

# LinksPlatform's Platform.Scopes Class Library

## 1.1 ./csharp/Platform.Scopes/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     /// <summary>
18     /// <para>
19     /// Represents the scope.
20     /// </para>
21     /// <para></para>
22     /// </summary>
23     /// <seealso cref="DisposableBase"/>
24     public class Scope : DisposableBase
25     {
26         /// <summary>
27         /// <para>
28         /// The auto explore.
29         /// </para>
30         /// <para></para>
31         /// </summary>
32         public static readonly Scope Global = new Scope(autoInclude: true, autoExplore: true);
33         private readonly bool _autoInclude;
34         private readonly bool _autoExplore;
35         private readonly Stack<object> _dependencies = new Stack<object>();
36         private readonly HashSet<object> _excludes = new HashSet<object>();
37         private readonly HashSet<object> _includes = new HashSet<object>();
38         private readonly HashSet<object> _blocked = new HashSet<object>();
39         private readonly Dictionary<Type, object> _resolutions = new Dictionary<Type, object>();
40
41         /// <summary>
42         /// <para>
43         /// Initializes a new <see cref="Scope"/> instance.
44         /// </para>
45         /// <para></para>
46         /// </summary>
47         /// <param name="autoInclude">
48         /// <para>A auto include.</para>
49         /// <para></para>
50         /// </param>
51         /// <param name="autoExplore">
52         /// <para>A auto explore.</para>
53         /// <para></para>
54         /// </param>
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public Scope(bool autoInclude, bool autoExplore)
57         {
58             _autoInclude = autoInclude;
59             _autoExplore = autoExplore;
60         }
61
62         /// <summary>
63         /// <para>
64         /// Initializes a new <see cref="Scope"/> instance.
65         /// </para>
66         /// <para></para>
67         /// </summary>
68         /// <param name="autoInclude">
69         /// <para>A auto include.</para>
70         /// <para></para>
71         /// </param>
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         public Scope(bool autoInclude) : this(autoInclude, false) { }
74
75         /// <summary>
76         /// <para>
77         /// Initializes a new <see cref="Scope"/> instance.
```

```

78     /// </para>
79     /// <para></para>
80     /// </summary>
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     public Scope() { }
83
84     #region Exclude
85
86     /// <summary>
87     /// <para>
88     /// Excludes the assembly of.
89     /// </para>
90     /// <para></para>
91     /// </summary>
92     /// <typeparam name="T">
93     /// <para>The .</para>
94     /// <para></para>
95     /// </typeparam>
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     public void ExcludeAssemblyOf<T>() => ExcludeAssemblyOf<Type>(typeof(T));
98
99     /// <summary>
100    /// <para>
101    /// Excludes the assembly of type using the specified type.
102    /// </para>
103    /// <para></para>
104    /// </summary>
105    /// <param name="type">
106    /// <para>The type.</para>
107    /// <para></para>
108    /// </param>
109    [MethodImpl(MethodImplOptions.AggressiveInlining)]
110    public void ExcludeAssemblyOf<Type>(Type type) => ExcludeAssembly(type.GetAssembly());
111
112    /// <summary>
113    /// <para>
114    /// Excludes the assembly using the specified assembly.
115    /// </para>
116    /// <para></para>
117    /// </summary>
118    /// <param name="assembly">
119    /// <para>The assembly.</para>
120    /// <para></para>
121    /// </param>
122    [MethodImpl(MethodImplOptions.AggressiveInlining)]
123    public void ExcludeAssembly(Assembly assembly) =>
124        ↪ assembly.GetCachedLoadableTypes().ForEach(Exclude);
125
126    /// <summary>
127    /// <para>
128    /// Excludes this instance.
129    /// </para>
130    /// <para></para>
131    /// </summary>
132    /// <typeparam name="T">
133    /// <para>The .</para>
134    /// <para></para>
135    /// </typeparam>
136    [MethodImpl(MethodImplOptions.AggressiveInlining)]
137    public void Exclude<T>() => Exclude(typeof(T));
138
139    /// <summary>
140    /// <para>
141    /// Excludes the object.
142    /// </para>
143    /// <para></para>
144    /// </summary>
145    /// <param name="@object">
146    /// <para>The object.</para>
147    /// <para></para>
148    /// </param>
149    [MethodImpl(MethodImplOptions.AggressiveInlining)]
150    public void Exclude(object @object) => _excludes.Add(@object);
151
152    #endregion
153
154    #region Include
155
156    /// <summary>

```

