

LinksPlatform's Platform.Scopes Class Library

1.1 ./Scope.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Reflection;
4 using System.Linq;
5 using Platform.Interfaces;
6 using Platform.Exceptions;
7 using Platform.Disposables;
8 using Platform.Collections.Lists;
9 using Platform.Reflection;
10 using Platform.Singletons;
11 using System.Runtime.CompilerServices;
12
13 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15 namespace Platform.Scopes
16 {
17     public class Scope : DisposableBase
18     {
19         public static readonly Scope Global = new Scope(autoInclude: true, autoExplore: true);
20
21         private readonly bool _autoInclude;
22         private readonly bool _autoExplore;
23         private readonly Stack<object> _dependencies = new Stack<object>();
24         private readonly HashSet<object> _excludes = new HashSet<object>();
25         private readonly HashSet<object> _includes = new HashSet<object>();
26         private readonly HashSet<object> _blocked = new HashSet<object>();
27         private readonly Dictionary<Type, object> _resolutions = new Dictionary<Type, object>();
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public Scope(bool autoInclude, bool autoExplore)
31         {
32             _autoInclude = autoInclude;
33             _autoExplore = autoExplore;
34         }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public Scope(bool autoInclude) : this(autoInclude, false) { }
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public Scope() { }
41
42         #region Exclude
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public void ExcludeAssemblyOf<T>() => ExcludeAssemblyOf<Type>(typeof(T));
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         public void ExcludeAssemblyOf<Type>(Type type) => ExcludeAssembly(type.GetAssembly());
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public void ExcludeAssembly(Assembly assembly) =>
52             ↪ assembly.GetCachedLoadableTypes().ForEach(Exclude);
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         public void Exclude<T>() => Exclude(typeof(T));
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         public void Exclude(object @object) => _excludes.Add(@object);
59
60         #endregion
61
62         #region Include
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         public void IncludeAssemblyOf<T>() => IncludeAssemblyOf<Type>(typeof(T));
66
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         public void IncludeAssemblyOf<Type>(Type type) => IncludeAssembly(type.GetAssembly());
69
70         [MethodImpl(MethodImplOptions.AggressiveInlining)]
71         public void IncludeAssembly(Assembly assembly) =>
72             ↪ assembly.GetExportedTypes().ForEach(Include);
73
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         public void Include<T>()
76         {
77             var types = Types<T>.Array;
78             if (types.Length > 0)
```

```

78         types.ForEach(Include);
79     }
80     else
81     {
82         Include(typeof(T));
83     }
84 }
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 public void Include(object @object)
88 {
89     if (@object == null)
90     {
91         return;
92     }
93     if (_includes.Add(@object))
94     {
95         var type = @object as Type;
96         if (type != null)
97         {
98             type.GetInterfaces().ForEach(Include);
99             Include(type.GetBaseType());
100         }
101     }
102 }
103
104 #endregion
105
106 #region Use
107
108 /// <remarks>
109 /// TODO: Use Default[T].Instance if the only constructor object has is parameterless.
110 /// TODO: Think of interface chaining IDoubletLinks[T] (default) -> IDoubletLinks[T]
111 ///      (checker) -> IDoubletLinks[T] (synchronizer) (may be UseChain[IDoubletLinks[T],
112 ///      Types[DefaultLinks, DefaultLinksDependencyChecker, DefaultSynchronizedLinks]]
113 /// TODO: Add support for factories
114 /// </remarks>
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 public T Use<T>()
117 {
118     if (_excludes.Contains(typeof(T)))
119     {
120         throw new InvalidOperationException($"Type {typeof(T).Name} is excluded and
121         ↪ cannot be used.");
122     }
123     if (_autoInclude)
124     {
125         Include<T>();
126     }
127     if (!TryResolve(out T resolved))
128     {
129         throw new InvalidOperationException($"Dependency of type {typeof(T).Name}
130         ↪ cannot be resolved.");
131     }
132     if (!_autoInclude)
133     {
134         Include<T>();
135     }
136     Use(resolved);
137     return resolved;
138 }
139
140 [MethodImpl(MethodImplOptions.AggressiveInlining)]
141 public T UseSingleton<T>(IFactory<T> factory) => UseAndReturn(Singleton.Get(factory));
142
143 [MethodImpl(MethodImplOptions.AggressiveInlining)]
144 public T UseSingleton<T>(Func<T> creator) => UseAndReturn(Singleton.Get(creator));
145
146 [MethodImpl(MethodImplOptions.AggressiveInlining)]
147 public T UseAndReturn<T>(T @object)
148 {
149     Use(@object);
150     return @object;
151 }
152
153 [MethodImpl(MethodImplOptions.AggressiveInlining)]
154 public void Use(object @object)
155 {

```

