1.3 Module System

1.3.1 Introduction

StudioX provides an infrastructure to build modules and compose them to create an application. A module can depend on another module. Generally, an assembly is considered as a module. If you created an application with more than one assembly, it's suggested to create a module definition for each assembly.

Module system is currently focused on server side rather than client side.

1.3.2 Module Definition

A module is defined with a class that is derived from **StudioXModule**. Say that we're developing a Blog module that can be used in different applications. Simplest module definition can be as shown below:

```
public class MyBlogApplicationModule : StudioXModule
{
    public override void Initialize()
    {
        IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());
    }
}
```

Module definition class is responsible to register it's classes to dependency injection (can be done conventionally as shown above) if needed. It can configure application and other modules, add new features to the application and so on...

1.3.3 Lifecycle Methods

StudioX calls some specific methods of modules on application startup and shutdown. You can override these methods to perform some specific tasks.

StudioX calls these methods **ordered by dependecies**. If module A depends on module B, module B is initialized before module A. Exact order of startup methods: PreInitialize-B, PreInitialize-A, Initialize-B, Initialize-A, PostInitialize-B and PostInitialize-A. This is true for all dependency graph. **Shutdown** method is also similar but in **reverse order**.

- **PreInitialize**: This method is called first when application starts. It's usual method to configure the framework and other modules before they initialize. Also, you can write some specific code here to run before dependency injection registrations. For example, if you create a conventional registration class, you should register it here using IocManager.AddConventionalRegisterer method.
- **Initialize:** It's the usual place where dependency injection registration should be done. It's generally done using IocManager.RegisterAssemblyByConvention method. If you want to define custom dependency registration, see dependency injection documentation.
- **PostInitialize:** This method is called lastly in startup progress. It's safe to resolve a dependency here.
- **Shutdown:** This method is called when the application shuts down.

1.3.4 Module Dependencies

A module can be dependent to another. It's required to explicitly declare dependencies using DependsOn attribute like below:

```
[DependsOn(typeof(MyBlogCoreModule))]
public class MyBlogApplicationModule : StudioXModule
{
    public override void Initialize(){}
}
```

Thus, we declare to StudioX that MyBlogApplicationModule depends on MyBlogCoreModule and the MyBlogCoreModule should be initialized before the MyBlogApplicationModule. StudioX can resolve dependencies recursively beginning from the startup module and initialize them accordingly. Startup module initialized as the last module.

1.3.5 PlugIn Modules

While modules are investigated beginning from startup module and going to dependencies, StudioX can also load modules **dynamically**. **StudioXBootstrapper** class defines **PlugInSources** property which can be used to add sources to dynamically load plugin modules. A plugin source can be any class

implements **IPlugInSource** interface. **PlugInFolderSource** class implements it to get plugin modules from assemblies located in a folder.

ASP.NET Core: StudioX ASP.NET Core module defines options in **AddStudioX** extension method to add plugin sources in **Startup** class:

```
services.AddStudioX<MyStartupModule>(options =>
{
    options.PlugInSources.Add(new FolderPlugInSource(@"C:\MyPlugIns"));
});
```

We could use **AddFolder** extension method for a simpler syntax:

```
services.AddStudioX<MyStartupModule>(options => {
    options.PlugInSources.AddFolder(@"C:\MyPlugIns");
});
```

ASP.NET MVC, Web API: For classic ASP.NET MVC applications, we can add plugin folders by overriding **Application_Start** in **global.asax** as shown below:

```
public class MvcApplication : StudioXWebApplication<MyStartupModule>
{
    protected override void Application_Start(object sender, EventArgs e)
    {
        StudioXBootstrapper.PlugInSources.AddFolder(@"C:\MyPlugIns");
        //...
        base.Application_Start(sender, e);
    }
}
```

Controllers in PlugIns: If your modules include MVC or Web API Controllers, ASP.NET can not investigate your controllers. To overcome this issue, you can change global.asax file like below:

```
[assembly: PreApplicationStartMethod(typeof(PreStarter), "Start")]
namespace MyDemoApp.Web
{
    public class MvcApplication : StudioXWebApplication<MyStartupModule>{ }

    public static class PreStarter
    {
        public static void Start()
        {
            //...
            MvcApplication.StudioXBootstrapper.PlugInSources.AddFolder(@"C:\MyPlugIns\");
            MvcApplication.StudioXBootstrapper.PlugInSources.AddToBuildManager();
```

```
        }
    }
}
```

1.3.6 Additional Assemblies

Default implementations for IAssemblyFinder and ITypeFinder (which is used by StudioX to investigate specific classes in the application) only finds module assemblies and types in those assemblies. We can override **GetAdditionalAssemblies** method in our module to include additional assemblies.

1.3.7 Custom Module Methods

Your modules also can have custom methods those can be used by other modules depend on this module. Assume that MyModule2 depends on MyModule1 and wants to call a method of MyModule1 in PreInitialize.

```
public class MyModule1 : StudioXModule
{
    public override void Initialize()
    {
        IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());
    }

    public void MyModuleMethod1()
    {
        //this is a custom method of this module
    }
}

[DependsOn(typeof(MyModule1))]
public class MyModule2 : StudioXModule
{
    private readonly MyModule1 myModule1;

    public MyModule2(MyModule1 myModule1)
    {
        this.myModule1 = myModule1;
    }

    public override void PreInitialize()
    {
        //Call MyModule1's method
        this.myModule1.MyModuleMethod1();
    }

    public override void Initialize()
    {
```

```
        IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());
    }
}
```

Here, we constructor-injected MyModule1 to MyModule2, so MyModule2 can call MyModule1's custom method. This is only possible if Module2 depends on Module1.

1.3.8 Module Configuration

While custom module methods can be used to configure modules, we suggest to use startup configuration system to define and set configuration for modules.

1.3.9 Module Lifetime

Module classes are automatically registered as singleton.