```

156     /// <para>
157     /// Includes the assembly of.
158     /// </para>
159     /// <para></para>
160     /// </summary>
161     /// <typeparam name="T">
162     /// <para>The .</para>
163     /// <para></para>
164     /// </typeparam>
165     [MethodImpl(MethodImplOptions.AggressiveInlining)]
166     public void IncludeAssemblyOf<T>() => IncludeAssemblyOfType(typeof(T));
167
168     /// <summary>
169     /// <para>
170     /// Includes the assembly of type using the specified type.
171     /// </para>
172     /// <para></para>
173     /// </summary>
174     /// <param name="type">
175     /// <para>The type.</para>
176     /// <para></para>
177     /// </param>
178     [MethodImpl(MethodImplOptions.AggressiveInlining)]
179     public void IncludeAssemblyOfType(Type type) => IncludeAssembly(type.GetAssembly());
180
181     /// <summary>
182     /// <para>
183     /// Includes the assembly using the specified assembly.
184     /// </para>
185     /// <para></para>
186     /// </summary>
187     /// <param name="assembly">
188     /// <para>The assembly.</para>
189     /// <para></para>
190     /// </param>
191     [MethodImpl(MethodImplOptions.AggressiveInlining)]
192     public void IncludeAssembly(Assembly assembly) =>
193         ↪ assembly.GetExportedTypes().ForEach(Include);
194
195     /// <summary>
196     /// <para>
197     /// Includes this instance.
198     /// </para>
199     /// <para></para>
200     /// </summary>
201     /// <typeparam name="T">
202     /// <para>The .</para>
203     /// <para></para>
204     /// </typeparam>
205     [MethodImpl(MethodImplOptions.AggressiveInlining)]
206     public void Include<T>()
207     {
208         var types = Types<T>.Array;
209         if (types.Length > 0)
210         {
211             types.ForEach(Include);
212         }
213         else
214         {
215             Include(typeof(T));
216         }
217     }
218
219     /// <summary>
220     /// <para>
221     /// Includes the object.
222     /// </para>
223     /// <para></para>
224     /// </summary>
225     /// <param name="@object">
226     /// <para>The object.</para>
227     /// <para></para>
228     /// </param>
229     [MethodImpl(MethodImplOptions.AggressiveInlining)]
230     public void Include(object @object)
231     {
232         if (@object == null)
233         {

```

```

233         return;
234     }
235     if (_includes.Add(@object))
236     {
237         var type = @object as Type;
238         if (type != null)
239         {
240             type.GetInterfaces().ForEach(Include);
241             Include(type.GetBaseType());
242         }
243     }
244 }
245
246 #endregion
247
248 #region Use
249
250 /// <remarks>
251 /// TODO: Use Default[T].Instance if the only constructor object has is parameterless.
252 /// TODO: Think of interface chaining IDoubletLinks[T] (default) -> IDoubletLinks[T]
253     ↪ (checker) -> IDoubletLinks[T] (synchronizer) (may be UseChain[IDoubletLinks[T],
254     ↪ Types[DefaultLinks, DefaultLinksDependencyChecker, DefaultSynchronizedLinks]]
255 /// TODO: Add support for factories
256 /// </remarks>
257 [MethodImpl(MethodImplOptions.AggressiveInlining)]
258 public T Use<T>()
259 {
260     if (_excludes.Contains(typeof(T)))
261     {
262         throw new InvalidOperationException($"Type {typeof(T).Name} is excluded and
263         ↪ cannot be used.");
264     }
265     if (_autoInclude)
266     {
267         Include<T>();
268     }
269     if (!TryResolve(out T resolved))
270     {
271         throw new InvalidOperationException($"Dependency of type {typeof(T).Name}
272         ↪ cannot be resolved.");
273     }
274     if (!_autoInclude)
275     {
276         Include<T>();
277     }
278     Use(resolved);
279     return resolved;
280 }
281
282 /// <summary>
283 /// <para>
284 /// Uses the singleton using the specified factory.
285 /// </para>
286 /// <para></para>
287 /// </summary>
288 /// <typeparam name="T">
289 /// <para>The .</para>
290 /// <para></para>
291 /// </typeparam>
292 /// <param name="factory">
293 /// <para>The factory.</para>
294 /// <para></para>
295 /// </param>
296 /// <returns>
297 /// <para>The</para>
298 /// <para></para>
299 /// </returns>
300 [MethodImpl(MethodImplOptions.AggressiveInlining)]
301 public T UseSingleton<T>(IFactory<T> factory) => UseAndReturn(Singleton.Get(factory));
302
303 /// <summary>
304 /// <para>
305 /// Uses the singleton using the specified creator.
306 /// </para>
307 /// <para></para>
308 /// </summary>
309 /// <typeparam name="T">
310 /// <para>The .</para>

```

