Simple Injector Documentation

Release 4

Simple Injector Contributors

Sep 23, 2017

Contents

1 Quick Start **3**
- 1.1 Overview . . . 3
- 1.2 Getting started . . . 3
- 1.3 A Quick Example . . . 4
  - 1.3.1 Dependency Injection . . . 4
  - 1.3.2 Introducing Simple Injector . . . 5
- 1.4 More information . . . 6

2 Using Simple Injector **7**
- 2.1 Resolving instances . . . 9
- 2.2 Configuring Simple Injector . . . 9
  - 2.2.1 Automatic/Batch-registration . . . 11
- 2.3 Collections . . . 12
  - 2.3.1 Collection types . . . 14
  - 2.3.2 Batch-registering collections . . . 14
  - 2.3.3 Adding registrations to an existing collection . . . 15
- 2.4 Verifying the container's configuration . . . 15
- 2.5 Automatic constructor injection / auto-wiring . . . 15
- 2.6 More information . . . 16

3 Object Lifetime Management **17**
- 3.1 Transient . . . 18
- 3.2 Singleton . . . 18
- 3.3 Scoped . . . 19
  - 3.3.1 Order of disposal . . . 20
- 3.4 Thread Scoped . . . 20
- 3.5 Async Scoped (async/await) . . . 21
- 3.6 Web Request . . . 22
- 3.7 Web Async Scoped lifestyle vs. Web Request lifestyle . . . 22
- 3.8 WCF Operation . . . 23
- 3.9 Per Graph . . . 23
- 3.10 Instance Per Dependency . . . 24
- 3.11 Per Thread . . . 24
- 3.12 Per HTTP Session . . . 24
- 3.13 Hybrid . . . 24
- 3.14 Developing a Custom Lifestyle . . . 25

4 Integration Guide **27**
4.1 Console Application Integration Guide . . . 27
4.2 ASP.NET Core MVC Integration Guide . . . 29
4.2.1 Wiring custom middleware . . . 30
4.2.2 Cross-wiring ASP.NET and third party services . . . 31
4.2.3 Working with ASP.NET Core Identity . . . 31
4.2.4 Working with *IOption<T>* . . . 33
4.3 ASP.NET MVC Integration Guide . . . 34
4.4 ASP.NET Web API Integration Guide . . . 34
4.4.1 Basic setup . . . 34
4.4.2 Extra features . . . 35
4.5 ASP.NET Web Forms Integration Guide . . . 38
4.6 OWIN Integration Guide . . . 40
4.6.1 Basic setup . . . 40
4.6.2 Extra features . . . 41
4.7 Windows Forms Integration Guide . . . 41
4.8 WCF Integration Guide . . . 42
4.8.1 WAS Hosting and Non-HTTP Activation . . . 43
4.9 Windows Presentation Foundation Integration Guide . . . 45
4.9.1 Step 1: . . . 45
4.9.2 Step 2: . . . 45
4.9.3 Step 3: . . . 46
4.9.4 Usage . . . 46
4.10 Silverlight Integration Guide . . . 47
4.10.1 Usage . . . 47
4.11 Other Technologies . . . 48
4.12 Patterns . . . 49

5 Diagnostic Services **51**
5.1 Supported Warnings . . . 51
5.1.1 Diagnostic Warning - Lifestyle Mismatches . . . 51
5.1.2 Diagnostic Warning - Short Circuited Dependencies . . . 54
5.1.3 Diagnostic Warning - Torn Lifestyle . . . 56
5.1.4 Diagnostic Warning - Ambiguous Lifestyles . . . 57
5.1.5 Diagnostic Warning - Disposable Transient Components . . . 59
5.2 Supported Information messages . . . 61
5.2.1 Diagnostic Warning - Potential Single Responsibility Violations . . . 61
5.2.2 Diagnostic Warning - Container-registered Types . . . 63
5.3 How to view diagnostic results . . . 65
5.4 Suppressing warnings . . . 67
5.5 Limitations . . . 68

6 How To **71**
6.1 Register factory delegates . . . 71
6.1.1 Lazy . . . 74
6.2 Resolve instances by key . . . 75
6.3 Register multiple interfaces with the same implementation . . . 77
6.4 Override existing registrations . . . 77
6.5 Verify the container's configuration . . . 78
6.6 Work with dependency injection in multi-threaded applications . . . 80
6.7 Package registrations . . . 82

7 Advanced Scenarios **85**
7.1 Generics . . . 85

7.2 Batch / Automatic registration . . . 86
7.3 Registration of open generic types . . . 88
7.4 Mixing collections of open-generic and non-generic components . . . 90
7.5 Unregistered type resolution . . . 91
7.6 Context based injection . . . 91
7.7 Property injection . . . 93
7.7.1 Implicit property injection . . . 93
7.7.2 Explicit property injection . . . 93
7.7.3 Enabling property injection . . . 93
7.7.4 IPropertySelectionBehavior . . . 94
7.8 Covariance and Contravariance . . . 94
7.9 Registering plugins dynamically . . . 96

8 Aspect-Oriented Programming 97
8.1 Decoration . . . 97
8.1.1 Applying Decorators conditionally . . . 99
8.1.2 Applying Decorators conditionally using type constraints . . . 99
8.1.3 Decorators with Func<T> decoratee factories . . . 100
8.1.4 Decorated collections . . . 103
8.1.5 Using contextual information inside decorators . . . 104
8.1.6 Applying decorators conditionally based on consumer . . . 105
8.1.7 Decorator registration factories . . . 106
8.2 Interception using Dynamic Proxies . . . 106

9 Extensibility Points 109
9.1 Overriding Constructor Resolution Behavior . . . 109
9.2 Overriding Property Injection Behavior . . . 111
9.3 Overriding Parameter Injection Behavior . . . 113
9.4 Resolving Unregistered Types . . . 113
9.5 Overriding Lifestyle Selection Behavior . . . 113
9.6 Intercepting the Creation of Types . . . 115
9.7 Building up External Instances . . . 116
9.8 Interception of Resolved Object Graphs . . . 116

10 Simple Injector Pipeline 119
10.1 Registration Pipeline . . . 119
10.2 Resolve Pipeline . . . 121

11 Design Principles 125
11.1 Make simple use cases easy, make complex use cases possible . . . 125
11.2 Push developers into best practices . . . 125
11.3 Fast by default . . . 126
11.4 Don't force vendor lock-in . . . 126
11.5 Never fail silently . . . 126
11.6 Features should be intuitive . . . 126
11.7 Communicate errors clearly and describe how to solve them . . . 127

12 Design Decisions 129
12.1 The container is locked after the first call to resolve . . . 129
12.2 The API clearly differentiates the registration of collections . . . 130
12.3 No support for XML based configuration . . . 132
12.4 Never force users to release what they resolve . . . 132
12.5 Don't allow resolving scoped instances outside an active scope . . . 133
12.6 No out-of-the-box support for property injection . . . 133
12.7 No out-of-the-box support for interception . . . 134

12.8 Limited batch-registration API . . . 134
12.9 No per-thread lifestyle . . . 135
12.10 Allow only a single constructor . . . 135

13 Legal stuff **137**
13.1 Simple Injector License . . . 137
13.2 Contributions . . . 137
13.3 Simple Injector Trademark Policy . . . 137

14 How to Contribute **139**

15 Appendix **141**
15.1 Interception Extensions . . . 141
15.2 Variance Extensions . . . 145
15.3 Resolution conflicts caused by dynamic assembly loading . . . 147
15.3.1 What's the problem? . . . 147
15.3.2 So what do you need to do? . . . 147
15.4 Compiler Error: 'InjectionConsumerInfo.ServiceType' is deprecated . . . 148
15.4.1 What's the problem? . . . 148
15.4.2 So what should I do instead? . . . 148

16 Indices and tables **151**

Simple Injector is an easy-to-use Dependency Injection (DI) library for .NET that supports .NET Core, Xamarin, Mono and Universal apps. Simple Injector is easily integrated with frameworks such as Web API, MVC, WCF, ASP.NET Core and many others. It's easy to implement the dependency injection pattern with loosely coupled components using Simple Injector.

Use:

- Get official builds from NuGet[1] or run the following command in the Package Manager Console: *PM> Install-Package SimpleInjector*
- Download the binaries[2].
- Browse the source code[3].
- Browse questions on Stackoverflow[4] and Github[5].

Engage:

- Discuss[6]
- Contribute[7]
- Twitter[8]

Contents:

[1] https://simpleinjector.org/nuget
[2] https://simpleinjector.org/download
[3] https://simpleinjector.org/source
[4] https://simpleinjector.org/stackoverflow
[5] https://simpleinjector.org/issues
[6] https://simpleinjector.org/forum
[7] https://simpleinjector.org/contribute
[8] https://simpleinjector.org/twitter

CHAPTER 1

Quick Start

Overview

The goal of Simple Injector is to provide .NET application developers with an easy, flexible, and fast Inversion of Control library[9] that promotes best practice to steer developers towards the pit of success.

Many of the existing DI libraries have a big complicated legacy API or are new, immature, and lack features often required by large scale development projects. Simple Injector fills this gap by supplying a simple implementation with a carefully selected and complete set of features. File and attribute based configuration methods have been abandoned (they invariably result in brittle and maintenance heavy applications), favoring simple code based configuration instead. This is enough for most applications, requiring only that the configuration be performed at the start of the program. The core library contains many features and allows almost any *advanced scenario*.

The following platforms are supported:

- *.NET 4.0* and up.
- *.NET Standard* including: * *Universal Windows Programs*. * *Mono*. * *.NET Core*. * *Xamarin*.

Simple Injector is carefully designed to run in **partial / medium trust**, and it is fast; blazingly fast.

Getting started

The easiest way to get started is by installing the available NuGet packages[10].

[9] https://martinfowler.com/articles/injection.html
[10] https://simpleinjector.org/nuget

A Quick Example

Dependency Injection

The general idea behind Simple Injector (or any DI library for that matter) is that you design your application around loosely coupled components using the Dependency Injection pattern[11] while adhering to the Dependency Inversion Principle[12]. Take for instance the following **CancelOrderHandler** class:

```
public class CancelOrderHandler {
    private readonly IOrderRepository repository;
    private readonly ILogger logger;
    private readonly IEventPublisher publisher;

    // Use constructor injection for the dependencies
    public CancelOrderHandler(
        IOrderRepository repository, ILogger logger, IEventPublisher publisher) {
        this.repository = repository;
        this.logger = logger;
        this.publisher = publisher;
    }

    public void Handle(CancelOrder command) {
        this.logger.Log("Cancelling order " + command.OrderId);
        var order = this.repository.GetById(command.OrderId);
        order.Status = OrderStatus.Cancelled;
        this.repository.Save(order);
        this.publisher.Publish(new OrderCancelled(command.OrderId));
    }
}

public class SqlOrderRepository : IOrderRepository {
    private readonly ILogger logger;

    // Use constructor injection for the dependencies
    public SqlOrderRepository(ILogger logger) {
        this.logger = logger;
    }

    public Order GetById(Guid id) {
        this.logger.Log("Getting Order " + order.Id);
        // Retrieve from db.
    }

    public void Save(Order order) {
        this.logger.Log("Saving order " + order.Id);
        // Save to db.
    }
}
```

The *CancelOrderHandler* class depends on the *IOrderRepository*, *ILogger* and *IEventPublisher* interfaces. By not depending on concrete implementations, we can test *CancelOrderHandler* in isolation. But ease of testing is only one of a number of things that Dependency Injection gives us. It also enables us, for example, to design highly flexible systems that can be completely composed in one specific location (often the startup path) of the application.

[11] https://en.wikipedia.org/wiki/Dependency_injection
[12] https://en.wikipedia.org/wiki/Dependency_inversion_principle

Introducing Simple Injector

Using Simple Injector, the configuration of the application using the *CancelOrderHandler* and *SqlOrderRepository* classes shown above, might look something like this:

```
using SimpleInjector;

static class Program
{
    static readonly Container container;

    static Program() {
        // 1. Create a new Simple Injector container
        container = new Container();

        // 2. Configure the container (register)
        container.Register<IOrderRepository, SqlOrderRepository>();
        container.Register<ILogger, FileLogger>(Lifestyle.Singleton);
        container.Register<CancelOrderHandler>();

        // 3. Verify your configuration
        container.Verify();
    }

    static void Main(string[] args)) {
        // 4. Use the container
        var handler = container.GetInstance<CancelOrderHandler>();

        var orderId = Guid.Parse(args[0]);
        var command = new CancelOrder { OrderId = orderId };

        handler.Handle(command);
    }
}
```

The given configuration registers implementations for the *ICancelOrderHandler*, *IOrderRepository* and *ILogger* interfaces. The code snippet shows a few interesting things. First of all, you can map concrete instances (such as *SqlOrderRepository*) to an interface or base type (such as *IOrderRepository*). In the given example, every time you ask the container for an *IOrderRepository*, it will always create a new *SqlOrderRepository* on your behalf (in DI terminology: an object with a **Transient** lifestyle).

The second registration maps the *ILogger* interface to a *FileLogger* implementation. This *FileLogger* is registered with the **Singleton** lifestyle; only one instance of *FileLogger* will ever be created by the **Container**.

Further more, you can map a concrete implementation to itself (as shown with the *CancelOrderHandler*). This registration is a short-hand for the following registration:

```
container.Register<CancelOrderHandler, CancelOrderHandler>();
```

This basically means, every time you request a *CancelOrderHandler*, you'll get a new *CancelOrderHandler*.

Using this configuration, when a *CancelOrderHandler* is requested, the following object graph is constructed:

```
new CancelOrderHandler(
    new SqlOrderRepository(
        logger),
    logger);
```

Note that object graphs can become very deep. What you can see is that not only *CancelOrderHandler* contains

dependencies, so does *SqlOrderRepository*. In this case *SqlOrderRepository* itself contains an *ILogger* dependency. Simple Injector will not only resolve the dependencies of *CancelOrderHandler* but will instead build a whole tree structure of any level deep for you.

And this is all it takes to start using Simple Injector. Design your classes around the SOLID principles and the Dependency Injection pattern (which is actually the hard part) and configure them during application initialization. Some frameworks (such as ASP.NET MVC) will do the rest for you, other frameworks (like ASP.NET Web Forms) will need a little bit more work. See the *Integration Guide* for examples of integrating with many common frameworks.

Please go to the *Using Simple Injector* section in the documentation to see more examples.

More information

For more information about Simple Injector please visit the following links:

- *Using Simple Injector* will guide you through the Simple Injector basics.
- The *Object Lifetime Management* page explains how to configure lifestyles such as *Transient*, *Singleton*, and many others.
- See the Reference library[13] for the complete API documentation.
- See the *Integration Guide* for more information about how to integrate Simple Injector into your specific application framework.
- For more information about dependency injection in general, please visit this page on Stackoverflow[14].
- If you have any questions about how to use Simple Injector or about dependency injection in general, the experts at Stackoverflow.com[15] are waiting for you.
- For all other Simple Injector related question and discussions, such as bug reports and feature requests, the Simple Injector discussion forum[16] will be the place to start.
- The book Dependency Injection in .NET[17] presents core DI patterns in plain C# so you'll fully understand how DI works.

Happy injecting!

[13] https://simpleinjector.org/ReferenceLibrary/
[14] https://stackoverflow.com/tags/dependency-injection/info
[15] https://stackoverflow.com/questions/ask?tags=simple-injector%20ioc-container%20dependency-injection%20.net%20c%23
[16] https://simpleinjector.org/forum
[17] https://manning.com/seemann/

CHAPTER 2

Using Simple Injector

This section will walk you through the basics of Simple Injector. After reading this section, you will have a good idea how to use Simple Injector.

Good practice is to minimize the dependency between your application and the DI library. This increases the testability and the flexibility of your application, results in cleaner code, and makes it easier to migrate to another DI library (if ever required). The technique for keeping this dependency to a minimum can be achieved by designing the types in your application around the constructor injection pattern: Define all dependencies of a class in the single public constructor of that type; do this for all service types that need to be resolved and resolve only the top most types in the application directly (i.e. let the container build up the complete graph of dependent objects for you).

