

# LinksPlatform's Platform.Scopes Class Library

./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
12 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14 namespace Platform.Scopes
15 {
16     public class Scope : DisposableBase
17     {
18         public static readonly Scope Global = new Scope(autoInclude: true, autoExplore: true);
19
20         private readonly bool _autoInclude;
21         private readonly bool _autoExplore;
22         private readonly Stack<object> _dependencies = new Stack<object>();
23         private readonly HashSet<object> _excludes = new HashSet<object>();
24         private readonly HashSet<object> _includes = new HashSet<object>();
25         private readonly HashSet<object> _blocked = new HashSet<object>();
26         private readonly Dictionary<Type, object> _resolutions = new Dictionary<Type, object>();
27
28         public Scope(bool autoInclude, bool autoExplore)
29         {
30             _autoInclude = autoInclude;
31             _autoExplore = autoExplore;
32         }
33
34         public Scope(bool autoInclude) : this(autoInclude, false) { }
35
36         public Scope() { }
37
38         #region Exclude
39
40         public void ExcludeAssemblyOf<T>() => ExcludeAssemblyOfType(typeof(T));
41
42         public void ExcludeAssemblyOfType(Type type) => ExcludeAssembly(type.GetAssembly());
43
44         public void ExcludeAssembly(Assembly assembly) =>
45             ↪ assembly.GetCachedLoadableTypes().ForEach(Exclude);
46
47         public void Exclude<T>() => Exclude(typeof(T));
48
49         public void Exclude(object @object) => _excludes.Add(@object);
50
51         #endregion
52
53         #region Include
54
55         public void IncludeAssemblyOf<T>() => IncludeAssemblyOfType(typeof(T));
56
57         public void IncludeAssemblyOfType(Type type) => IncludeAssembly(type.GetAssembly());
58
59         public void IncludeAssembly(Assembly assembly) =>
60             ↪ assembly.GetExportedTypes().ForEach(Include);
61
62         public void Include<T>()
63         {
64             var types = Types<T>.Array;
65             if (types.Length > 0)
66             {
67                 types.ForEach(Include);
68             }
69             else
70             {
71                 Include(typeof(T));
72             }
73         }
74
75         public void Include(object @object)
76         {
77             if (@object == null)
78             {
79                 return;
80             }
81         }
82     }
83 }
```

```

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

```

```

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

```

```

226     protected virtual void SortConstructors(List<ConstructorInfo> resultConstructors) =>
227     ↪     resultConstructors.Sort((x, y) =>
228     ↪     -x.GetParameters().Length.CompareTo(y.GetParameters().Length));
229
230     protected virtual bool TryResolveInstance(List<ConstructorInfo> constructors, out object
231     ↪     resolved)
232     {
233         for (var i = 0; i < constructors.Count; i++)
234         {
235             try
236             {
237                 var resultConstructor = constructors[i];
238                 if (TryResolveConstructorArguments(resultConstructor, out object[]
239                 ↪                 arguments))
240                 {
241                     resolved = resultConstructor.Invoke(arguments);
242                     return true;
243                 }
244             }
245             catch (Exception exception)
246             {
247                 exception.Ignore();
248             }
249         }
250         resolved = null;
251         return false;
252     }
253
254     private ConstructorInfo[] GetValidConstructors(Type type)
255     {
256         var constructors = type.GetConstructors();
257         if (!_autoExplore)
258         {
259             constructors = constructors.ToArray(x =>
260             {
261                 var parameters = x.GetParameters();
262                 for (var i = 0; i < parameters.Length; i++)
263                 {
264                     if (!_includes.Contains(parameters[i].ParameterType))
265                     {
266                         return false;
267                     }
268                 }
269                 return true;
270             });
271         }
272         return constructors;
273     }
274
275     private bool TryResolveConstructorArguments(ConstructorInfo constructor, out object[]
276     ↪     arguments)
277     {
278         var parameters = constructor.GetParameters();
279         arguments = new object[parameters.Length];
280         for (var i = 0; i < parameters.Length; i++)
281         {
282             if (!TryResolve(parameters[i].ParameterType, out object argument))
283             {
284                 return false;
285             }
286             Use(argument);
287             arguments[i] = argument;
288         }
289         return true;
290     }
291
292     #endregion
293
294     protected override void Dispose(bool manual, bool wasDisposed)
295     {
296         if (!wasDisposed)
297         {
298             while (_dependencies.Count > 0)
299             {
300                 _dependencies.Pop().DisposeIfPossible();
301             }
302         }
303     }

```

```
299     }
300 }
```

#### ./Scope[TInclude].cs

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Scopes
4  {
5      public class Scope<TInclude> : Scope
6      {
7          public Scope() : this(false, false) { }
8          public Scope(bool autoInclude) : this(autoInclude, false) { }
9          public Scope(bool autoInclude, bool autoExplore) : base(autoInclude, autoExplore) =>
10             ↪ Include<TInclude>();
11     }
12 }
```

#### ./Use.cs

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

## Index

./Scope.cs, 1

./Scope[TInclude].cs, 5

./Use.cs, 5