```

307     /// <para></para>
308     /// </typeparam>
309     /// <param name="creator">
310     /// <para>The creator.</para>
311     /// <para></para>
312     /// </param>
313     /// <returns>
314     /// <para>The</para>
315     /// <para></para>
316     /// </returns>
317     [MethodImpl(MethodImplOptions.AggressiveInlining)]
318     public T UseSingleton<T>(Func<T> creator) => UseAndReturn(Singleton.Get(creator));
319
320     /// <summary>
321     /// <para>
322     /// Uses the and return using the specified object.
323     /// </para>
324     /// <para></para>
325     /// </summary>
326     /// <typeparam name="T">
327     /// <para>The .</para>
328     /// <para></para>
329     /// </typeparam>
330     /// <param name="@object">
331     /// <para>The object.</para>
332     /// <para></para>
333     /// </param>
334     /// <returns>
335     /// <para>The object.</para>
336     /// <para></para>
337     /// </returns>
338     [MethodImpl(MethodImplOptions.AggressiveInlining)]
339     public T UseAndReturn<T>(T @object)
340     {
341         Use(@object);
342         return @object;
343     }
344
345     /// <summary>
346     /// <para>
347     /// Uses the object.
348     /// </para>
349     /// <para></para>
350     /// </summary>
351     /// <param name="@object">
352     /// <para>The object.</para>
353     /// <para></para>
354     /// </param>
355     [MethodImpl(MethodImplOptions.AggressiveInlining)]
356     public void Use(object @object)
357     {
358         Include(@object);
359         _dependencies.Push(@object);
360     }
361
362     #endregion
363
364     #region Resolve
365
366     /// <summary>
367     /// <para>
368     /// Determines whether this instance try resolve.
369     /// </para>
370     /// <para></para>
371     /// </summary>
372     /// <typeparam name="T">
373     /// <para>The .</para>
374     /// <para></para>
375     /// </typeparam>
376     /// <param name="resolved">
377     /// <para>The resolved.</para>
378     /// <para></para>
379     /// </param>
380     /// <returns>
381     /// <para>The result.</para>
382     /// <para></para>
383     /// </returns>
384     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

385 public bool TryResolve<T>(out T resolved)
386 {
387     resolved = default;
388     var result = TryResolve(typeof(T), out object resolvedObject);
389     if (result)
390     {
391         resolved = (T)resolvedObject;
392     }
393     return result;
394 }
395
396 /// <summary>
397 /// <para>
398 /// Determines whether this instance try resolve.
399 /// </para>
400 /// <para></para>
401 /// </summary>
402 /// <param name="requiredType">
403 /// <para>The required type.</para>
404 /// <para></para>
405 /// </param>
406 /// <param name="resolved">
407 /// <para>The resolved.</para>
408 /// <para></para>
409 /// </param>
410 /// <returns>
411 /// <para>The bool</para>
412 /// <para></para>
413 /// </returns>
414 [MethodImpl(MethodImplOptions.AggressiveInlining)]
415 public bool TryResolve(Type requiredType, out object resolved)
416 {
417     resolved = null;
418     if (!_blocked.Add(requiredType))
419     {
420         return false;
421     }
422     try
423     {
424         if (_excludes.Contains(requiredType))
425         {
426             return false;
427         }
428         if (_resolutions.TryGetValue(requiredType, out resolved))
429         {
430             return true;
431         }
432         if (_autoExplore)
433         {
434             IncludeAssemblyOfType(requiredType);
435         }
436         var resultInstances = new List<object>();
437         var resultConstructors = new List<ConstructorInfo>();
438         foreach (var include in _includes)
439         {
440             if (!_excludes.Contains(include))
441             {
442                 var type = include as Type;
443                 if (type != null)
444                 {
445                     if (requiredType.IsAssignableFrom(type))
446                     {
447                         resultConstructors.AddRange(GetValidConstructors(type));
448                     }
449                     else if (type.GetTypeInfo().IsGenericTypeDefinition &&
450                             ↪ requiredType.GetTypeInfo().IsGenericType &&
451                             ↪ type.GetInterfaces().Any(x => x.Name == requiredType.Name))
452                     {
453                         var genericType =
454                             ↪ type.MakeGenericType(requiredType.GenericTypeArguments);
455                         if (requiredType.IsAssignableFrom(genericType))
456                         {
457                             resultConstructors.AddRange(GetValidConstructors(genericType)
458                                     ↪ );
459                         }
460                     }
461                 }
462             }
463         }
464     }
465 }

```