Simple Injector's main type is the Container[18] class. An instance of *Container* is used to register mappings between each abstraction (service) and its corresponding implementation (component). Your application code should depend on abstractions and it is the role of the *Container* to supply the application with the right implementation. The easiest way to view the *Container* is as a big dictionary where the type of the abstraction is used as key, and each key's related value is the definition of how to create that particular implementation. Each time the application requests a service, a look up is made within the dictionary and the correct implementation is returned.

Tip: You should typically create a single *Container* instance for the whole application (one instance per app domain); *Container* instances are thread-safe.

Warning: Registering types in a *Container* instance should be done from one single thread. Requesting instances from the *Container* is thread-safe but registration of types is not.

Warning: Do not create an infinite number of *Container* instances (such as one instance per request). Doing so will drain the performance of your application. The library is optimized for using a very limited number of *Container* instances. Creating and initializing *Container* instances has a large overhead, but resolving from the *Container* is extremely fast once initialized.

Creating and configuring a *Container* is done by newing up an instance and calling the **Register** overloads to register each of your services:

```
var container = new SimpleInjector.Container();
```

[18] https://simpleinjector.org/ReferenceLibrary/?topic=html/T_SimpleInjector_Container.htm

```
// Registrations here
container.Register<ILogger, FileLogger>();
```

Ideally, the only place in an application that should directly reference and use Simple Injector is the startup path. For an ASP.NET Web Forms or MVC application this will usually be the **{"Application_OnStart"}** event in the Global.asax[19] page of the web application project. For a Windows Forms or console application this will be the *Main* method in the application assembly.

Tip: For more information about usage of Simple Injector for a specific technology, please see the *Integration Guide*.

The usage of Simple Injector consists of four to six steps:

1. Create a new container
2. Configure the container (*Register*)
3. [Optionally] verify the container
4. Store the container for use by the application
5. Retrieve instances from the container (*Resolve*)
6. [Optionally] *Dispose* the container instance when the application ends.

The first four steps are performed only once at application startup. The fifth step is usually performed multiple times (usually once per request) for the majority of applications. The first three steps are platform agnostic but the last three steps depend on a mix of personal preference and which presentation framework is being used. Below is an example for the configuration of an ASP.NET MVC application:

```
using System.Web.Mvc;
using SimpleInjector;
using SimpleInjector.Integration.Web.Mvc;

public class Global : System.Web.HttpApplication {

    protected void Application_Start(object sender, EventArgs e) {
        // 1. Create a new Simple Injector container
        var container = new Container();

        // 2. Configure the container (register)
        // See below for more configuration examples
        container.Register<IUserService, UserService>(Lifestyle.Transient);
        container.Register<IUserRepository, SqlUserRepository>(Lifestyle.Singleton);

        // 3. Optionally verify the container's configuration.
        container.Verify();

        // 4. Store the container for use by the application
        DependencyResolver.SetResolver(
            new SimpleInjectorDependencyResolver(container));
    }
}
```

In the case of MVC, the fifth step is the responsibility of the MVC framework. For each received web requests, the MVC framework will map that request to a *Controller* type and ask the application's *IDependencyResolver* to create an instance of that controller type. The registration of the **SimpleInjectorDependencyResolver** (part of the **SimpleInjector.Integration.Web.Mvc.dll**) will ensure that the request for creating an instance is forwarded on to Simple Injector. Simple Injector will create that controller with all of its nested dependencies.

[19] https://msdn.microsoft.com/en-us/library/1xaas8a2%28VS.71%29.aspx

The example below is a very basic MVC Controller:

```
using System;
using System.Web.Mvc;

public class UserController : Controller {
    private readonly IUserRepository repository;
    private readonly ILogger logger;

    public UserController(IUserRepository repository, ILogger logger) {
        this.repository = repository;
        this.logger = logger;
    }

    [HttpGet]
    public ActionResult Index(Guid id) {
        this.logger.Log("Index called.");
        User user = this.repository.GetById(id);
        return this.View(user);
    }
}
```

Resolving instances

Simple Injector supports two scenarios for retrieving component instances:

1. **Getting an object by a specified type**

```
var repository = container.GetInstance<IUserRepository>();

// Alternatively, you can use the weakly typed version
var repository = (IUserRepository)container.GetInstance(typeof(IUserRepository));
```

2. **Getting a collection of objects by their type**

```
IEnumerable<ICommand> commands = container.GetAllInstances<ICommand>();

// Alternatively, you can use the weakly typed version
IEnumerable<object> commands = container.GetAllInstances(typeof(ICommand));
```

Configuring Simple Injector

The *Container* class consists of several methods that enable registering instances for retrieval when requested by the application. These methods enable most common scenarios. Here are many of these common scenarios with a code example for each:

Configuring an automatically constructed single instance (Singleton) to always be returned:

The following example configures a single instance of type *RealUserService* to always be returned when an instance of *IUserService* is requested. The *RealUserService* will be constructed using *automatic constructor injection*.

```
// Configuration
container.Register<IUserService, RealUserService>(Lifestyle.Singleton);
```

```
// Usage
IUserService service = container.GetInstance<IUserService>();
```

Note: instances that are declared as *Singleton* should be thread-safe in a multi-threaded environment.

Configuring a single - manually created - instance (Singleton) to always be returned:

The following example configures a single instance of a manually created object *SqlUserRepository* to always be returned when a type of *IUserRepository* is requested.

```
// Configuration
container.RegisterSingleton<IUserRepository>(new SqlUserRepository());

// Usage
IUserRepository repository = container.GetInstance<IUserRepository>();
```

Note: Registering types using *automatic constructor injection* (auto-wiring) is the preferred method of registering types. Only new up instances manually when automatic constructor injection is not possible.

Configuring a single instance using a delegate:

This example configures a single instance as a delegate. The *Container* will ensure that the delegate is only called once.

```
// Configuration
container.Register<IUserRepository>(() => new SqlUserRepository("some constr"),
    Lifestyle.Singleton);

// Usage
IUserRepository repository = container.GetInstance<IUserRepository>();
```

Note: Registering types using *automatic constructor injection* (auto-wiring) is the recommended method of registering types. Only new up instances manually when automatic constructor injection is not possible.

Configuring an automatically constructed new instance to be returned:

By supplying the service type and the created implementation as generic types, the container can create new instances of the implementation (*MoveCustomerHandler* in this case) by *automatic constructor injection*.

```
// Configuration
container.Register<IHandler<MoveCustomerCommand>, MoveCustomerHandler>();

// Alternatively you can supply the transient Lifestyle with the same effect.
container.Register<IHandler<MoveCustomerCommand>, MoveCustomerHandler>(
    Lifestyle.Transient);

// Usage
var handler = container.GetInstance<IHandler<MoveCustomerCommand>>();
```

Configuring a new instance to be returned on each call using a delegate:

By supplying a delegate, types can be registered that cannot be created by using *automatic constructor injection*.

By referencing the *Container* instance within the delegate, the *Container* can still manage as much of the object creation work as possible:

```
// Configuration
container.Register<IHandler<MoveCustomerCommand>>(() => {
    // Get a new instance of the concrete MoveCustomerHandler class:
    var handler = container.GetInstance<MoveCustomerHandler>();
```

```
    // Configure the handler:
    handler.ExecuteAsynchronously = true;

    return handler;
});

container.Register<IHandler<MoveCustomerCommand>>(() => { ... }, Lifestyle.Transient);
// Alternatively you can supply the transient Lifestyle with the same effect.
// Usage
var handler = container.GetInstance<IHandler<MoveCustomerCommand>>();
```

Initializing auto-wired instances:

For types that need to be injected we recommend that you define a single public constructor that contains all dependencies. In scenarios where its impossible to fully configure a type using constructor injection, the *RegisterInitializer* method can be used to add additional initialization for such type. The previous example showed an example of property injection but a more preferred approach is to use the **RegisterInitializer** method:

```
// Configuration
container.Register<IHandler<MoveCustomerCommand>>, MoveCustomerHandler>();
container.Register<IHandler<ShipOrderCommand>>, ShipOrderHandler>();

// IHandler<T> implements IHandler
container.RegisterInitializer<IHandler>(handlerToInitialize => {
    handlerToInitialize.ExecuteAsynchronously = true;
});

// Usage
var handler1 = container.GetInstance<IHandler<MoveCustomerCommand>>();
Assert.IsTrue(handler1.ExecuteAsynchronously);

var handler2 = container.GetInstance<IHandler<ShipOrderCommand>>();
Assert.IsTrue(handler2.ExecuteAsynchronously);
```

The *Action<T>* delegate that is registered by the **RegisterInitializer** method is called once the *Container* has created a new instance of *T* (or any instance that inherits from or implements *T* depending on exactly how you have configured your registrations). In the example *MoveCustomerHandler* implements *IHandler* and because of this the *Action<IHandler>* delegate will be called with a reference to the newly created instance.

Note: The *Container* will not be able to call an initializer delegate on a type that is manually constructed using the *new* operator. Use *automatic constructor injection* whenever possible.

Tip: Multiple initializers can be applied to a concrete type and the *Container* will call all initializers that apply. They are **guaranteed** to run in the same order that they are registered.

Automatic/Batch-registration

When an application starts to grow, so does the number of types to register in the container. This can cause a lot of maintenance on part of your application that holds your container registration. When working with a team, you'll start to experience many merge conflicts which increases the chance of errors.

To minimize these problems, Simple Injector allows groups of types to be registered with a few lines of code. Especially when registering a family of types that are defined using the same (generic) interface. For instance, the previous example with the *IHandler<T>* registrations can be reduced to the following code:

```
// Configuration
Assembly[] assemblies = // determine list of assemblies to search in
container.Register(typeof(IHandler<>), assemblies);
```

When supplying a list of assemblies to the **Register** method, Simple Injector will go through the assemblies and will register all types that implement the given interface. In this example, an open-generic type (*IHandler<T>*) is supplied. Simple Injector will automatically find all implementations of this interface.

Note: For more information about batch registration, please see the *Batch-registration* section.

Collections

Simple Injector contains several methods for registering and resolving collections of types. Here are some examples:

```
// Configuration
// Registering a list of instances that will be created by the container.
// Supplying a collection of types is the preferred way of registering collections.
container.RegisterCollection<ILogger>(new[] { typeof(MailLogger), typeof(SqlLogger) }
↪);

// Register a fixed list (these instances should be thread-safe).
container.RegisterCollection<ILogger>(new[] { new MailLogger(), new SqlLogger() });

// Using a collection from another subsystem
container.RegisterCollection<ILogger>(Logger.Providers);

// Usage
var loggers = container.GetAllInstances<ILogger>();
```

Note: When zero instances are registered using **RegisterCollection**, each call to **Container.GetAllInstances** will return an empty list.

Warning: Simple Injector requires a call to **RegisterCollection** to be made, even in the absence of any instances. Without a call to **RegisterCollection**, Simple Injector will throw an exception.

Just as with normal types, Simple Injector can inject collections of instances into constructors:

```
// Definition
public class Service : IService {
    private readonly IEnumerable<ILogger> loggers;

    public Service(IEnumerable<ILogger> loggers) {
        this.loggers = loggers;
    }

    void IService.DoStuff() {
        // Log to all loggers
        foreach (var logger in this.loggers) {
            logger.Log("Some message");
        }
    }
}

// Configuration
container.RegisterCollection<ILogger>(new[] { typeof(MailLogger), typeof(SqlLogger) }
↪);
container.Register<IService, Service>(Lifestyle.Singleton);
```

```
// Usage
var service = container.GetInstance<IService>();
service.DoStuff();
```

The **RegisterCollection** overloads that take a collection of *Type* instances rely on the *Container* to create an instance of each type just as it would for individual registrations. This means that the same rules we have seen above apply to each item in the collection. Take a look at the following configuration:

```
// Configuration
container.Register<MailLogger>(Lifestyle.Singleton);
container.Register<ILogger, FileLogger>();

container.RegisterCollection<ILogger>(new[] {
    typeof(MailLogger),
    typeof(SqlLogger),
    typeof(ILogger)
});
```

When the registered collection of *ILogger* instances are resolved, the *Container* will resolve each of them applying the specific rules of their configuration. When no registration exists, the type is created with the default **Transient** lifestyle (*transient* means that a new instance is created every time the returned collection is iterated). In the example, the *MailLogger* type is registered as **Singleton**, and so each resolved *ILogger* collection will always have the same instance of *MailLogger* in their collection.

Since the creation is forwarded, abstract types can also be registered using **RegisterCollection**. In the above example the *ILogger* type itself is registered using **RegisterCollection**. This seems like a recursive definition, but it will work nonetheless. In this particular case you could imagine this to be a registration with a default ILogger registration which is also included in the collection of *ILogger* instances as well. A more usual scenario however is the use of a composite as shown next.

While resolving collections is useful and also works with *automatic constructor injection*, the registration of *Composites* is preferred over the use of collections as constructor arguments in application code. Register a composite whenever possible, as shown in the example below:

```
// Definition
public class CompositeLogger : ILogger {
    private readonly IEnumerable<ILogger> loggers;

    public CompositeLogger(IEnumerable<ILogger> loggers) {
        this.loggers = loggers;
    }

    public void Log(string message) {
        foreach (var logger in this.loggers) {
            logger.Log(message);
        }
    }
}

// Configuration
container.Register<IService, Service>(Lifestyle.Singleton);
container.Register<ILogger, CompositeLogger>(Lifestyle.Singleton);
container.RegisterCollection<ILogger>(new[] { typeof(MailLogger), typeof(SqlLogger) }
↪);

// Usage
```

```
var service = container.GetInstance<IService>();
service.DoStuff();
```

When using this approach none of your services (except *CompositeLogger*) need a dependency on *IEnumerable<ILogger>* - they can all simply have a dependency on the *ILogger* interface itself.

Collection types

Besides *IEnumerable<ILogger>*, Simple Injector natively supports some other collection types as well. The following types are supported:

- *IEnumerable<T>*
- *ICollection<T>*
- *IList<T>*
- *IReadOnlyCollection<T>*
- *IReadOnlyList<T>*
- *T[]* (array)

Note: The *IReadOnlyCollection<T>* and *IReadOnlyList<T>* interfaces are new in .NET 4.5. Only the .NET 4.5 and .NET Standard (including .NET Core and Xamarin) builds of Simple Injector will be able to automatically inject these abstractions into your components. These interfaces are *not* supported by the .NET 4.0 version of Simple Injector.

Simple Injector preserves the lifestyle of instances that are returned from an injected *IEnumerable<T>*, *ICollection<T>*, *IList<T>*, *IReadOnlyCollection<T>* and *IReadOnlyList<T>* instance. In reality you should not see the the injected *IEnumerable<T>* as a collection of instances; you should consider it a **stream** of instances. Simple Injector will always inject a reference to the same stream (the *IEnumerable<T>* or *ICollection<T>* itself is a singleton) and each time you iterate the *IEnumerable<T>*, for each individual component, the container is asked to resolve the instance based on the lifestyle of that component.

Warning: In contrast to the collection abstractions, an **array** is registered as **transient**. An array is a mutable type; a consumer can change the contents of an array. Sharing the array (by making it singleton) might cause unrelated parts of your applications to fail because of changes to the array. Since an array is a concrete type, it can not function as a stream, causing the elements in the array to get the lifetime of the consuming component. This could cause *lifestyle mismatches* when the array wasn't registered as transient.

Batch-registering collections

Just as with one-to-one mappings, Simple Injector allows collections of types to be batch-registered. There are overloads of the **RegisterCollection** method that accept a list of *Assembly* instances. Simple Injector will go through those assemblies to look for implementations of the supplied type:

```
Assembly[] assemblies = // determine list of assemblies to search in
container.RegisterCollection(typeof(ILogger), assemblies);
```

The previous code snippet will register all *ILogger* implementations that can be found in the supplied assemblies as part of the collection.

