

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
34         private readonly bool _autoInclude;
35         private readonly bool _autoExplore;
36         private readonly Stack<object> _dependencies = new Stack<object>();
37         private readonly HashSet<object> _excludes = new HashSet<object>();
38         private readonly HashSet<object> _includes = new HashSet<object>();
39         private readonly HashSet<object> _blocked = new HashSet<object>();
40         private readonly Dictionary<Type, object> _resolutions = new Dictionary<Type, object>();
41
42         /// <summary>
43         /// <para>
44         /// Initializes a new <see cref="Scope"/> instance.
45         /// </para>
46         /// <para></para>
47         /// </summary>
48         /// <param name="autoInclude">
49         /// <para>A auto include.</para>
50         /// <para></para>
51         /// </param>
52         /// <param name="autoExplore">
53         /// <para>A auto explore.</para>
54         /// <para></para>
55         /// </param>
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         public Scope(bool autoInclude, bool autoExplore)
58         {
59             _autoInclude = autoInclude;
60             _autoExplore = autoExplore;
61         }
62
63         /// <summary>
64         /// <para>
65         /// Initializes a new <see cref="Scope"/> instance.
66         /// </para>
67         /// <para></para>
68         /// </summary>
69         /// <param name="autoInclude">
70         /// <para>A auto include.</para>
71         /// <para></para>
72         /// </param>
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         public Scope(bool autoInclude) : this(autoInclude, false) { }
75
76         /// <summary>
77         /// <para>
```

```

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

```

```

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

```

```

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

```

```

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

```

```

385 [MethodImpl(MethodImplOptions.AggressiveInlining)]
386 public bool TryResolve<T>(out T resolved)
387 {
388     resolved = default;
389     var result = TryResolve(typeof(T), out object resolvedObject);
390     if (result)
391     {
392         resolved = (T)resolvedObject;
393     }
394     return result;
395 }
396
397 /// <summary>
398 /// <para>
399 /// Determines whether this instance try resolve.
400 /// </para>
401 /// <para></para>
402 /// </summary>
403 /// <param name="requiredType">
404 /// <para>The required type.</para>
405 /// <para></para>
406 /// </param>
407 /// <param name="resolved">
408 /// <para>The resolved.</para>
409 /// <para></para>
410 /// </param>
411 /// <returns>
412 /// <para>The bool</para>
413 /// <para></para>
414 /// </returns>
415 [MethodImpl(MethodImplOptions.AggressiveInlining)]
416 public bool TryResolve(Type requiredType, out object resolved)
417 {
418     resolved = null;
419     if (!_blocked.Add(requiredType))
420     {
421         return false;
422     }
423     try
424     {
425         if (_excludes.Contains(requiredType))
426         {
427             return false;
428         }
429         if (_resolutions.TryGetValue(requiredType, out resolved))
430         {
431             return true;
432         }
433         if (_autoExplore)
434         {
435             IncludeAssemblyOfType(requiredType);
436         }
437         var resultInstances = new List<object>();
438         var resultConstructors = new List<ConstructorInfo>();
439         foreach (var include in _includes)
440         {
441             if (!_excludes.Contains(include))
442             {
443                 var type = include as Type;
444                 if (type != null)
445                 {
446                     if (requiredType.IsAssignableFrom(type))
447                     {
448                         resultConstructors.AddRange(GetValidConstructors(type));
449                     }
450                     else if (type.GetTypeInfo().IsGenericTypeDefinition &&
451                             ↪ requiredType.GetTypeInfo().IsGenericType &&
452                             ↪ type.GetInterfaces().Any(x => x.Name == requiredType.Name))
453                     {
454                         var genericType =
455                             ↪ type.MakeGenericType(requiredType.GenericTypeArguments);
456                         if (requiredType.IsAssignableFrom(genericType))
457                         {
458                             resultConstructors.AddRange(GetValidConstructors(genericType)
459                             ↪ ));
460                         }
461                     }
462                 }
463             }
464         }
465     }
466 }

```

```

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

```

```

532         return true;
533     }
534 }
535 catch (Exception exception)
536 {
537     exception.Ignore();
538 }
539 }
540 resolved = null;
541 return false;
542 }
543
544 /// <summary>
545 /// <para>
546 /// Gets the valid constructors using the specified type.
547 /// </para>
548 /// <para></para>
549 /// </summary>
550 /// <param name="type">
551 /// <para>The type.</para>
552 /// <para></para>
553 /// </param>
554 /// <returns>
555 /// <para>The constructors.</para>
556 /// <para></para>
557 /// </returns>
558 [MethodImpl(MethodImplOptions.AggressiveInlining)]
559 private ConstructorInfo[] GetValidConstructors(Type type)
560 {
561     var constructors = type.GetConstructors();
562     if (!_autoExplore)
563     {
564         constructors = constructors.ToArray(x =>
565         {
566             var parameters = x.GetParameters();
567             for (var i = 0; i < parameters.Length; i++)
568             {
569                 if (!_includes.Contains(parameters[i].ParameterType))
570                 {
571                     return false;
572                 }
573             }
574             return true;
575         });
576     }
577     return constructors;
578 }
579
580 /// <summary>
581 /// <para>
582 /// Determines whether this instance try resolve constructor arguments.
583 /// </para>
584 /// <para></para>
585 /// </summary>
586 /// <param name="constructor">
587 /// <para>The constructor.</para>
588 /// <para></para>
589 /// </param>
590 /// <param name="arguments">
591 /// <para>The arguments.</para>
592 /// <para></para>
593 /// </param>
594 /// <returns>
595 /// <para>The bool</para>
596 /// <para></para>
597 /// </returns>
598 [MethodImpl(MethodImplOptions.AggressiveInlining)]
599 private bool TryResolveConstructorArguments(ConstructorInfo constructor, out object[]
    ↪ arguments)
600 {
601     var parameters = constructor.GetParameters();
602     arguments = new object[parameters.Length];
603     for (var i = 0; i < parameters.Length; i++)
604     {
605         if (!TryResolve(parameters[i].ParameterType, out object argument))
606         {
607             return false;
608         }
609     }

```



```

609         Use(argument);
610         arguments[i] = argument;
611     }
612     return true;
613 }
614
615 #endregion
616
617 /// <summary>
618 /// <para>
619 /// Disposes the manual.
620 /// </para>
621 /// <para></para>
622 /// </summary>
623 /// <param name="manual">
624 /// <para>The manual.</para>
625 /// <para></para>
626 /// </param>
627 /// <param name="wasDisposed">
628 /// <para>The was disposed.</para>
629 /// <para></para>
630 /// </param>
631 [MethodImpl(MethodImplOptions.AggressiveInlining)]
632 protected override void Dispose(bool manual, bool wasDisposed)
633 {
634     if (!wasDisposed)
635     {
636         while (_dependencies.Count > 0)
637         {
638             _dependencies.Pop().DisposeIfPossible();
639         }
640     }
641 }
642 }
643 }

```

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 }

```

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`, 10