuration.Localization.Languages.Add(new LanguageInfo("tr", "Türkçe", "flag-tr"));

        //Add a localization source
        Configuration.Localization.Sources.Add(
            new XmlLocalizationSource(
                "SimpleTaskSystem",
                HttpContext.Current.Server.MapPath("~/Localization/SimpleTaskSystem")
                )
            );

        //Configure navigation/menu
        Configuration.Navigation.Providers.Add<SimpleTaskSystemNavigationProvider>();
    }

    public override void Initialize()
    {
        IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());
    }
}
```

StudioX is designed modularity in mind. Different modules can configure StudioX. For example, different modules can add navigation provider to add their own menu items to the main menu. (See localization and navigation documents for details on configuring them).

Replacing Built-In Services

Configuration.ReplaceService method can be used to **override** a built-in service. For example, you can replace IStudioXSession service with your custom implementation as shown below:

```
Configuration.ReplaceService<IStudioXSession, MySession>(DependencyLifeStyle.Transient);
```

ReplaceService method has an overload to pass an **action** to make replacement in a custom way (you can directly use Castle Windsor for advanced registration API).

Same service can be replaced multiple times (especially, in different modules). Last replaced will be valid (As you know, module PreInitialize methods are executed by dependency order).

1.4.2 Configuring Modules

Beside framework's own startup configuration, a module can extend **IStudioXModuleConfigurations** interface to provide configuration points for the module. Example:

```
using StudioX.Web.Configuration;

public override void PreInitialize()
{
    Configuration.Modules.StudioXWebCommon().SendAllExceptionsToClients = true;
}
```

In this example, we configured StudioXWebCommon module to send all exceptions to clients.

Not every module should define this type of configuration. It's generally needed when a module will be re-usable in different applications and needs to be configured on startup.

1.4.3 Creating Configuration For a Module

Assume that we have a module named MyModule and it has some configuration properties. First, we create a class for these cofigurable properties:

```
public class MyModuleConfig
{
    public bool SampleConfig1 { get; set; }

    public string SampleConfig2 { get; set; }
}
```

Then we register this class to Dependency Injection on **PreInitialize** event of MyModule (Thus, it will be injectable):

```
IocManager.Register<MyModuleConfig>();
```

It should be registered as **Singleton** as in this sample. Now, we can use the following code to configure MyModule in our module's PreInitialize method:

```
Configuration.Get<MyModuleConfig>().SampleConfig1 = false;
```

While we can use IStudioXStartupConfiguration.Get method as shown below, we can create an extension method to IModuleConfigurations like that:

```
public static class MyModuleConfigurationExtensions
{
    public static MyModuleConfig MyModule(this IModuleConfigurations moduleConfigurations)
    {
        return moduleConfigurations.StudioXConfiguration.Get<MyModuleConfig>();
    }
}
```

Now, other modules can configure this module using the extension method:

```
Configuration.Modules.MyModule().SampleConfig1 = false;
Configuration.Modules.MyModule().SampleConfig2 = "test";
```

This makes easy to investigate module configurations and collect them in a single place (Configuration.Modules...). StudioX itself defines extension methods for it's own module configurations.

At some point, MyModule needs to this configuration. You can inject MyModuleConfig and use configured values. Example:

```
public class MyService : ITransientDependency
{
    private readonly MyModuleConfig configuration;
    public MyService(MyModuleConfig configuration)
    {
        this.configuration = configuration;
    }

    public void DoIt()
    {
        if (this.configuration.SampleConfig2 == "test"){ }
    }
}
```

Thus, modules can create central configuration points in StudioX system.