Warning: This **RegisterCollection** overload will request all the types from the supplied *Assembly* instances. The CLR however does not give *any* guarantees what so ever about the order in which these types are returned. Don't be surprised if the order of these types in the collection change after a recompile or an application restart.

Note: For more information about batch registration, please see the *Batch-registration* section.

Adding registrations to an existing collection

In most cases you would register a collection with a single line of code. There are cases where you need to append registrations to an already registered collection. Common use cases for this are integration scenarios where you need to interact with some framework that made its own registrations on your behalf, or in cases where you want to add extra types based on configuration settings. In these cases it might be benifecial to append registrations to an existing collection.

To be able to do this, Simple Injector contains the **AppendToCollection** extension method in the **SimpleInjector.Advanced** namespace.

Note: This extension method will not show up using IntelliSense, unless you include the **SimpleInjector.Advanced** namespace to your code file. This extension method is deliberately hidden to prevent polluting the main API; appending to existing collections is not a common use case.

```
Assembly[] assemblies = // determine list of assemblies to search in
container.RegisterCollection(typeof(ILogger), assemblies);

container.AppendToCollection(typeof(ILogger), typeof(ExtraLogger));
```

Verifying the container's configuration

You can call the *Verify* method of the *Container*. The *Verify* method provides a fail-fast mechanism to prevent your application from starting when the *Container* has been accidentally misconfigured. The *Verify* method checks the entire configuration by creating an instance of each registered type.

Tip: Calling **Verify** is not required, but is *highly encouraged*.

For more information about creating an application and container configuration that can be successfully verified, please read the *How To Verify the container's configuration*.

Automatic constructor injection / auto-wiring

Simple Injector uses the public constructor of a registered type and analyzes each constructor argument. The *Container* will resolve an instance for each argument type and then invoke the constructor using those instances. This mechanism is called *Automatic Constructor Injection* or *auto-wiring* and is one of the fundamental features that separates a DI Container from applying DI by hand.

Simple Injector has the following prerequisites to be able to provide auto-wiring:

1. Each type to be created must be concrete (not abstract, an interface or an open generic type). Types may be internal, although this can be limited if you're running in a sandbox (e.g. Silverlight or Windows Phone).
2. The type *should* have one public constructor (this may be a default constructor).
3. All the types of the arguments in that constructor must be resolvable by the *Container*; optional arguments are not supported.

Simple Injector can create a concrete type even if it hasn't been registered explicitly in the container by using constructor injection.

The following code shows an example of the use of automatic constructor injection. The example shows an *IUserRepository* interface with a concrete *SqlUserRepository* implementation and a concrete *UserService* class. The *UserService* class has one public constructor with an *IUserRepository* argument. Because the dependencies of the *UserService* are registered, Simple Injector is able to create a new *UserService* instance.

```
// Definitions
public interface IUserRepository { }
public class SqlUserRepository : IUserRepository { }
public class UserService : IUserService {
    private readonly IUserRepository repository;
    public UserService(IUserRepository repository) {
        this.repository = repository;
    }
}

// Configuration
var container = new Container();

container.Register<IUserRepository, SqlUserRepository>(Lifestyle.Singleton);
container.Register<IUserService, UserService>(Lifestyle.Singleton);

// Usage
var service = container.GetInstance<IUserService>();
```

Note: Because *UserService* is a concrete type, calling *container.GetInstance<UserService>()* without registering it explicitly will work. This feature can simplify the *Container*'s configuration for some scenarios. Always keep in mind that best practice is to program to an interface not a concrete type. Prevent using and depending on concrete types as much possible.

Warning: Even though registration of concrete types is not required, it is adviced to register all root types explicitly. For instance, register all ASP.NET MVC Controller instances explicitly in the container (Controller instances are requested directly and are therefore called 'root objects'). This way the container can check the complete dependency graph starting from the root object when you call **Verify()**.

More information

For more information about Simple Injector please visit the following links:

- The *Object Lifetime Management* page explains how to configure lifestyles such as **transient**, **singleton**, and many others.
- See the *Integration Guide* for more information about how to integrate Simple Injector into your specific application framework.
- For more information about dependency injection in general, please visit this page on Stackoverflow[20].
- If you have any questions about how to use Simple Injector or about dependency injection in general, the experts at Stackoverflow.com[21] are waiting for you.
- For all other Simple Injector related question and discussions, such as bug reports and feature requests, the Simple Injector discussion forum[22] will be the place to start.

[20] https://stackoverflow.com/tags/dependency-injection/info
[21] https://stackoverflow.com/questions/ask?tags=simple-injector%20ioc-container%20dependency-injection%20.net%20c%23
[22] https://simpleinjector.org/forum

CHAPTER 3

Object Lifetime Management

Object Lifetime Management is the concept of controlling the number of instances a configured service will have and the duration of the lifetime of those instances. In other words, it allows you to determine how returned instances are cached. Most DI libraries have sophisticated mechanisms for lifestyle management, and Simple Injector is no exception with built-in support for the most common lifestyles. The three default lifestyles (transient, scoped and singleton) are part of the core library. Implementations for the scoped lifestyle can be found within some of the extension and integration packages. The built-in lifestyles will suit about 99% of cases. For anything else custom lifestyles can be used.

Below is a list of the most common lifestyles with code examples of how to configure them using Simple Injector:

Lifestyle	Description	Disposal
Transient	A new instance of the component will be created each time the service is requested from the container. If multiple consumers depend on the service within the same graph, each consumer will get its own new instance of the given service.	Never
Scoped	For every request within an implicitly or explicitly defined scope.	Instances will be disposed when their scope ends.
Singleton	There will be at most one instance of the registered service type and the container will hold on to that instance until the container is disposed or goes out of scope. Clients will always receive that same instance from the container.	Instances will be disposed when the container is disposed.

Many different platform and framework specific flavors are available for the *Scoped* lifestyle. Please see the *Scoped* section for more information.

Further reading:

- *Per Graph*
- *Instance Per Dependency*
- *Per Thread*
- *Per HTTP Session*
- *Hybrid*

- *Developing a Custom Lifestyle*

Transient

A new instance of the service type will be created each time the service is requested from the container. If multiple consumers depend on the service within the same graph, each consumer will get its own new instance of the given service.

This example instantiates a new *IService* implementation for each call, while leveraging the power of *automatic constructor injection*.

```
container.Register<IService, RealService>(Lifestyle.Transient);

// Alternatively, you can use the following short cut
container.Register<IService, RealService>();
```

The next example instantiates a new *RealService* instance on each call by using a delegate.

```
container.Register<IService>(() => new RealService(new SqlRepository()),
    Lifestyle.Transient);
```

Note: It is normally recommended that registrations are made using **Register<TService, TImplementation>()**. It is easier, leads to less fragile configuration, and results in faster retrieval than registrations using a *Func<T>* delegate. Always try the former approach before resorting to using delegates.

Warning: Transient instances are not tracked by the container. This means that Simple Injector will not dispose transient instances. Simple Injector will detect disposable instances that are registered as transient when calling *container.Verify()*. Please view *Diagnostic Warning - Disposable Transient Components* for more information.

Singleton

There will be at most one instance of the registered service type and the container will hold on to that instance until the container is disposed or goes out of scope. Clients will always receive that same instance from the container.

There are multiple ways to register singletons. The most simple and common way to do this is by specifying both the service type and the implementation as generic type arguments. This allows the implementation type to be constructed using automatic constructor injection:

```
container.Register<IService, RealService>(Lifestyle.Singleton);
```

You can also use the *RegisterSingleton<T>(T)* overload to assign a constructed instance manually:

```
var service = new RealService(new SqlRepository());
container.RegisterSingleton<IService>(service);
```

There is also an overload that takes an *Func<T>* delegate. The container guarantees that this delegate is called only once:

```
container.Register<IService>(() => new RealService(new SqlRepository()),
    Lifestyle.Singleton);

// Or alternatively:
container.RegisterSingleton<IService>(() => new RealService(new SqlRepository()));
```

Alternatively, when needing to register a concrete type as singleton, you can use the parameterless **RegisterSingleton<T>()** overload. This will inform the container to automatically construct that concrete type (at most) once, and return that instance on each request:

```
container.RegisterSingleton<RealService>();

// Which is a more convenient short cut for:
container.Register<RealService, RealService>(Lifestyle.Singleton);
```

Registration for concrete singletons is necessarily, because unregistered concrete types will be treated as transient.

Warning: Simple Injector guarantees that there is at most one instance of the registered **Singleton** inside that **Container** instance, but if multiple **Container** instances are created, each **Container** instance will get its own instance of the registered **Singleton**.

Note: Simple Injector will cache a **Singleton** instance for the lifetime of the **Container** instance and will dispose any auto-wired instance (that implements *IDisposable*) when **Container.Dispose()** is called. This includes registrations using **RegisterSingleton<TService, TImplementation>()**, **RegisterSingleton<TConcrete>()** and **RegisterSingleton(Type, Type)**. Non-auto-wired instances that are created using factory delegates will be disposed as well. This includes **RegisterSingleton<TService>(Func<TService>)** and **RegisterSingleton(Type, Func<object>)**.

Warning: Already existing instances that are supplied to the container using **RegisterSingleton<TService>(TService)** and **RegisterSingleton(Type, object)** will not be disposed by the container. They are considered to be 'externally owned'.

Note: Simple Injector guarantees that instances are disposed in opposite order of creation. See: *Order of disposal* for more information.

Scoped

For every request within an implicitly or explicitly defined scope, a single instance of the service will be returned and that instance will be disposed when the scope ends.

Simple Injector contains the following scoped lifestyles:

Lifestyle	Description	Disposal
Thread Scoped	Within a certain (explicitly defined) scope, there will be only one instance of a given service type A created scope is specific to one particular thread, and can't be moved across threads.	Instance will be disposed when their scope gets disposed.
Async Scoped	There will be only one instance of a given service type within a certain (explicitly defined) scope. This scope will automatically flow with the logical flow of control of asynchronous methods.	Instance will be disposed when their scope gets disposed.
Web Request	Only one instance will be created by the container per web request. Use this lifestyle in ASP.NET Web Forms and ASP.NET MVC applications.	Instances will be disposed when the web request ends.
WCF Operation	Only one instance will be created by the container during the lifetime of the WCF service class.	Instances will be disposed when the WCF service class is released.

Web Request and *WCF Operation* implement scoping implicitly, which means that the user does not have to start or finish the scope to allow the lifestyle to end and to dispose cached instances. The *Container* does this for you. With the *Thread Scoped* and *Async Scoped* lifestyles on the other hand, you explicitly define a scope (just like you would do with .NET's TransactionScope class).

Most of the time, you will only use one particular scoped lifestyle per application. To simplify this, Simple Injector allows configuring the default scoped lifestyle in the container. After configuring the default scoped lifestyle, the rest of the configuration can access this lifestyle by calling **Lifestyle.Scoped**, as can be seen in the following example:

```
var container = new Container();
// Set the scoped lifestyle one directly after creating the container
container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

// Use the Lifestyle.Scoped everywhere in your configuration.
container.Register<IUserContext, AspNetUserContext>(Lifestyle.Scoped);
container.Register<MyAppUnitOfWork>(() => new MyAppUnitOfWork("constr"),
    Lifestyle.Scoped);
```

Just like *Singleton* registrations, instances of scoped registrations that are created by the container will be disposed when the their scope ends. Scoped lifestyles are especially useful for implementing patterns such as the Unit of Work[23].

Order of disposal

Simple Injector guarantees that instances are disposed in opposite order of creation.

When a component *A* depends on component *B*, *B* will be created before *A*. This means that *A* will be disposed before *B* (assuming both implement *IDisposable*), since the guarantee of opposite order of creation. This allows *A* to use *B* while *A* is being disposed.

Thread Scoped

Within a certain (explicitly defined) scope, there will be only one instance of a given service type in that thread and the instance will be disposed when the scope ends. A created scope is specific to one particular thread, and can't be moved across threads.

Warning: A thread scoped lifestyle can't be used for asynchronous operations (using the async/await keywords in C#).

SimpleInjector.Lifestyles.ThreadScopedLifestyle is part of the Simple Injector core library. The following examples shows its typical usage:

```
var container = new Container();
container.Options.DefaultScopedLifestyle = new ThreadScopedLifestyle();

container.Register<IUnitOfWork, NorthwindContext>(Lifestyle.Scoped);
```

Within an explicitly defined scope, there will be only one instance of a service that is defined with the *Thread Scoped* lifestyle:

```
using (ThreadScopedLifestyle.BeginScope(container)) {
    var uow1 = container.GetInstance<IUnitOfWork>();
    var uow2 = container.GetInstance<IUnitOfWork>();

    Assert.AreSame(uow1, uow2);
}
```

Warning: The *ThreadScopedLifestyle* is *thread-specific*. A single scope should **not** be used over multiple threads. Do not pass a scope between threads and do not wrap an ASP.NET HTTP request with a *ThreadScopedLifestyle*, since ASP.NET can finish a web request on different thread to the thread the request is started on. Use *Web Request Lifestyle* scoping for ASP.NET Web Forms and MVC web applications while running inside a web request. Use *Async Scoped Lifestyle* when using ASP.NET Web API or ASP.NET Core. *ThreadScopedLifestyle* however, can still be used in web

[23] http://martinfowler.com/eaaCatalog/unitOfWork.html

applications on background threads that are created by web requests or when processing commands in a Windows Service (where each command gets its own scope). For developing multi-threaded applications, take *these guidelines* into consideration.

Outside the context of a thread scoped lifestyle, i.e. *using (ThreadScopedLifestyle.BeginScope(container))* no instances can be created. An exception is thrown when a thread scoped registration is requested outside of a scope instance.

Scopes can be nested and each scope will get its own set of instances:

```
using (ThreadScopedLifestyle.BeginScope(container)) {
    var outer1 = container.GetInstance<IUnitOfWork>();
    var outer2 = container.GetInstance<IUnitOfWork>();

    Assert.AreSame(outer1, outer2);

    using (ThreadScopedLifestyle.BeginScope(container)) {
        var inner1 = container.GetInstance<IUnitOfWork>();
        var inner2 = container.GetInstance<IUnitOfWork>();

        Assert.AreSame(inner1, inner2);

        Assert.AreNotSame(outer1, inner1);
    }
}
```

Async Scoped (async/await)

There will be only one instance of a given service type within a certain (explicitly defined) scope and that instance will be disposed when the scope ends. This scope will automatically flow with the logical flow of control of asynchronous methods.

This lifestyle is meant for applications that work with the new asynchronous programming model.

SimpleInjector.Lifestyles.AsyncScopedLifestyle is part of the Simple Injector core library. The following examples shows its typical usage:

```
var container = new Container();
container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

container.Register<IUnitOfWork, NorthwindContext>(Lifestyle.Scoped);
```

Within an explicitly defined scope, there will be only one instance of a service that is defined with the *Async Scoped* lifestyle:

```
using (AsyncScopedLifestyle.BeginScope(container)) {
    var uow1 = container.GetInstance<IUnitOfWork>();
    await SomeAsyncOperation();
    var uow2 = container.GetInstance<IUnitOfWork>();
    await SomeOtherAsyncOperation();

    Assert.AreSame(uow1, uow2);
}
```

Note: A scope is specific to the asynchronous flow. A method call on a different (unrelated) thread, will get its own scope.

Outside the context of an active async scope no instances can be created. An exception is thrown when this happens.

Scopes can be nested and each scope will get its own set of instances:

```
using (AsyncScopedLifestyle.BeginScope(container)) {
    var outer1 = container.GetInstance<IUnitOfWork>();
    await SomeAsyncOperation();
    var outer2 = container.GetInstance<IUnitOfWork>();

    Assert.AreSame(outer1, outer2);

    using (AsyncScopedLifestyle.BeginScope(container)) {
        var inner1 = container.GetInstance<IUnitOfWork>();

        await SomeOtherAsyncOperation();

        var inner2 = container.GetInstance<IUnitOfWork>();

        Assert.AreSame(inner1, inner2);

        Assert.AreNotSame(outer1, inner1);
    }
}
```

Web Request

Only one instance will be created by the container per web request and the instance will be disposed when the web request ends.