```

458         else if (requiredType.IsInstanceOfType(include) ||
459                 ↪ requiredType.IsAssignableFrom(include.GetType()))
460         {
461             resultInstances.Add(include);
462         }
463     }
464     if (resultInstances.Count == 0 && resultConstructors.Count == 0)
465     {
466         return false;
467     }
468     else if (resultInstances.Count > 0)
469     {
470         resolved = resultInstances[0];
471     }
472     else
473     {
474         SortConstructors(resultConstructors);
475         if (!TryResolveInstance(resultConstructors, out resolved))
476         {
477             return false;
478         }
479     }
480     _resolutions.Add(requiredType, resolved);
481     return true;
482 }
483 finally
484 {
485     _blocked.Remove(requiredType);
486 }
487 }
488
489 /// <summary>
490 /// <para>
491 /// Sorts the constructors using the specified result constructors.
492 /// </para>
493 /// <para></para>
494 /// </summary>
495 /// <param name="resultConstructors">
496 /// <para>The result constructors.</para>
497 /// <para></para>
498 /// </param>
499 [MethodImpl(MethodImplOptions.AggressiveInlining)]
500 protected virtual void SortConstructors(List<ConstructorInfo> resultConstructors) =>
501     ↪ resultConstructors.Sort((x, y) =>
502     ↪ -x.GetParameters().Length.CompareTo(y.GetParameters().Length));
503
504 /// <summary>
505 /// <para>
506 /// Determines whether this instance try resolve instance.
507 /// </para>
508 /// <para></para>
509 /// </summary>
510 /// <param name="constructors">
511 /// <para>The constructors.</para>
512 /// <para></para>
513 /// </param>
514 /// <param name="resolved">
515 /// <para>The resolved.</para>
516 /// <para></para>
517 /// </param>
518 /// <returns>
519 /// <para>The bool</para>
520 /// <para></para>
521 /// </returns>
522 [MethodImpl(MethodImplOptions.AggressiveInlining)]
523 protected virtual bool TryResolveInstance(List<ConstructorInfo> constructors, out object
524     ↪ resolved)
525 {
526     for (var i = 0; i < constructors.Count; i++)
527     {
528         try
529         {
530             var resultConstructor = constructors[i];
531             if (TryResolveConstructorArguments(resultConstructor, out object[]
532                 ↪ arguments))
533             {
534                 resolved = resultConstructor.Invoke(arguments);
535             }
536         }
537         catch { }
538     }
539     return resolved != null;
540 }

```