```

152     Include(@object);
153     _dependencies.Push(@object);
154 }
155
156 #endregion
157
158 #region Resolve
159
160 [MethodImpl(MethodImplOptions.AggressiveInlining)]
161 public bool TryResolve<T>(out T resolved)
162 {
163     resolved = default;
164     var result = TryResolve(typeof(T), out object resolvedObject);
165     if (result)
166     {
167         resolved = (T)resolvedObject;
168     }
169     return result;
170 }
171
172 [MethodImpl(MethodImplOptions.AggressiveInlining)]
173 public bool TryResolve(Type requiredType, out object resolved)
174 {
175     resolved = null;
176     if (!_blocked.Add(requiredType))
177     {
178         return false;
179     }
180     try
181     {
182         if (_excludes.Contains(requiredType))
183         {
184             return false;
185         }
186         if (_resolutions.TryGetValue(requiredType, out resolved))
187         {
188             return true;
189         }
190         if (_autoExplore)
191         {
192             IncludeAssemblyOfType(requiredType);
193         }
194         var resultInstances = new List<object>();
195         var resultConstructors = new List<ConstructorInfo>();
196         foreach (var include in _includes)
197         {
198             if (!_excludes.Contains(include))
199             {
200                 var type = include as Type;
201                 if (type != null)
202                 {
203                     if (requiredType.IsAssignableFrom(type))
204                     {
205                         resultConstructors.AddRange(GetValidConstructors(type));
206                     }
207                     else if (type.GetTypeInfo().IsGenericTypeDefinition &&
208                             ↪ requiredType.GetTypeInfo().IsGenericType &&
209                             ↪ type.GetInterfaces().Any(x => x.Name == requiredType.Name))
210                     {
211                         var genericType =
212                             ↪ type.MakeGenericType(requiredType.GenericTypeArguments);
213                         if (requiredType.IsAssignableFrom(genericType))
214                         {
215                             resultConstructors.AddRange(GetValidConstructors(genericType)
216                             ↪ );
217                         }
218                     }
219                 }
220                 else if (requiredType.IsInstanceOfType(include) ||
221                         ↪ requiredType.IsAssignableFrom(include.GetType()))
222                 {
223                     resultInstances.Add(include);
224                 }
225             }
226         }
227         if (resultInstances.Count == 0 && resultConstructors.Count == 0)
228         {
229             return false;
230         }
231     }
232     catch { }
233 }

```

```

225     }
226     else if (resultInstances.Count > 0)
227     {
228         resolved = resultInstances[0];
229     }
230     else
231     {
232         SortConstructors(resultConstructors);
233         if (!TryResolveInstance(resultConstructors, out resolved))
234         {
235             return false;
236         }
237     }
238     _resolutions.Add(requiredType, resolved);
239     return true;
240 }
241 finally
242 {
243     _blocked.Remove(requiredType);
244 }
245 }
246
247 [MethodImpl(MethodImplOptions.AggressiveInlining)]
248 protected virtual void SortConstructors(List<ConstructorInfo> resultConstructors) =>
    ↪ resultConstructors.Sort((x, y) =>
    ↪ -x.GetParameters().Length.CompareTo(y.GetParameters().Length));
249
250 [MethodImpl(MethodImplOptions.AggressiveInlining)]
251 protected virtual bool TryResolveInstance(List<ConstructorInfo> constructors, out object
    ↪ resolved)
252 {
253     for (var i = 0; i < constructors.Count; i++)
254     {
255         try
256         {
257             var resultConstructor = constructors[i];
258             if (TryResolveConstructorArguments(resultConstructor, out object[]
    ↪ arguments))
259             {
260                 resolved = resultConstructor.Invoke(arguments);
261                 return true;
262             }
263         }
264         catch (Exception exception)
265         {
266             exception.Ignore();
267         }
268     }
269     resolved = null;
270     return false;
271 }
272
273 [MethodImpl(MethodImplOptions.AggressiveInlining)]
274 private ConstructorInfo[] GetValidConstructors(Type type)
275 {
276     var constructors = type.GetConstructors();
277     if (!_autoExplore)
278     {
279         constructors = constructors.ToArray(x =>
280         {
281             var parameters = x.GetParameters();
282             for (var i = 0; i < parameters.Length; i++)
283             {
284                 if (!_includes.Contains(parameters[i].ParameterType))
285                 {
286                     return false;
287                 }
288             }
289             return true;
290         });
291     }
292     return constructors;
293 }
294
295 [MethodImpl(MethodImplOptions.AggressiveInlining)]
296 private bool TryResolveConstructorArguments(ConstructorInfo constructor, out object[]
    ↪ arguments)
297 {

```

```

298     var parameters = constructor.GetParameters();
299     arguments = new object[parameters.Length];
300     for (var i = 0; i < parameters.Length; i++)
301     {
302         if (!TryResolve(parameters[i].ParameterType, out object argument))
303         {
304             return false;
305         }
306         Use(argument);
307         arguments[i] = argument;
308     }
309     return true;
310 }
311
312 #endregion
313
314 [MethodImpl(MethodImplOptions.AggressiveInlining)]
315 protected override void Dispose(bool manual, bool wasDisposed)
316 {
317     if (!wasDisposed)
318     {
319         while (_dependencies.Count > 0)
320         {
321             _dependencies.Pop().DisposeIfPossible();
322         }
323     }
324 }
325 }
326 }

```

1.2 ./Scope[TInclude].cs

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Scopes
6 {
7     public class Scope<TInclude> : Scope
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public Scope() : this(false, false) { }
11
12        [MethodImpl(MethodImplOptions.AggressiveInlining)]
13        public Scope(bool autoInclude) : this(autoInclude, false) { }
14
15        [MethodImpl(MethodImplOptions.AggressiveInlining)]
16        public Scope(bool autoInclude, bool autoExplore) : base(autoInclude, autoExplore) =>
17            ↪ Include<TInclude>();
18    }
19 }

```

1.3 ./Use.cs

```

1 using System.Runtime.CompilerServices;
2 using Platform.Disposables;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Scopes
7 {
8     public static class Use<T>
9     {
10        public static T Single
11        {
12            [MethodImpl(MethodImplOptions.AggressiveInlining)]
13            get => Scope.Global.Use<T>();
14        }
15
16        public static Disposable<T> New
17        {
18            [MethodImpl(MethodImplOptions.AggressiveInlining)]
19            get
20            {
21                var scope = new Scope(autoInclude: true, autoExplore: true);
22                return new Disposable<T, Scope>(scope.Use<T>(), scope);
23            }
24        }
25    }
26 }

```

Index

./Scope.cs, 1

./Scope[TInclude].cs, 5

./Use.cs, 5