The ASP.NET Integration NuGet Package[24] is available (and available as **SimpleInjector.Integration.Web.dll** in the default download) contains a **WebRequestLifestyle** class that enable easy *Per Web Request* registrations:

```
var container = new Container();
container.Options.DefaultScopedLifestyle = new WebRequestLifestyle();

container.Register<IUserRepository, SqlUserRepository>(Lifestyle.Scoped);
container.Register<IOrderRepository, SqlOrderRepository>(Lifestyle.Scoped);
```

Tip: For ASP.NET MVC, there's a Simple Injector MVC Integration Quick Start[25] NuGet Package available that helps you get started with Simple Injector in MVC applications quickly.

Web Async Scoped lifestyle vs. Web Request lifestyle

The lifestyles and scope implementations **Web Request** and **Async Scoped** in Simple Injector are based on different technologies. **AsyncScopedLifestyle** works well both inside and outside of IIS. i.e. It can function in a self-hosted Web API project where there is no *HttpContext.Current*. As the name implies, an async scope registers itself flows with *async* operations across threads (e.g. a continuation after *await* on a different thread still has access to the scope regardless of whether *ConfigureAwait()* was used with *true* or *false*).

In contrast, the **Scope** of the **WebRequestLifestyle** is stored within the *HttpContext.Items* dictionary. The *HttpContext* can be used with Web API when it is hosted in IIS but care must be taken because it will not always flow with the async operation, because the current *HttpContext* is stored in the *IllogicalCallContext* (see Understanding SynchronizationContext in ASP.NET[26]). If you use *await* with *ConfigureAwait(false)* the continuation may lose track of

[24] https://nuget.org/packages/SimpleInjector.Integration.Web
[25] https://nuget.org/packages/SimpleInjector.MVC3
[26] https://blogs.msdn.com/b/pfxteam/archive/2012/06/15/executioncontext-vs-synchronizationcontext.aspx

the original *HttpContext* whenever the async operation does not execute synchronously. A direct effect of this is that it would no longer be possible to resolve the instance of a previously created service with **WebRequestLifestyle** from the container (e.g. in a factory that has access to the container) - and an exception would be thrown because *HttpContext.Current* would be null.

The recommendation is to use **AsyncScopedLifestyle** in for applications that solely consist of a Web API (or other asynchronous technologies such as ASP.NET Core) and use **WebRequestLifestyle** for applications that contain a mixture of Web API and MVC.

AsyncScopedLifestyle offers the following benefits when used in Web API:

- The Web API controller can be used outside of IIS (e.g. in a self-hosted project)
- The Web API controller can execute *free-threaded* (or *multi-threaded*) *async* methods because it is not limited to the ASP.NET *SynchronizationContext*.

For more information, check out the blog entry of Stephen Toub regarding the difference between ExecutionContext and SynchronizationContext[27].

WCF Operation

Only one instance will be created by the container during the lifetime of the WCF service class and the instance will be disposed when the WCF service class is released.

The WCF Integration NuGet Package[28] is available (and available as **SimpleInjector.Integration.Wcf.dll** in the default download) contains a **WcfOperationLifestyle** class that enable easy *Per WCF Operation* registrations:

```
var container = new Container();
container.Options.DefaultScopedLifestyle = new WcfOperationLifestyle();

container.Register<IUserRepository, SqlUserRepository>(Lifestyle.Scoped);
container.Register<IOrderRepository, SqlOrderRepository>(Lifestyle.Scoped);
```

Warning: Instead of what the name of the **WcfOperationLifestyle** class seems to imply, components that are registered with this lifestyle might actually outlive a single WCF operation. This behavior depends on how the WCF service class is configured. WCF is in control of the lifetime of the service class and contains three lifetime types as defined by the InstanceContextMode enumeration[29]. Components that are registered *PerWcfOperation* live as long as the WCF service class they are injected into.

For more information about integrating Simple Injector with WCF, please see the *WCF integration guide*.

Per Graph

For each explicit call to **Container.GetInstance<T>** a new instance of the service type will be created, but the instance will be reused within the object graph that gets constructed.

Compared to **Transient**, there will be just a single instance per explicit call to the container, while **Transient** services can have multiple new instances per explicit call to the container. This lifestyle is not supported by Simple Injector but can be simulated by using one of the *Scoped* lifestyles.

[27] https://vegetarianprogrammer.blogspot.de/2012/12/understanding-synchronizationcontext-in.html
[28] https://nuget.org/packages/SimpleInjector.Integration.Wcf
[29] https://msdn.microsoft.com/en-us/library/system.servicemodel.instancecontextmode.aspx

Instance Per Dependency

Each consumer will get a new instance of the given service type and that dependency is expected to get live as long as its consuming type.

This lifestyle behaves the same as the built-in **Transient** lifestyle, but the intend is completely different. A **Transient** instance is expected to have a very short lifestyle and injecting it into a consumer with a longer lifestyle (such as **Singleton**) is an error. Simple Injector will prevent this from happening by checking for *lifestyle mismatches*. With the *Instance Per Dependency* lifestyle on the other hand, the created component is expected to stay alive as long as the consuming component does. So when the *Instance Per Dependency* component is injected into a **Singleton** component, we intend it to be kept alive by its consumer.

This lifestyle is deliberately left out of Simple Injector, because its usefulness is very limited compared to the **Transient** lifestyle. It ignores *lifestyle mismatch checks* and this can easily lead to errors, and it ignores the fact that application components should be immutable. In case a component is immutable, it's very unlikely that each consumer requires its own instance of the injected dependency.

Per Thread

There will be one instance of the registered service type per thread.

This lifestyle is deliberately left out of Simple Injector because *it is considered to be harmful*. Instead of using Per-Thread lifestyle, you will usually be better of using the *Thread Scoped Lifestyle*.

Per HTTP Session

There will be one instance of the registered session per (user) session in a ASP.NET web application.

This lifestyle is deliberately left out of Simple Injector because it is be used with care[30]. Instead of using Per HTTP Session lifestyle, you will usually be better of by writing a stateless service that can be registered as singleton and let it communicate with the ASP.NET Session cache to handle cached user-specific data.

Hybrid

A hybrid lifestyle is a mix between two or more lifestyles where the the developer defines the context for which the wrapped lifestyles hold.

Simple Injector has no built-in hybrid lifestyles, but has a simple mechanism for defining them:

```
var container = new Container();

container.Options.DefaultScopedLifestyle = Lifestyle.CreateHybrid(
    defaultLifestyle: new ThreadScopedLifestyle(),
    fallbackLifestyle: new WebRequestLifestyle());

container.Register<IUserRepository, SqlUserRepository>(Lifestyle.Scoped);
container.Register<ICustomerRepository, SqlCustomerRepository>(Lifestyle.Scoped);
```

[30] https://stackoverflow.com/questions/17702546

In the example a hybrid lifestyle is defined wrapping the *Thread Scoped Lifestyle* and the *Web Request Lifestyle*. This hybrid lifestyle will use the *ThreadScopedLifestyle*, but will fall back to the *WebRequestLifestyle* in case there is no active thread scope.

A hybrid lifestyle is useful for registrations that need to be able to dynamically switch lifestyles throughout the lifetime of the application. The shown hybrid example might be useful in a web application, where some operations need to be run in isolation (with their own instances of scoped registrations such as unit of works) or run outside the context of an *HttpContext* (in a background thread for instance).

Please note though that when the lifestyle doesn't have to change throughout the lifetime of the application, a hybrid lifestyle is not needed. A normal lifestyle can be registered instead:

```
bool runsOnWebServer = ReadConfigurationValue<bool>("RunsOnWebServer");

var container = new Container();
container.Options.DefaultScopedLifestyle =
    runsOnWebServer ? new WebRequestLifestyle() : new ThreadScopedLifestyle();

container.Register<IUserRepository, SqlUserRepository>(Lifestyle.Scoped);
container.Register<ICustomerRepository, SqlCustomerRepository>(Lifestyle.Scoped);
```

Developing a Custom Lifestyle

The lifestyles supplied by Simple Injector should be sufficient for most scenarios, but in rare circumstances defining a custom lifestyle might be useful. This can be done by creating a class that inherits from Lifestyle[31] and let it return Custom Registration[32] instances. This however is a lot of work, and a shortcut is available in the form of the Lifestyle.CreateCustom[33].

A custom lifestyle can be created by calling the **Lifestyle.CreateCustom** factory method. This method takes two arguments: the name of the lifestyle to create (used mainly by the *Diagnostic Services*) and a CreateLifestyleApplier[34] delegate:

```
public delegate Func<object> CreateLifestyleApplier(
    Func<object> transientInstanceCreator)
```

The **CreateLifestyleApplier** delegate accepts a *Func<object>* that allows the creation of a transient instance of the registered type. This *Func<object>* is created by Simple Injector supplied to the registered **CreateLifestyleApplier** delegate for the registered type. When this *Func<object>* delegate is called, the creation of the type goes through the *Simple Injector pipeline*. This keeps the experience consistent with the rest of the library.

When Simple Injector calls the **CreateLifestyleApplier**, it is your job to return another *Func<object>* delegate that applies the caching based on the supplied *instanceCreator*. A simple example would be the following:

```
var sillyTransientLifestyle = Lifestyle.CreateCustom(
    name: "Silly Transient",
    // instanceCreator is of type Func<object>
    lifestyleApplierFactory: instanceCreator => {
        // A Func<object> is returned that applies caching.
        return () => instanceCreator.Invoke();
    });

var container = new Container();
```

[31] https://simpleinjector.org/ReferenceLibrary/?topic=html/T_SimpleInjector_Lifestyle.htm
[32] https://simpleinjector.org/ReferenceLibrary/?topic=html/T_SimpleInjector_Registration.htm
[33] https://simpleinjector.org/ReferenceLibrary/?topic=html/M_SimpleInjector_Lifestyle_CreateCustom.htm
[34] https://simpleinjector.org/ReferenceLibrary/?topic=html/T_SimpleInjector_CreateLifestyleApplier.htm

```
container.Register<IService, MyService>(sillyTransientLifestyle);
```

Here we create a custom lifestyle that applies no caching and simply returns a delegate that will on invocation always call the wrapped *instanceCreator*. Of course this would be rather useless and using the built-in **Lifestyle.Transient** would be much better in this case. It does however demonstrate its use.

The *Func<object>* delegate that you return from your **CreateLifestyleApplier** delegate will get cached by Simple Injector per registration. Simple Injector will call the delegate once per registration and stores the returned *Func<object>* for reuse. This means that each registration will get its own *Func<object>*.

Here's an example of the creation of a more useful custom lifestyle that caches an instance for 10 minutes:

```
var tenMinuteLifestyle = Lifestyle.CreateCustom(
    name: "Absolute 10 Minute Expiration",
    lifestyleApplierFactory: instanceCreator => {
        TimeSpan timeout = TimeSpan.FromMinutes(10);
        var syncRoot = new object();
        var expirationTime = DateTime.MinValue;
        object instance = null;

        return () => {
            lock (syncRoot) {
                if (expirationTime < DateTime.UtcNow) {
                    instance = instanceCreator.Invoke();
                    expirationTime = DateTime.UtcNow.Add(timeout);
                }
                return instance;
            }
        };
    });

var container = new Container();

// We can reuse the created lifestyle for multiple registrations.
container.Register<IService, MyService>(tenMinuteLifestyle);
container.Register<AnotherService, MeTwoService>(tenMinuteLifestyle);
```

In this example the **Lifestyle.CreateCustom** method is called and supplied with a delegate that returns a delegate that applies the 10 minute cache. This example makes use of the fact that each registration gets its own delegate by using four closures (timeout, syncRoot, expirationTime and instance). Since each registration (in the example *IService* and *AnotherService*) will get its own *Func<object>* delegate, each registration gets its own set of closures. The closures are therefore static per registration.

One of the closure variables is the *instance* and this will contain the cached instance that will change after 10 minutes has passed. As long as the time hasn't passed, the same instance will be returned.

Since the constructed *Func<object>* delegate can be called from multiple threads, the code needs to do its own synchronization. Both the DateTime comparison and the DateTime assignment are not thread-safe and this code needs to handle this itself.

Do note that even though locking is used to synchronize access, this custom lifestyle might not work as expected, because when the expiration time passes while an object graph is being resolved, it might result in an object graph that contains two instances of the registered component, which might not be what you want. This example therefore is only for demonstration purposes.

In case you wish to develop a custom lifestyle, we strongly advice posting a question on the Forum. We will be able to guide you through this process.

CHAPTER 4

Integration Guide

Simple Injector can be used in a wide range of .NET technologies, both server side as client side. Jump directly to the integration page for the application framework of your choice. When the framework of your choice is not listed, doesn't mean it isn't supported, but just that we didn't have the time write it :-)

Console Application Integration Guide

Simple Injector contains several integration packages that simplify plugging-in Simple Injector into a wide variety of frameworks. These packages allow you to hook Simple Injector onto the framework's integration point.

When it comes to writing Console applications however, there are no integration packages. That's because Console applications are not backed by a particular framework. .NET just does the bare minimum when a Console application is started and there are no integration hooks that you can use. This means that you are in complete control over the application.

When a Console application is short-lived, and runs just a single operation and exits, the application could have a structure similar to the following:

```
using SimpleInjector;

static class Program
{
    static readonly Container container;

    static Program()
    {
        container = new Container();

        container.Register<IUserRepository, SqlUserRepository>();
        container.Register<MyRootType>();

        container.Verify();
    }
```

```
    static void Main()
    {
        var service = container.GetInstance<MyRootType>();
        service.DoSomething();
    }
}
```

Console applications can also be built to be long running, just like Windows services and web services are. In that case such Console application will typically handle many 'requests'.

The term 'request' is used loosely here. It could be an incoming request over the network, an action triggered by a message on an incoming queue, or an action triggered by timer or scheduler. Either way, a request is something that should typically run in a certain amount of isolation. It might for instance have its own set of state, specific to that request. An Entity Framework *DbContext* for instance, is typically an object that is particular to a single request.

In the absence of any framework code, you are yourself responsible to tell Simple Injector that certain code must run in isolation. This can be done with Scoping. There are two types of scoped lifestyles that can be used. *ThreadScopedLifestyle* allows wrapping code that runs on a single thread in a scope, where *AsyncScopedLifestyle* allows wrapping a block of code that flows asynchronously (using async await).

The following example demonstrates a simple Console application that runs indefinitely, and executes a request every second. The request is wrapped in a scope:

```
using SimpleInjector;
using SimpleInjector.Lifestyles;

static class Program {
    static readonly Container container;

    static Program() {
        container = new Container();
        container.Options.DefaultScopedLifestyle = new ThreadScopedLifestyle();

        container.Register<IUserRepository, SqlUserRepository>(Lifestyle.Scoped);
        container.Register<MyRootType>();

        container.Verify();
    }

    static void Main()  {
        while (true) {
            using (ThreadScopedLifestyle.BeginScope(container)) {
                var service = container.GetInstance<MyRootType>();

                service.DoSomething();
            }

            Thread.Sleep(TimeSpan.FromSeconds(1));
        }
    }
}
```

ASP.NET Core MVC Integration Guide

Simple Injector offers the Simple Injector ASP.NET Core MVC Integration NuGet package[35].