```

531         return true;
532     }
533 }
534 catch (Exception exception)
535 {
536     exception.Ignore();
537 }
538 }
539 resolved = null;
540 return false;
541 }
542 [MethodImpl(MethodImplOptions.AggressiveInlining)]
543 private ConstructorInfo[] GetValidConstructors(Type type)
544 {
545     var constructors = type.GetConstructors();
546     if (!_autoExplore)
547     {
548         constructors = constructors.ToArray(x =>
549         {
550             var parameters = x.GetParameters();
551             for (var i = 0; i < parameters.Length; i++)
552             {
553                 if (!_includes.Contains(parameters[i].ParameterType))
554                 {
555                     return false;
556                 }
557             }
558             return true;
559         });
560     }
561     return constructors;
562 }
563 [MethodImpl(MethodImplOptions.AggressiveInlining)]
564 private bool TryResolveConstructorArguments(ConstructorInfo constructor, out object[]
    ↪ arguments)
565 {
566     var parameters = constructor.GetParameters();
567     arguments = new object[parameters.Length];
568     for (var i = 0; i < parameters.Length; i++)
569     {
570         if (!TryResolve(parameters[i].ParameterType, out object argument))
571         {
572             return false;
573         }
574         Use(argument);
575         arguments[i] = argument;
576     }
577     return true;
578 }
579
580 #endregion
581
582 /// <summary>
583 /// <para>
584 /// Disposes the manual.
585 /// </para>
586 /// <para></para>
587 /// </summary>
588 /// <param name="manual">
589 /// <para>The manual.</para>
590 /// <para></para>
591 /// </param>
592 /// <param name="wasDisposed">
593 /// <para>The was disposed.</para>
594 /// <para></para>
595 /// </param>
596 [MethodImpl(MethodImplOptions.AggressiveInlining)]
597 protected override void Dispose(bool manual, bool wasDisposed)
598 {
599     if (!wasDisposed)
600     {
601         while (_dependencies.Count > 0)
602         {
603             _dependencies.Pop().DisposeIfPossible();
604         }
605     }
606 }
607 }

```



## 1.2 ./csharp/Platform.Scopes/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      /// <summary>
8      /// <para>
9      /// Represents the scope.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="Scope"/>
14     public class Scope<TInclude> : Scope
15     {
16         /// <summary>
17         /// <para>
18         /// Initializes a new <see cref="Scope"/> instance.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public Scope() : this(false, false) { }
24
25         /// <summary>
26         /// <para>
27         /// Initializes a new <see cref="Scope"/> instance.
28         /// </para>
29         /// <para></para>
30         /// </summary>
31         /// <param name="autoInclude">
32         /// <para>A auto include.</para>
33         /// <para></para>
34         /// </param>
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public Scope(bool autoInclude) : this(autoInclude, false) { }
37
38         /// <summary>
39         /// <para>
40         /// Initializes a new <see cref="Scope"/> instance.
41         /// </para>
42         /// <para></para>
43         /// </summary>
44         /// <param name="autoInclude">
45         /// <para>A auto include.</para>
46         /// <para></para>
47         /// </param>
48         /// <param name="autoExplore">
49         /// <para>A auto explore.</para>
50         /// <para></para>
51         /// </param>
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         public Scope(bool autoInclude, bool autoExplore) : base(autoInclude, autoExplore) =>
54             ↪ Include<TInclude>();
55     }
56 }
```

## 1.3 ./csharp/Platform.Scopes/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      /// <summary>
9      /// <para>
10     /// Represents the use.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     public static class Use<T>
15     {
16         /// <summary>
17         /// <para>
```

```

18     /// Gets the single value.
19     /// </para>
20     /// <para></para>
21     /// </summary>
22     public static T Single
23     {
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         get => Scope.Global.Use<T>();
26     }
27
28     /// <summary>
29     /// <para>
30     /// Gets the new value.
31     /// </para>
32     /// <para></para>
33     /// </summary>
34     public static Disposable<T> New
35     {
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         get
38         {
39             var scope = new Scope(autoInclude: true, autoExplore: true);
40             return new Disposable<T, Scope>(scope.Use<T>(), scope);
41         }
42     }
43 }
44 }

```

#### 1.4 ./csharp/Platform.Scopes.Tests/ScopeTests.cs

```

1  using Xunit;
2  using Platform.Reflection;
3
4  namespace Platform.Scopes.Tests
5  {
6      /// <summary>
7      /// <para>
8      /// Represents the scope tests.
9      /// </para>
10     /// <para></para>
11     /// </summary>
12     public class ScopeTests
13     {
14         /// <summary>
15         /// <para>
16         /// Defines the interface.
17         /// </para>
18         /// <para></para>
19         /// </summary>
20         public interface IInterface
21         {
22         }
23
24         /// <summary>
25         /// <para>
26         /// Represents the .
27         /// </para>
28         /// <para></para>
29         /// </summary>
30         /// <seealso cref="IInterface"/>
31         public class Class : IInterface
32         {
33         }
34
35         /// <summary>
36         /// <para>
37         /// Tests that single dependency test.
38         /// </para>
39         /// <para></para>
40         /// </summary>
41         [Fact]
42         public static void SingleDependencyTest()
43         {
44             using var scope = new Scope();
45             scope.IncludeAssemblyOf<IInterface>();
46             var instance = scope.Use<IInterface>();
47             Assert.IsType<Class>(instance);
48         }
49
50         /// <summary>

```

```
51     /// <para>
52     /// Tests that type parameters test.
53     /// </para>
54     /// <para></para>
55     /// </summary>
56     [Fact]
57     public static void TypeParametersTest()
58     {
59         using var scope = new Scope<Types<Class>>>();
60         var instance = scope.Use<IInterface>();
61         Assert.IsType<Class>(instance);
62     }
63 }
64 }
```

## Index

- ./csharp/Platform.Scopes.Tests/ScopeTests.cs, 10
- ./csharp/Platform.Scopes/Scope.cs, 1
- ./csharp/Platform.Scopes/Scope[TInclude].cs, 9
- ./csharp/Platform.Scopes/Use.cs, 9