The following code snippet shows how to use the integration package to apply Simple Injector to your web application's *Startup* class.

```
// You'll need to include the following namespaces
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc.Controllers;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.AspNetCore.Http;

using SimpleInjector;
using SimpleInjector.Lifestyles;
using SimpleInjector.Integration.AspNetCore;
using SimpleInjector.Integration.AspNetCore.Mvc;

public class Startup {
    private Container container = new Container();

    public Startup(IHostingEnvironment env) {
        // ASP.NET default stuff here
    }

    // This method gets called by the runtime.
    public void ConfigureServices(IServiceCollection services) {
        // ASP.NET default stuff here
        services.AddMvc();

        IntegrateSimpleInjector(services);
    }

    private void IntegrateSimpleInjector(IServiceCollection services) {
        container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

        services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();

        services.AddSingleton<IControllerActivator>(
            new SimpleInjectorControllerActivator(container));
        services.AddSingleton<IViewComponentActivator>(
            new SimpleInjectorViewComponentActivator(container));

        services.EnableSimpleInjectorCrossWiring(container);
        services.UseSimpleInjectorAspNetRequestScoping(container);
    }

    // Configure is called after ConfigureServices is called.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env,
        ILoggerFactory factory) {

        InitializeContainer(app);
```

[35] https://www.nuget.org/packages/SimpleInjector.Integration.AspNetCore.Mvc

```
        container.Register<CustomMiddleware1>();
        container.Register<CustomMiddleware2>();

        container.Verify();

        // Add custom middleware
        app.Use((c, next) => container.GetInstance<CustomMiddleware1>().Invoke(c,␣
↪next));
        app.Use((c, next) => container.GetInstance<CustomMiddleware2>().Invoke(c,␣
↪next));

        // ASP.NET default stuff here
        app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }

    private void InitializeContainer(IApplicationBuilder app) {
        // Add application presentation components:
        container.RegisterMvcControllers(app);
        container.RegisterMvcViewComponents(app);

        // Add application services. For instance:
        container.Register<IUserService, UserService>(Lifestyle.Scoped);

        // Cross-wire ASP.NET services (if any). For instance:
        container.CrossWire<ILoggerFactory>(app);

        // NOTE: Do prevent cross-wired instances as much as possible.
        // See: https://simpleinjector.org/blog/2016/07/
    }
}
```

Wiring custom middleware

The previous *Startup* snippet already showed how a custom middleware class can be used in the ASP.NET Core pipeline. The following code snippet shows how such *CustomMiddleware* might look like:

```
// Example of some custom user-defined middleware component.
public sealed class CustomMiddleware {
    private readonly ILoggerFactory loggerFactory;
    private readonly IUserService userService;

    public CustomMiddleware(ILoggerFactory loggerFactory, IUserService userService) {
        this.loggerFactory = loggerFactory;
        this.userService = userService;
    }

    public async Task Invoke(HttpContext context, Func<Task> next) {
        // Do something before
        await next();
        // Do something after
```

```
    }
}
```

Notice how the *CustomMiddleware* class contains dependencies. Because of this, the *CustomMiddleware* class is resolved from Simple Injector on each request.

In contrast to what the official ASP.NET Core documentation advises[36], the *RequestDelegate* or *Func<Task> next* delegate can best be passed in using **Method Injection** (through the *Invoke* method), instead of by using Constructor Injection. Reason for this is that this delegate is runtime data and runtime data should not be passed in through the constructor[37]. Moving it to the *Invoke* method makes it possible to reliably verify the application's DI configuration and it simplifies your configuration.

Cross-wiring ASP.NET and third party services

When your application code (i.e. a *Controller*) needs a service which integrates with the ASP.NET Core configuration system it is sometimes necessary to cross-wire these dependencies. Cross-wiring is the process where a type is created and maintained by the ASP.NET Core configuration system and is fed to Simple Injector so Simple Injector can use the created instance to supply it as a dependency to your application code.

To use this feature, Simple Injector contains the **CrossWire<TService>** extension method. This method does the required blumbing such as making sure the type is registered with the same lifestyle as configured in ASP.NET Core.

To setup cross-wiring first you must make a call to **EnableSimpleInjectorCrossWiring** on *IServiceCollection* in the *ConfigureServices* method of your *Startup* class.

```
services.EnableSimpleInjectorCrossWiring(container);
```

When cross-wiring is enabled cross-wiring is as simple as:

```
container.CrossWire<ILoggerFactory>(app);
```

NOTE: Do prevent the use of cross-wiring as much as possible. In most cases cross-wiring is not the best solution and is a violation of the Dependency Inversion Principle[38]. Don't depend directly upon Framework components and instead create application specific proxy and/or adapter implementations.

Working with ASP.NET Core Identity

The default Visual Studio template comes with built-in authentication through the use of ASP.NET Core Identity. To get the code from the template working only a few services from Identity need to be cross-wired.

You can use this code snippet to get things working quickly

```
public class Startup
{
    private readonly Container container = new Container();

    public Startup(IHostingEnvironment env) {
        // ASP.NET default stuff here
    }

    // This method gets called by the runtime.
    public void ConfigureServices(IServiceCollection services) {
```

[36] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware#writing-middleware
[37] https://www.cuttingedge.it/blogs/steven/pivot/entry.php?id=99
[38] https://en.wikipedia.org/wiki/Dependency_inversion_principle

```
        // Add framework services for Identity.
        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection
↪")));

        services.AddIdentity<ApplicationUser, IdentityRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>()
            .AddDefaultTokenProviders();

        services.AddMvc();

        IntegrateSimpleInjector(services);
    }

    private void IntegrateSimpleInjector(IServiceCollection services) {
        container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

        services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();

        services.AddSingleton<IControllerActivator>(
            new SimpleInjectorControllerActivator(container));
        services.AddSingleton<IViewComponentActivator>(
            new SimpleInjectorViewComponentActivator(container));

        services.EnableSimpleInjectorCrossWiring(container);
        services.UseSimpleInjectorAspNetRequestScoping(container);
    }

    // Configure is called after ConfigureServices is called.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env,
        ILoggerFactory loggerFactory) {

        InitializeContainer(app);

        container.Verify();

        // ASP.NET default stuff here
        // Add Identity middleware
        app.UseIdentity();

        app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }

    private void InitializeContainer(IApplicationBuilder app) {
        // Add application presentation components:
        container.RegisterMvcControllers(app);
        container.RegisterMvcViewComponents(app);

        // Add application services for AccountController
        container.RegisterSingleton<IEmailSender, AuthMessageSender>();
        container.RegisterSingleton<ISmsSender, AuthMessageSender>();

        // Cross wire Identity services
```

```
        container.CrossWire<UserManager<ApplicationUser>>(app);
        container.CrossWire<SignInManager<ApplicationUser>>(app);

        // Cross wire other AccountController dependencies
        container.CrossWire<ILoggerFactory>(app);
        container.CrossWire<IOptions<IdentityCookieOptions>>(app);

        // NOTE: It is highly advisable to refactor the AccountController
        // and NOT to depend on IOptions<IdentityCookieOptions> and ILoggerFactory
        // See: https://simpleinjector.org/aspnetcore#working-with-ioption-t
    }
}
```

Working with *IOption<T>*

ASP.NET Core contains a new configuration model based on an *IOption<T>* abstraction. We advise against injecting *IOption<T>* dependencies into your application components. Instead let components depend directly on configuration objects and register them as *Singleton*. This ensures that configuration values are read during application start up and it allows verifying them at that point in time, allowing the application to fail-fast.

Letting application components depend on *IOptions<T>* has some unfortunate downsides. First of all, it causes application code to take an unnecessary dependency on a framework abstraction. This is a violation of the Dependency Injection Principle that prescribes the use of application-tailored abstractions. Injecting an *IOptions<T>* into an application component only makes this component more difficult to test, while providing no benefits. Application components should instead depend directly on the configuration values they require.

IOptions<T> configuration values are read lazily. Although the configuration file might be read upon application start up, the required configuration object is only created when *IOptions<T>.Value* is called for the first time. When deserialization fails, because of application misconfiguration, such error will only be appear after the call to *IOptions<T>.Value*. This can cause misconfigurations to keep undetected for much longer than required. By reading -and verifying- configuration values at application start up, this problem can be prevented. Configuration values can be injected as singletons into the component that requires them.

To make things worse, in case you forget to configure a particular section (by omitting a call to *services.Configure<T>*) or when you make a typo while retrieving the configuration section (by supplying the wrong name to *Configuration.GetSection(name)*), the configuration system will simply supply the application with a default and empty object instead of throwing an exception! This may make sense in some cases but it will easily lead to fragile applications.

Since you want to verify the configuration at start-up, it makes no sense to delay reading it, and that makes injecting IOption<T> into your components plain wrong. Depending on *IOptions<T>* might still be useful when bootstrapping the application, but not as a dependency anywhere else.

Once you have a correctly read and verified configuration object, registration of the component that requires the configuration object is as simple as this:

```
MyMailSettings mailSettings =
    config.GetSection("Root:SectionName").Get<MyMailSettings>();

// Verify mailSettings here (if required)

// Supply mailSettings as constructor argument to a type that requires it,
container.Register<IMessageSender>(() => new MailMessageSender(mailSettings));

// or register MailSettings as singleton in the container.
container.RegisterSingleton<MyMailSettings>(mailSettings);
container.Register<IMessageSender, MailMessageSender>();
```

ASP.NET MVC Integration Guide

Simple Injector contains Simple Injector MVC Integration Quick Start NuGet package[39].

Warning: If you are starting from an Empty MVC project template (File | New | Project | MVC 4 | Empty Project Template) you have to manually setup *System.Web.Mvc* binding redirects, or reference System.Web.Mvc from the GAC.

The following code snippet shows how to use the use the integration package (note that the quick start package injects this code into your Visual Studio MVC project).

```
// You'll need to include the following namespaces
using System.Web.Mvc;
using SimpleInjector;
using SimpleInjector.Integration.Web;
using SimpleInjector.Integration.Web.Mvc;

// This is the Application_Start event from the Global.asax file.
protected void Application_Start(object sender, EventArgs e) {
    // Create the container as usual.
    var container = new Container();
    container.Options.DefaultScopedLifestyle = new WebRequestLifestyle();

    // Register your types, for instance:
    container.Register<IUserRepository, SqlUserRepository>(Lifestyle.Scoped);

    // This is an extension method from the integration package.
    container.RegisterMvcControllers(Assembly.GetExecutingAssembly());

    container.Verify();

    DependencyResolver.SetResolver(new SimpleInjectorDependencyResolver(container));
}
```

ASP.NET Web API Integration Guide

Simple Injector contains Simple Injector ASP.NET Web API Integration Quick Start NuGet package for IIS-hosted applications[40]. If you're not using NuGet, you must include the **SimpleInjector.Integration.WebApi.dll** in your Web API application, which is part of the standard download on Github.

Note: To be able to run the Web API integration packages, you need[41] *.NET 4.5* or above.

Basic setup

The following code snippet shows how to use the integration package (note that the quick start package injects this code for you).

```
// You'll need to include the following namespaces
using System.Web.Http;
using SimpleInjector;
using SimpleInjector.Lifestyles;
```

[39] https://nuget.org/packages/SimpleInjector.MVC3
[40] https://www.nuget.org/packages/SimpleInjector.Integration.WebApi.WebHost.QuickStart
[41] https://stackoverflow.com/questions/22392032/are-there-any-technical-reasons-simpleinjector-cannot-support-webapi-on-net-4-0

```
using SimpleInjector.Integration.WebApi;

// This is the Application_Start event from the Global.asax file.
protected void Application_Start() {
    // Create the container as usual.
    var container = new Container();
    container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

    // Register your types, for instance using the scoped lifestyle:
    container.Register<IUserRepository, SqlUserRepository>(Lifestyle.Scoped);

    // This is an extension method from the integration package.
    container.RegisterWebApiControllers(GlobalConfiguration.Configuration);

    container.Verify();

    GlobalConfiguration.Configuration.DependencyResolver =
        new SimpleInjectorWebApiDependencyResolver(container);

    // Here your usual Web API configuration stuff.
}
```

With this configuration, ASP.NET Web API will create new *IHttpController* instances through the container. Because controllers are concrete classes, the container will be able to create them without any registration. However, to be able to *verify* and *diagnose* the container's configuration, it is important to register all root types explicitly, which is done by calling **RegisterWebApiControllers**.

Note: For Web API applications the use of the **AsyncScopedLifestyle** is advised over the **WebRequestLifestyle**. Please take a look at the *Web API Request Object Lifestyle Management wiki page* for more information.

Given the configuration above, an actual controller could look like this:

```
public class UserController : ApiController {
    private readonly IUserRepository repository;

    // Use constructor injection here
    public UserController(IUserRepository repository) {
        this.repository = repository;
    }

    public IEnumerable<User> GetAllUsers() => this.repository.GetAll();

    public User GetUserById(int id) {
        try {
            return this.repository.GetById(id);
        } catch (KeyNotFoundException) {
            throw new HttpResponseException(HttpStatusCode.NotFound);
        }
    }
}
```

Extra features

The basic features of the Web API integration package are the **SimpleInjectorWebApiDependencyResolver** class and the **RegisterWebApiControllers** extension method. Besides these basic features, the integration package contains extra features that can make your life easier.

Getting the current request's HttpRequestMessage

When working with Web API you will often find yourself wanting access to the current *HttpRequestMessage*. Simple Injector allows fetching the current *HttpRequestMessage* by calling the *container.GetCurrentHttpRequestMessage()* extension method. To be able to request the current *HttpRequestMessage* you need to explicitly enable this as follows:

```
container.EnableHttpRequestMessageTracking(GlobalConfiguration.Configuration);
```

There are several ways to get the current *HttpRequestMessage* in your services, but since it is discouraged to inject the *Container* itself into any services, the best way is to define an abstraction for this. For instance:

```
public interface IRequestMessageAccessor {
    HttpRequestMessage CurrentMessage { get; }
}
```

This abstraction can be injected into your services, which can call the *CurrentMessage* property to get the *HttpRequestMessage*. Close to your DI configuration you can now create an implementation for this interface as follows:

```
private sealed class RequestMessageAccessor : IRequestMessageAccessor {
    private readonly Container container;

    public RequestMessageAccessor(Container container) {
        this.container = container;
    }

    public HttpRequestMessage CurrentMessage =>
        this.container.GetCurrentHttpRequestMessage();
}
```

This implementation can be implemented as follows:

```
container.RegisterSingleton<IRequestMessageAccessor>(
    new RequestMessageAccessor(container));
```

Injecting dependencies into Web API filter attributes

Web API caches filter attribute instances indefinitely per action, effectively making them singletons. This makes them unsuited for dependency injection, since the attribute's dependencies will be accidentally promoted to singleton as well, which can cause all sorts of concurrency issues.

Since dependency injection is not an option here, an other mechanism is advised. There are basically two options here. Which one is best depends on the amount of filter attributes your application needs. If the number of attributes is limited (to a few), the simplest solution is to revert to the Service Locator pattern within your attributes. If the number of attributes is larger, it might be better to make attributes passive.

Reverting to the Service Locator pattern means that you need to do the following:

- Extract all the attribute's logic -with its dependencies- into a new service class.
- Resolve this service from within the filter attribute's *OnActionExecXXX* methods, but don't store the resolved service in a private field.
- Call the service's method.

The following example visualizes this:

```
public class MinimumAgeActionFilter : FilterAttribute {
    public readonly int MinimumAge;

    public MinimumAgeActionFilter(int minimumAge) {
        this.MinimumAge = minimumAge;
    }

    public override Task OnActionExecutingAsync(HttpActionContext actionContext,
        CancellationToken cancellationToken)
    {
        var checker = GlobalConfiguration.Configuration.DependencyResolver
            .GetService(typeof(IMinimumAgeChecker)) as IMinimumAgeChecker;

        checker.VerifyCurrentUserAge(this.MinimumAge);

        return TaskHelpers.Completed();
    }
}
```

By moving all the logic and dependencies out of the attribute, the attribute becomes a small infrastructural piece of code; a humble object that simply forwards the call to the real service.

If the number of required filter attributes grows, a different model might be in place. In that case you might want to make your attributes passive[42] as explained here[43].

Injecting dependencies into Web API message handlers

The default mechanism in Web API to use HTTP Message Handlers to 'decorate' requests is by adding them to the global *MessageHandlers* collection as shown here:

```
GlobalConfiguration.Configuration.MessageHandlers.Add(new MessageHandler1());
```

The problem with this approach is that this effectively hooks in the *MessageHandler1* into the Web API pipeline as a singleton. This is fine when the handler itself has no state and no dependencies, but in a system that is based on the SOLID design principles, it's very likely that those handlers will have dependencies of their own and its very likely that some of those dependencies need a lifetime that is shorter than singleton.

If that's the case, such message handler should not be created as singleton, since in general, a component should never have a lifetime that is longer than the lifetime of its dependencies.

The solution is to define a proxy class that sits in between. Since Web API lacks that functionality, we need to build this ourselves as follows:

```
public sealed class DelegatingHandlerProxy<THandler> : DelegatingHandler
    where THandler : DelegatingHandler {
    private readonly Container container;

    public DelegatingHandlerProxy(Container container) {
        this.container = container;
    }

    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken) {

        // Important: Trigger the creation of the scope.
```

[42] http://blog.ploeh.dk/2014/06/13/passive-attributes/
[43] https://www.cuttingedge.it/blogs/steven/pivot/entry.php?id=98

```
        request.GetDependencyScope();

        var handler = this.container.GetInstance<THandler>();

        if (!object.ReferenceEquals(handler.InnerHandler, this.InnerHandler)) {
            handler.InnerHandler = this.InnerHandler;
        }

        var invoker = new HttpMessageInvoker(handler);

        return invoker.SendAsync(request, cancellationToken);
    }
}
```

This *DelegatingHandlerProxy<THandler>* can be added as singleton to the global *MessageHandlers* collection, and it will resolve the given *THandler* on each request, allowing it to be resolved according to its lifestyle.

The *DelegatingHandlerProxy<THandler>* can be used as follows:

```
container.Register<MessageHandler1>();

GlobalConfiguration.Configuration.MessageHandlers.Add(
    new DelegatingHandlerProxy<MessageHandler1>(container));
```

ASP.NET Web Forms Integration Guide

ASP.NET Web Forms was never designed with dependency injection in mind. Although using constructor injection in our **Page** classes, user controls and HTTP handlers would be preferable, it is unfortunately not possible, because ASP.NET expects those types to have a default constructor.

Instead of doing constructor injection, there are alternatives. The simplest thing to do is to fall back to property injection and initialize the page in the constructor.

```
using System;
using System.ComponentModel.Composition;
using System.Linq;
using System.Reflection;
using System.Web;
using System.Web.Compilation;
using System.Web.UI;
using Microsoft.Web.Infrastructure.DynamicModuleHelper;
using SimpleInjector;
using SimpleInjector.Advanced;

[assembly: PreApplicationStartMethod(
    typeof(MyWebApplication.PageInitializerModule),
    "Initialize")]

namespace MyWebApplication
{
    public sealed class PageInitializerModule : IHttpModule {
        public static void Initialize() {
            DynamicModuleUtility.RegisterModule(typeof(PageInitializerModule));
        }

        void IHttpModule.Init(HttpApplication app) {
```

```
            app.PreRequestHandlerExecute += (sender, e) => {
                var handler = app.Context.CurrentHandler;
                if (handler != null) {
                    string name = handler.GetType().Assembly.FullName;
                    if (!name.StartsWith("System.Web") &&
                        !name.StartsWith("Microsoft")) {
                        Global.InitializeHandler(handler);
                    }
                }
            };
        }

        void IHttpModule.Dispose() { }
    }

    public class Global : HttpApplication {
        private static Container container;

        public static void InitializeHandler(IHttpHandler handler) {
            container.GetRegistration(handler.GetType(), true).Registration
                .InitializeInstance(handler);
        }

        protected void Application_Start(object sender, EventArgs e) {
            Bootstrap();
        }

        private static void Bootstrap() {
            // 1. Create a new Simple Injector container.
            var container = new Container();

            // Register a custom PropertySelectionBehavior to enable property␣
↪injection.
            container.Options.PropertySelectionBehavior =
                new ImportAttributePropertySelectionBehavior();

            // 2. Configure the container (register)
            container.Register<IUserRepository, SqlUserRepository>();
            container.RegisterPerWebRequest<IUserContext, AspNetUserContext>();

            // Register your Page classes to allow them to be verified and diagnosed.
            RegisterWebPages(container);

            // 3. Store the container for use by Page classes.
            Global.container = container;

            // 3. Verify the container's configuration.
            container.Verify();
        }

        private static void RegisterWebPages(Container container) {
            var pageTypes =
                from assembly in BuildManager.GetReferencedAssemblies().Cast<Assembly>
↪()
                where !assembly.IsDynamic
                where !assembly.GlobalAssemblyCache
                from type in assembly.GetExportedTypes()
                where type.IsSubclassOf(typeof(Page))
```

```
                where !type.IsAbstract && !type.IsGenericType
                select type;

            foreach (Type type in pageTypes) {
                var reg = Lifestyle.Transient.CreateRegistration(type, container);
                reg.SuppressDiagnosticWarning(
                    DiagnosticType.DisposableTransientComponent,
                    "ASP.NET creates and disposes page classes for us.");
                container.AddRegistration(type, reg);
            }
        }

        class ImportAttributePropertySelectionBehavior : IPropertySelectionBehavior {
            public bool SelectProperty(Type serviceType, PropertyInfo propertyInfo) {
                // Makes use of the System.ComponentModel.Composition assembly
                return typeof(Page).IsAssignableFrom(serviceType) &&
                    propertyInfo.GetCustomAttributes<ImportAttribute>().Any();
            }
        }
    }
}
```

With this code in place, we can now write our page classes as follows:

```
public partial class Default : Page {
    [Import] public IUserRepository UserRepository { get; set; }
    [Import] public IUserContext UserContext { get; set; }

    protected void Page_Load(object sender, EventArgs e) {
        if (this.UserContext.IsAdministrator) {
            this.UserRepository.DoSomeStuff();
        }
    }
}
```

OWIN Integration Guide

Basic setup

To allow scoped instances to be resolved during an OWIN request, the following registration needs to be added to the *IAppBuilder* instance:

```
// You'll need to include the following namespaces
using Owin;
using SimpleInjector;
using SimpleInjector.Lifestyles;

public void Configuration(IAppBuilder app) {
    app.Use(async (context, next) => {
        using (AsyncScopedLifestyle.BeginScope(container)) {
            await next();
        }
    });
}
```

Scoped instances need to be registered with the *AsyncScopedLifestyle* lifestyle:

```
var container = new Container();
container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

container.Register<IUnitOfWork, MyUnitOfWork>(Lifestyle.Scoped);
```

Extra features

Besides this basic integration, other tips and tricks can be applied to integrate Simple Injector with OWIN.

Getting the current request's IOwinContext

When working with OWIN you will occasionally find yourself wanting access to the current *IOwinContext*. Retrieving the current *IOwinContext* is easy as using the following code snippet:

```
public interface IOwinContextAccessor {
    IOwinContext CurrentContext { get; }
}

public class CallContextOwinContextAccessor : IOwinContextAccessor {
    public static AsyncLocal<IOwinContext> OwinContext = new AsyncLocal<IOwinContext>
↪();
    public IOwinContext CurrentContext => OwinContext.Value;
}
```

The code snippet above defines an *IOwinContextAccessor* and an implementation. Consumers can depend on the *IOwinContextAccessor* and can call its *CurrentContext* property to get the request's current *IOwinContext*.

The following code snippet can be used to register this *IOwinContextAccessor* and its implementation:

```
app.Use(async (context, next) => {
    CallContextOwinContextAccessor.OwinContext.Value = context;
    await next();
});

container.RegisterSingleton<IOwinContextAccessor>(new␣
↪CallContextOwinContextAccessor());
```

Windows Forms Integration Guide

Doing dependency injection in Windows Forms is easy, since Windows Forms does not lay any constraints on the constructors of your Form classes. You can therefore simply use constructor injection in your form classes and let the container resolve them.

The following code snippet is an example of how to register Simple Injector container in the *Program* class:

```
using System;
using System.Windows.Forms;
using SimpleInjector;

static class Program {
    private static Container container;
```

```
    [STAThread]
    static void Main() {

        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Bootstrap();
        Application.Run(container.GetInstance<Form1>());
    }

    private static void Bootstrap() {
        // Create the container as usual.
        container = new Container();

        // Register your types, for instance:
        container.Register<IUserRepository, SqlUserRepository>(Lifestyle.Singleton);
        container.Register<IUserContext, WinFormsUserContext>();
        container.Register<Form1>();

        // Optionally verify the container.
        container.Verify();
    }
}
```

With this code in place, we can now write our *Form* classes as follows:

```
public partial class Form1 : Form {
    private readonly IUserRepository userRepository;
    private readonly IUserContext userContext;

    public Form1(IUserRepository userRepository, IUserContext userContext) {
        this.userRepository = userRepository;
        this.userContext = userContext;

        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e) {
        if (this.userContext.IsAdministrator) {
            this.userRepository.ControlSomeStuff();
        }
    }
}
```

Note: It is not possible to use *Constructor Injection* in *User Controls*. *User Controls* are required to have a default constructor. Instead, pass on dependencies to your *User Controls* using *Method Injection*.

WCF Integration Guide

The Simple Injector WCF Integration NuGet Package[44] allows WCF services to be resolved by the container, which enables constructor injection.

After installing this NuGet package, it must be initialized in the start-up path of the application by calling the **SimpleInjectorServiceHostFactory.SetContainer** method:

[44] https://nuget.org/packages/SimpleInjector.Integration.Wcf

```
protected void Application_Start(object sender, EventArgs e) {
    // Create the container as usual.
    var container = new Container();
    container.Options.DefaultScopedLifestyle = new WcfOperationLifestyle();

    // Register your types, for instance:
    container.Register<IUserRepository, SqlUserRepository>();
    container.Register<IUnitOfWork, EfUnitOfWork>(Lifestyle.Scoped);

    // Register the container to the SimpleInjectorServiceHostFactory.
    SimpleInjectorServiceHostFactory.SetContainer(container);
}
```

Warning: Instead of what the name of the **WcfOperationLifestyle** class seems to imply, components that are registered with this lifestyle might actually outlive a single WCF operation. This behavior depends on how the WCF service class is configured. WCF is in control of the lifetime of the service class and contains three lifetime types as defined by the InstanceContextMode enumeration[45]. Components that are registered *PerWcfOperation* live as long as the WCF service class they are injected into.

For each service class, you should supply a factory attribute in the .SVC file of each service class. For instance:

```
<%@ ServiceHost
    Service="UserService"
    CodeBehind="UserService.svc.cs"
    Factory="SimpleInjector.Integration.Wcf.SimpleInjectorServiceHostFactory,
        SimpleInjector.Integration.Wcf"
%>
```

Note: Instead of having a WCF service layer consisting of many service classes and methods, consider a design that consists of just a single service class with a single method as explained in this article[46]. A design where operations are communicated through messages allows the creation of highly maintainable WCF services. With such a design, this integration package will be redundant.

WAS Hosting and Non-HTTP Activation

When hosting WCF Services in WAS (Windows Activation Service), you are not given an opportunity to build your container in the Application_Start event defined in your Global.asax because WAS doesn't use the standard ASP.NET pipeline. A workaround for this is to move the container initialization to a static constructor:

```
public static class Bootstrapper {
    public static readonly Container Container;

    static Bootstrapper() {
        var container = new Container();
        container.Options.DefaultScopedLifestyle = new WcfOperationLifestyle();

        // register all your components with the container here:
        // container.Register<IService1, Service1>()
        // container.Register<IDataContext, DataContext>(Lifestyle.Scoped);

        container.Verify();

        Container = container;
```

[45] https://msdn.microsoft.com/en-us/library/system.servicemodel.instancecontextmode.aspx
[46] http://www.cuttingedge.it/blogs/steven/pivot/entry.php?id=95

```
    }
}
```

Your custom *ServiceHostFactory* can now use the static **Bootstrapper.Container** field:

```
public class WcfServiceFactory : SimpleInjectorServiceHostFactory {
    protected override ServiceHost CreateServiceHost(Type serviceType,
        Uri[] baseAddresses) {
        return new SimpleInjectorServiceHost(
            Bootstrapper.Container,
            serviceType,
            baseAddresses);
    }
}
```

Optionally, you can apply your custom service behaviors and contract behaviors to the service host:

```
public class WcfServiceFactory : SimpleInjectorServiceHostFactory {
    protected override ServiceHost CreateServiceHost(Type serviceType,
        Uri[] baseAddresses) {
        var host = new SimpleInjectorServiceHost(
            Bootstrapper.Container,
            serviceType,
            baseAddresses);

        // This is all optional
        this.ApplyServiceBehaviors(host);
        this.ApplyContractBehaviors(host);

        return host;
    }

    private void ApplyServiceBehaviors(ServiceHost host) {
        foreach (var behavior in this.container.GetAllInstances<IServiceBehavior>()) {
            host.Description.Behaviors.Add(behavior);
        }
    }

    private void ApplyContractBehaviors(SimpleInjectorServiceHost host) {
        foreach (var behavior in this.container.GetAllInstances<IContractBehavior>())
↪{
            foreach (var contract in host.GetImplementedContracts()) {
                contract.Behaviors.Add(behavior);
            }
        }
    }
}
```

For each service class, you should supply a factory attribute in the .SVC file of each service class. Assuming the customly defined factory is defined in the *MyComp.MyWcfService.Common* namespace of the *MyComp.MyWcfService* assembly, the markup would be the following:

```
<%@ ServiceHost
    Service="UserService"
    CodeBehind="UserService.svc.cs"
    Factory="MyComp.MyWcfService.Common.WcfServiceFactory, MyComp.MyWcfService"
%>
```

Windows Presentation Foundation Integration Guide

Simple Injector can build up *Window* classes with their dependencies. Use the following steps as a how-to guide:

Step 1:

Change the App.xaml markup by removing the *StartUri* property:

```
<Application x:Class="SimpleInjectorWPF.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <!-- Remove the StartupUri property, start the application from a static Main -->
    <Application.Resources>
    </Application.Resources>
</Application>
```

Step 2:

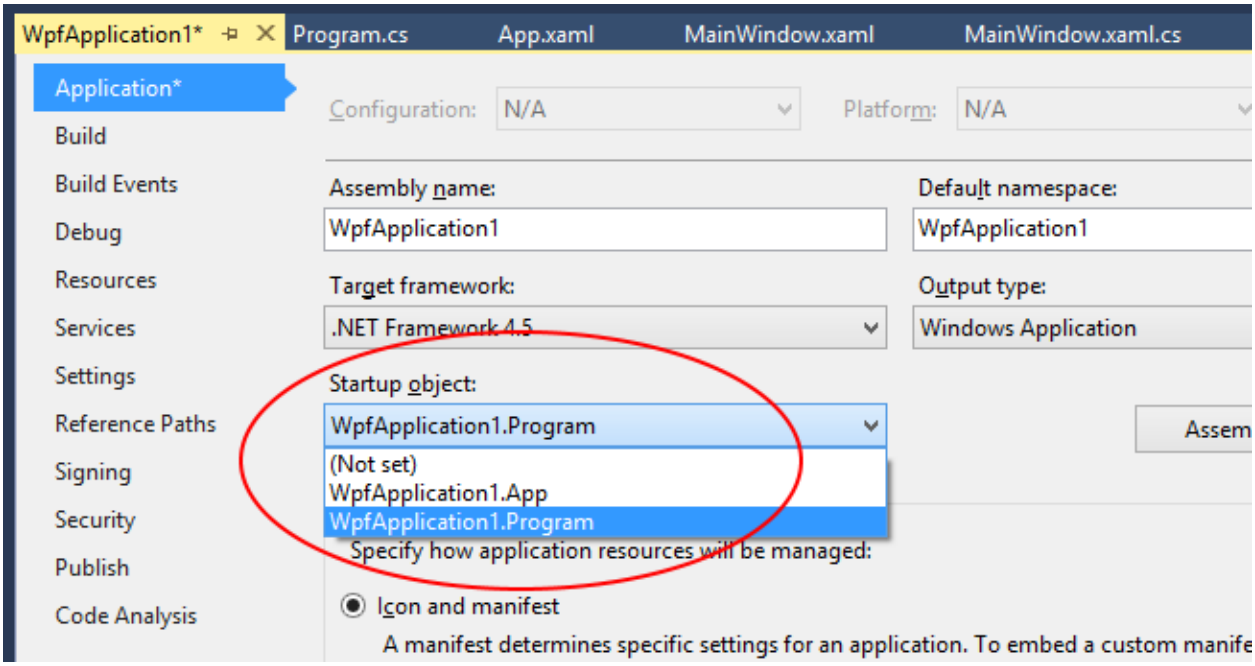
Add a *Program.cs* file to your project to be the new entry point for the application:

```
using System;
using System.Windows;
using SimpleInjector;

static class Program
{
    [STAThread]
    static void Main() {
        var container = Bootstrap();

        // Any additional other configuration, e.g. of your desired MVVM toolkit.

        RunApplication(container);
    }

    private static Container Bootstrap() {
        // Create the container as usual.
        var container = new Container();

        // Register your types, for instance:
        container.Register<IQueryProcessor, QueryProcessor>(Lifestyle.Singleton);
        container.Register<IUserContext, WpfUserContext>();

        // Register your windows and view models:
        container.Register<MainWindow>();
        container.Register<MainWindowViewModel>();

        container.Verify();

        return container;
    }

    private static void RunApplication(Container container) {
        try {
            var app = new App();
```

```
            var mainWindow = container.GetInstance<MainWindow>();
            app.Run(mainWindow);
        } catch (Exception ex) {
            //Log the exception and exit
        }
    }
}
```

Step 3:

Change the 'Startup object' in the properties of your project to be the newly created *Program* class:



Usage

Constructor injection can now be used in any window (e.g. *MainWindow*) and view model:

```
using System.Windows;

public partial class MainWindow : Window {
    public MainWindow(MainWindowViewModel viewModel) {
        InitializeComponent();

        // Assign to the data context so binding can be used.
        base.DataContext = viewModel;
    }
}

public class MainWindowViewModel {
    private readonly IQueryProcessor queryProcessor;
    private readonly IUserContext userContext;

    public MainWindowViewModel(IQueryProcessor queryProcessor,
```

```
        IUserContext userContext) {
        this.queryProcessor = queryProcessor;
        this.userContext = userContext;
    }

    public IEnumerable<IUser> Users => this.queryProcessor.Execute(new GetAllUsers());
}
```

Silverlight Integration Guide

Configuring Simple Injector to build up *Page* classes with their dependencies is simply a matter of adding a *Bootstrap()* method to the *App* class and resolving the *MainPage* during *Application_Startup*:

```
using System;
using System.Windows;
using SimpleInjector;

public partial class App : Application
{
    public App() {
        this.Startup += this.Application_Startup;
        this.Exit += this.Application_Exit;
        this.UnhandledException += this.Application_UnhandledException;

        InitializeComponent();
    }

    private void Application_Startup(object sender, StartupEventArgs e) {
        var container = Bootstrap();

        this.RootVisual = container.GetInstance<MainPage>();
    }

    public static Container Bootstrap() {
        // Create the container as usual.
        var container = new Container();

        // Register your pages and view models:
        container.Register<MainPage>();
        container.Register<MainPageViewModel>();

        container.Verify();

        return container;
    }

    // Rest of the code
}
```

Usage

Constructor injection can now be used in any pages (e.g. *MainPage*) and view models:

```
using System.Windows;

public partial class MainPage : Page {
    public MainPage(MainPageViewModel viewModel) {
        InitializeComponent();

        // Assign to the data context so binding can be used.
        base.DataContext = viewModel;
    }
}

public class MainPageViewModel {
    private readonly IQueryProcessor queryProcessor;
    private readonly IUserContext userContext;

    public MainPageViewModel(IQueryProcessor queryProcessor,
        IUserContext userContext) {
        this.queryProcessor = queryProcessor;
        this.userContext = userContext;
    }

    public IEnumerable<IUser> Users => this.queryProcessor.Execute(new GetAllUsers());
}
```

Other Technologies

Besides integration with standard .NET technologies, Simple Injector can be integrated with a wide range of other technologies. Here are a few links to help you get started quickly:

- RavenDB[47]
- SignalR[48]
- Fluent Validations[49]
- PetaPoco[50]
- Quartz.NET[51]
- Membus[52]
- Web Forms MVP[53]
- Nancy[54]
- Castle DynamicProxy Interception[55]
- ASP.NET Identity Framework[56]

[47] https://stackoverflow.com/questions/10940988/how-to-configure-simple-injector-ioc-to-use-ravendb
[48] https://stackoverflow.com/questions/10555791/using-simpleinjector-with-signalr
[49] https://stackoverflow.com/questions/9984144/what-is-the-correct-way-to-register-fluentvalidation-with-simpleinjector
[50] https://simpleinjector.codeplex.com/discussions/283850
[51] https://stackoverflow.com/questions/14562176/constructor-injection-with-quartz-net-and-simple-injector
[52] https://stackoverflow.com/questions/16123641/membus-and-ioc-simpleinjector-wiring-command-handlers-automatically-by-interfa
[53] https://stackoverflow.com/questions/15123515/pass-runtime-value-to-constructor-using-simpleinjector
[54] https://github.com/NancyFx/Nancy/issues/1227#issuecomment-163279040
[55] https://stackoverflow.com/questions/24513530/using-simple-injector-with-castle-proxy-interceptor
[56] https://simpleinjector.codeplex.com/discussions/564822

- Drum[57]

Patterns

- Unit of Work pattern[58]
- Multi-tenant applications[59]
- Auto-Mocking container[60]

[57] https://stackoverflow.com/questions/26661621/how-to-register-drum-urimaker-using-simple-injector
[58] https://stackoverflow.com/questions/10585478
[59] https://simpleinjector.codeplex.com/discussions/434951
[60] https://github.com/simpleinjector/SimpleInjector/issues/290#issuecomment-243244930

CHAPTER 5

Diagnostic Services

The **Diagnostic Services** allow you to analyze the container's configuration to search for common configuration mistakes. Simple Injector knows about several common issues that might arise when composing the object graph, and can expose this information to the developer via its diagnostics API. Currently, Simple Injector classifies the issues in two severities: Warnings and Informational messages.

Information messages are are hints of things might want to look into, such as possible Single Responsibility Principle violations. Warnings on the other hand are strong indications that there is a problem with your configuration. See them as the Simple Injector equivalent of C# warnings: Your code will compile (as in: we will be able to resolve your objects), but you rather want to fix those warnings.

Supported Warnings

Diagnostic Warning - Lifestyle Mismatches

Severity

Warning

Cause

The component depends on a service with a lifestyle that is shorter than that of the component.

Warning Description

In general, components should only depend on other components that are configured to live at least as long. In other words, it is safe for a transient component to depend on a singleton, but not the other way around. Since components store a reference to their dependencies in (private) instance fields, those dependencies are kept alive for the lifetime of that component. This means that dependencies that are configured with a shorter lifetime than their consumer, accidentally live longer than intended. This can lead to all sorts of bugs, such as hard to debug multi-threading issues.

The Diagnostic Services detect this kind of misconfiguration and report it. The container will be able to compare all built-in lifestyles (and sometimes even custom lifestyles). Here is an overview of the built-in lifestyles ordered by their length:

- *Transient*
- *Scoped*
- *Singleton*

Note: This kind of error is also known as Captive Dependency[61].

Note: Lifestyle mismatches are such a common source of bugs, that the container always checks for mismatches the first time a component is resolved, no matter whether you call *Verify()* or not. This behavior can be suppressed by setting the **Container.Options.SuppressLifestyleMismatchVerification** property, but you are advised to keep the default settings.

How to Fix Violations

There are multiple ways to fix this violation:

- Change the lifestyle of the component to a lifestyle that is as short or shorter than that of the dependency.
- Change the lifestyle of the dependency to a lifestyle as long or longer than that of the component.
- Instead of injecting the dependency, inject a factory for the creation of that dependency and call that factory every time an instance is required.

When to Ignore Warnings

Do not ignore these warnings. False positives for this warning are rare and even when they occur, the registration or the application design can always be changed in a way that the warning disappears.

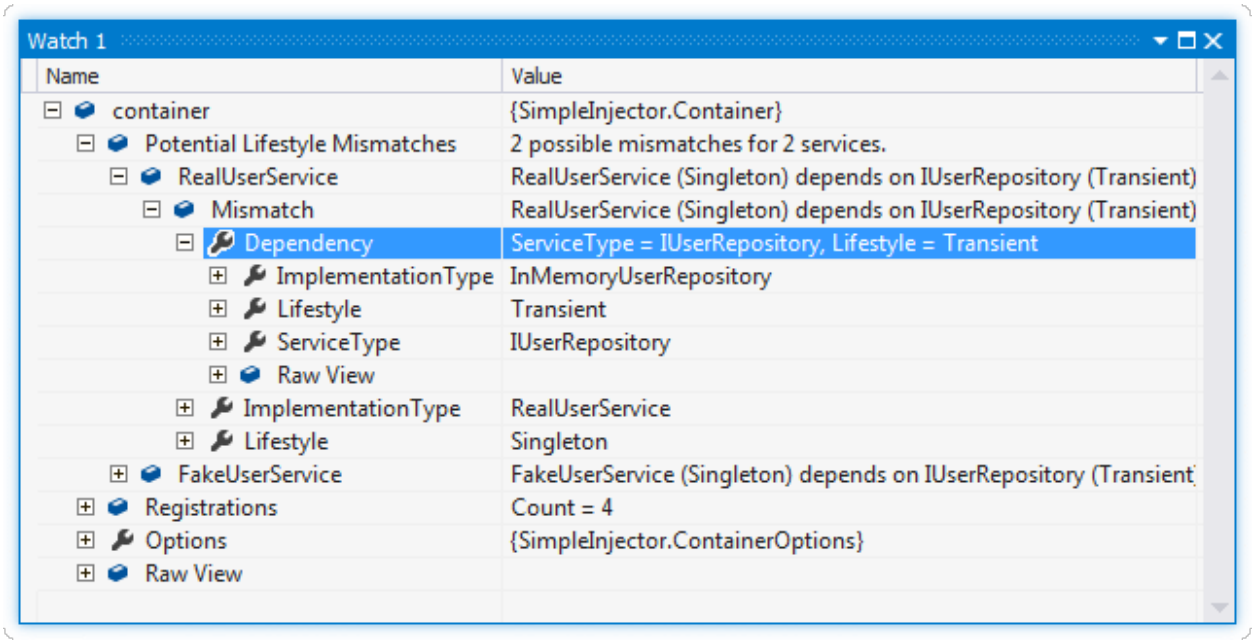
Example

The following example shows a configuration that will trigger the warning:

```
var container = new Container();

container.Register<IUserRepository, InMemoryUserRepository>(Lifestyle.Transient);

// RealUserService depends on IUserRepository
container.Register<RealUserService>(Lifestyle.Singleton);

// FakeUserService depends on IUserRepository
container.Register<FakeUserService>(Lifestyle.Singleton);

container.Verify();
```

The *RealUserService* component is registered as **Singleton** but it depends on *IUserRepository* which is configured with the shorter **Transient** lifestyle. Below is an image that shows the output for this configuration in a watch window. The watch window shows two mismatches and one of the warnings is unfolded.

[61] http://blog.ploeh.dk/2014/06/02/captive-dependency/



The following example shows how to query the Diagnostic API for Lifetime Mismatches:

```
// using SimpleInjector.Diagnostics;

var container = /* get verified container */;

var results = Analyzer.Analyze(container)
    .OfType<LifestyleMismatchDiagnosticResult>();

foreach (var result in results) {
    Console.WriteLine(result.Description);
    Console.WriteLine("Lifestyle of service: " +
        result.Relationship.Lifestyle.Name);

    Console.WriteLine("Lifestyle of service's dependency: " +
        result.Relationship.Dependency.Lifestyle.Name);
}
```

What about Hybrid lifestyles?

A *Hybrid lifestyle* is a mix between two or more other lifestyles. Here is an example of a custom lifestyle that mixes the **Transient** and **Singleton** lifestyles together:

```
var hybrid = Lifestyle.CreateHybrid(
    lifestyleSelector: () => someCondition,
    trueLifestyle: Lifestyle.Transient,
    falseLifestyle: Lifestyle.Singleton);
```

Note that this example is quite bizarre, since it is a very unlikely combination of lifestyles to mix together, but it serves us well for the purpose of this explanation.

As explained, components should only depend on equal length or longer lived components. But how long does a component with this hybrid lifestyle live? For components that are configured with the lifestyle defined above, it depends on the implementation of *someCondition*. But without taking this condition into consideration, we can say

that it will at most live as long as the longest wrapped lifestyle (Singleton in this case) and at least live as long as shortest wrapped lifestyle (in this case Transient).

From the Diagnostic Services' perspective, a component can only safely depend on a hybrid lifestyled service if the consuming component's lifestyle is shorter than or equal the shortest lifestyle the hybrid is composed of. On the other hand, a hybrid lifestyled component can only safely depend on another service when the longest lifestyle of the hybrid is shorter than or equal to the lifestyle of the dependency. Thus, when a relationship between a component and its dependency is evaluated by the Diagnostic Services, the **longest** lifestyle is used in the comparison when the hybrid is part of the consuming component, and the **shortest** lifestyle is used when the hybrid is part of the dependency.

This does imply that two components with the same hybrid lifestyle can't safely depend on each other. This is true since in theory the supplied predicate could change results in each call. In practice however, those components would usually be able safely relate, since it is normally unlikely that the predicate changes lifestyles within a single object graph. This is an exception the Diagnostic Services can make pretty safely. From the Diagnostic Services' perspective, components can safely be related when both share the exact same lifestyle instance and no warning will be displayed in this case. This does mean however, that you should be very careful using predicates that change the lifestyle during the object graph.

Diagnostic Warning - Short Circuited Dependencies

Severity

Warning

Cause

The component depends on an unregistered concrete type where there exists a registration that uses this concrete type as its implementation.

Warning Description

This warning signals the possible use of a short circuited dependency in a component. A short circuited dependency is:

- a concrete type
- that is not registered as itself (i.e. not registered as **Register<TConcrete>** or its non-generic equivalent)
- and is referenced by another component (most likely using a constructor argument)
- and exists as *TImplementation* in an explicitly made **Register<TService, TImplementation>()** registration (or its non-generic equivalent)

When a component depends on a short circuited dependency, the application might be wired incorrectly because the flagged component gets a different instance of that concrete type than other components in the application will get. This can result in incorrect behavior.

How to Fix Violations

Let the component depend on the abstraction described in the warning message instead of depending directly on the concrete type. If the warning is a false positive and the component (and all other components that depend directly on this concrete type) should indeed get a transient instance of that concrete type, register the concrete type explicitly in the container using the transient lifestyle.

When to Ignore Warnings

Do not ignore these warnings. False positives for this warning are rare and even when they occur, the registration or the application design can always be changed or the concrete type can be registered explicitly in the container.

Example

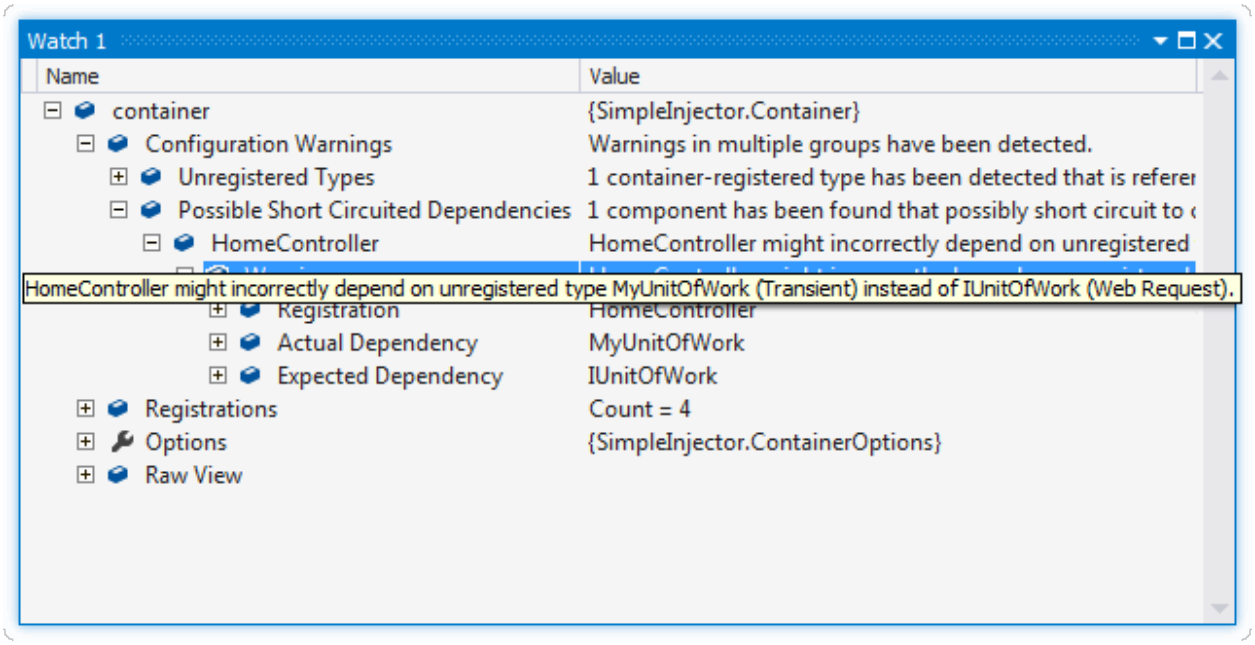
```
container.Register<IUnitOfWork, MyUnitOfWork>(Lifestyle.Scoped);
container.Register<HomeController>();

// Definition of HomeController
public class HomeController : Controller {
    private readonly MyUnitOfWork uow;

    public HomeController(MyUnitOfWork uow) {
        this.uow = uow;
    }
}
```

In this example *HomeController* depends on *MyUnitOfWork*. *MyUnitOfWork* however is not registered explicitly, but *IUnitOfWork* is. Furthermore *IUnitOfWork* is registered with a scoped lifestyle. However, since *MyUnitOfWork* is a concrete unregistered type, the container will create it on your behalf with the **Transient** lifestyle. This will typically be a problem, since during a request, the *HomeController* will get a different instance than other types that depend on *IUnitOfWork* while the intended use of *IUnitOfWork* is to have a single instance per web request.

For Unit of Work implementations this is typically a problem, since the unit of work defines an atomic operation and creating multiple instances of such a unit of work in a single web request means that the work is split up in multiple (database) transactions (breaking consistency) or could result in part of the work not being committed at all.

The *MyUnitOfWork* type is called 'short circuited' because *HomeController* skips the *IUnitOfWork* dependency and directly depends on *MyUnitOfWork*. In other words, *HomeController* short circuits to *MyUnitOfWork*.

Here is an example of a short circuited dependency in the watch window:



The following example shows how to query the Diagnostic API for Short Circuited Dependencies:

```
// using SimpleInjector.Diagnostics;

var container = /* get verified container */;

var results = Analyzer.Analyze(container)
    .OfType<ShortCircuitedDependencyDiagnosticResult>();

foreach (var result in results) {
    Console.WriteLine(result.Description);
    Console.WriteLine(
        "Lifestyle of service with the short circuited dependency: " +
        result.Relationship.Lifestyle.Name);

    Console.WriteLine("One of the following types was expected instead:");
    foreach (var expected in result.ExpectedDependencies) {
        Console.WriteLine("-" + expected.ServiceType.FullName);
    }
}
```

Diagnostic Warning - Torn Lifestyle

Severity

Warning

Cause

Multiple registrations with the same lifestyle map to the same component.

Warning Description

When multiple **Registration** instances with the same lifestyle map to the same component, the component is said to have a torn lifestyle. The component is considered torn because each **Registration** instance will have its own cache of the given component, which can potentially result in multiple instances of the component within a single *scope*. When the registrations are torn the application may be wired incorrectly which could lead to unexpected behavior.

Note: With the introduction of Simple Injector 4, the container will prevent the creation of multiple **Registration** instances for the same concrete type in most cases. This warning type should therefore be extremely rare. Torn lifestyles will only happen when a custom lifestyle circumvents the caching behavior of the **Lifestyle.CreateRegistration** overloads.

How to Fix Violations

Make sure the creation of **Registration** instances of your custom lifestyle goes through the **Lifestyle.CreateRegistration** method instead directly to **Lifestyle.CreateRegistrationCore**.

When to Ignore Warnings

This warning most likely signals a bug in a custom **Lifestyle** implementation, so warnings should typically not be ignored.

The warning can be suppressed on a per-registration basis as follows:

```
var fooRegistration = container.GetRegistration(typeof(IFoo)).Registration;
var barRegistration = container.GetRegistration(typeof(IBar)).Registration;
fooRegistration.SuppressDiagnosticWarning(DiagnosticType.TornLifestyle);
barRegistration.SuppressDiagnosticWarning(DiagnosticType.TornLifestyle);
```

Diagnostic Warning - Ambiguous Lifestyles

Severity

Warning

Cause

Multiple registrations with the different lifestyles map to the same component.

Warning Description

When multiple registrations with a different lifestyle map to the same component, the component is said to have ambiguous lifestyles. Having one single component with multiple lifestyles will cause instances of that component to be cached in different ways and this can lead to behavior that you might not expect.

For instance, having a single component that is registered both as Singleton and as Transient hardly ever makes sense, because the singleton registration implies that it is thread-safe, while the transient registration means that it, or one of its dependencies, is not thread-safe and should not be reused. One of the registrations of this component will likely be wrong.

How to Fix Violations

Make all registrations with the same lifestyle.

In case you really intended to have this single component to be registered with two different lifestyles, this is a clear indication that the component should actually be split into multiple smaller components, each with their specific lifestyle.

When to Ignore Warnings

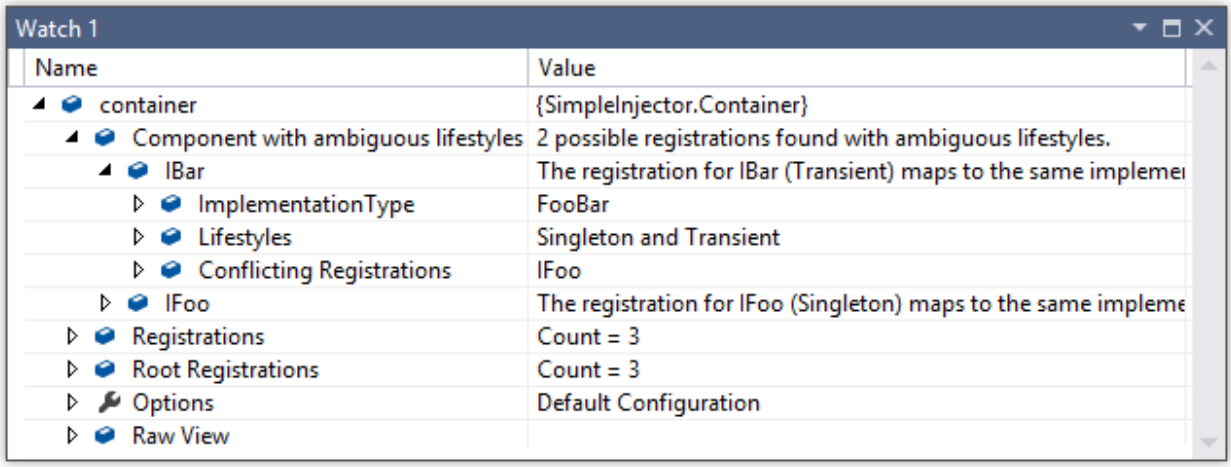
Do not ignore these warnings. False positives for this warning are rare and even when they occur, the registration or the application design can always be changed in a way that the warning disappears.

Example

The following example shows a configuration that will trigger the warning:

```
var container = new Container();

container.Register<IFoo, FooBar>(Lifestyle.Transient);
container.Register<IBar, FooBar>(Lifestyle.Singleton);

container.Verify();
```

The *FooBar* component is registered once as **Singleton** for *IFoo* and once as **Transient** for *IBar* (assuming that *FooBar* implements both *IFoo* and *IBar*). Below is an image that shows the output for this configuration in a watch window. The watch window shows two mismatches and one of the warnings is unfolded.



The issue can be fixed as follows:

```
var container = new Container();

container.Register<IFoo, FooBar>(Lifestyle.Singleton);
container.Register<IBar, FooBar>(Lifestyle.Singleton);

container.Verify();
```

Another way to fix this issue is by splitting *FooBar* into multiple smaller components:

```
var container = new Container();

// New component with singleton lifestyle
container.Register<IFooBarCommon, FooBarCommon>(Lifestyle.Singleton);

// Old component split into two, both depending on IFooBarCommon.
container.Register<IFoo, Foo>(Lifestyle.Transient);
container.Register<IBar, Bar>(Lifestyle.Singleton);

container.Verify();
```

The following example shows how to query the Diagnostic API for Torn Lifestyles:

```
// using SimpleInjector.Diagnostics;

var container = /* get verified container */;

var results = Analyzer.Analyze(container).OfType<AmbiguousLifestylesDiagnosticResult>
↪();

foreach (var result in results) {
    Console.WriteLine(result.Description);
    Console.WriteLine("Component name: " + result.ImplementationType.Name);
    Console.WriteLine("Lifestyles of component: " +
        string.Join(", ", result.Lifestyles.Select(l => l.Name)));
    Console.WriteLine("Conflicting registrations: " +
        string.Join(", ", result.ConflictingRegistrations.Select(
```

```
            r => r.ServiceType.Name)));
}
```

Diagnostic Warning - Disposable Transient Components

Severity

Warning

Cause

A registration has been made with the Transient lifestyle for a component that implements IDisposable.

Warning Description

A component that implements *IDisposable* would usually need deterministic clean-up but Simple Injector does not implicitly track and dispose components registered with the transient lifestyle.

How to Fix Violations

Register the component with the *scoped lifestyle* that is appropriate for the application you are working on. Scoped lifestyles ensure *Dispose* is called when an active scope ends.

When to Ignore Warnings

This warning can safely be ignored when:

- *Dispose* is called by the application code
- some manual registration ensures disposal
- a framework (such as ASP.NET) guarantees disposal of the component
- not disposing is not an issue.

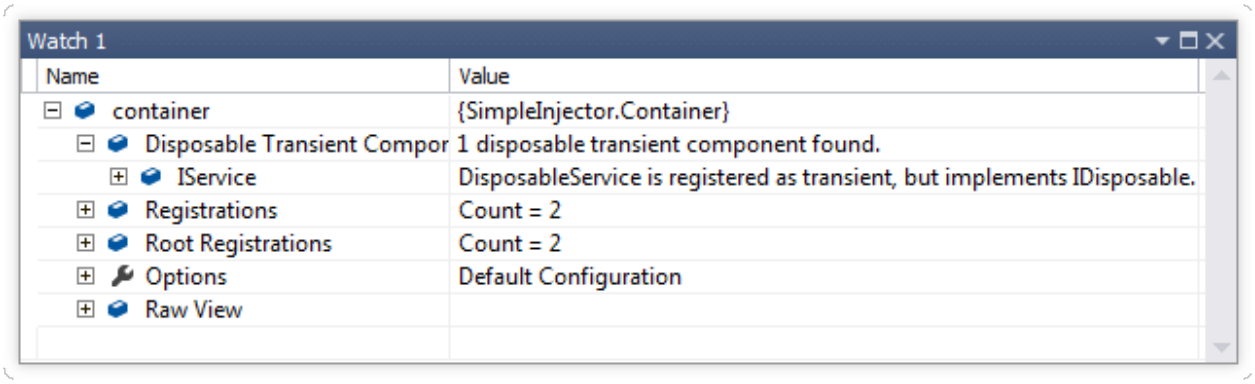
The warning can be suppressed on a per-registration basis as follows:

```
Registration registration = container.GetRegistration(typeof(IService)).Registration;

registration.SuppressDiagnosticWarning(DiagnosticType.DisposableTransientComponent,
    "Reason of suppression");
```

Example

The following example shows a configuration that will trigger the warning:

```
var container = new Container();

// DisposableService implements IDisposable
container.Register<IService, DisposableService>(Lifestyle.Transient);

container.Verify();
```



The issue can be fixed as follows:

```
var container = new Container();
// Select the scoped lifestyle that is appropriate for the application
// you are building. For instance:
container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

// DisposableService implements IDisposable
container.Register<IService, DisposableService>(Lifestyle.Scoped);

container.Verify();
```

The following example shows how to query the Diagnostic API for Disposable Transient Components:

```
// using SimpleInjector.Diagnostics;

var container = /* get verified container */;

var results = Analyzer.Analyze(container)
    .OfType<DisposableTransientComponentDiagnosticResult>();

foreach (var result in results) {
    Console.WriteLine(result.Description);
}
```

Optionally you can let transient services dispose when a scope ends. Here's an example of an extension method that allows registering transient instances that are disposed when the specified scope ends:

```
public static void RegisterDisposableTransient<TService, TImplementation>(
    this Container c)
    where TImplementation: class, IDisposable, TService
    where TService : class
{
    var scoped = Lifestyle.Scoped;
    var r = Lifestyle.Transient.CreateRegistration<TService, TImplementation>(c);
    r.SuppressDiagnosticWarning(DiagnosticType.DisposableTransientComponent, "ignore
↪");
    c.AddRegistration(typeof(TService), r);
    c.RegisterInitializer<TImplementation>(o => scoped.RegisterForDisposal(c, o));
}
```

The following code snippet show the usage of this extension method:

```
var container = new Container();
container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();
```

```
container.RegisterDisposableTransient<IService, ServiceImpl>();
```

This ensures that each time a *ServiceImpl* is created by the container, it is registered for disposal when the scope - a web request in this case - ends. This can of course lead to the creation and disposal of multiple *ServiceImpl* instances during a single request.

Note: To be able to dispose an instance, the **RegisterForDisposal** will store the reference to that instance in the scope. This means that the instance will be kept alive for the lifetime of that scope.

Warning: Be careful to not register any services for disposal that will outlive that scope (such as services registered as singleton), since a service cannot be used once it has been disposed. This would typically result in *ObjectDisposedExceptions* and this will cause your application to break.

Supported Information messages

Diagnostic Warning - Potential Single Responsibility Violations

Severity

Information

Cause

The component depends on too many services.

Warning Description

Psychological studies show that the human mind has difficulty dealing with more than seven things at once. This is related to the concept of High Fan In - Low Fan Out[62]. Lowering the number of dependencies (fan out) that a class has can therefore reduce complexity and increase maintainability of such class.

So in general, components should only depend on a few other components. When a component depends on many other components (usually caused by constructor over-injection), it might indicate that the component has too many responsibilities. In other words it might be a sign that the component violates the Single Responsibility Principle[63] (SRP). Violations of the SRP will often lead to maintainability issues later on in the application lifecycle.

The general consensus is that a constructor with more than 4 or 5 dependencies is a code smell. To prevent too many false positives, the threshold for the Diagnostic Services is 7 dependencies, so you'll start to see warnings on types with 8 or more dependencies.

How to Fix Violations

The article Dealing with constructor over-injection[64], Mark Seemann goes into detail how to about fixing the root cause of constructor over-injection. Two solutions in particular come to mind.

Note that moving dependencies out of the constructor and into properties might solve the constructor over-injection code smell, but does not solve a violation of the SRP, since the number of dependencies doesn't decrease.

[62] http://it.toolbox.com/blogs/enterprise-solutions/design-principles-fanin-vs-fanout-16088
[63] https://en.wikipedia.org/wiki/Single_responsibility_principle
[64] https://deals.manningpublications.com/DependencyInjectioninNET.pdf

Moving those properties to a base class also doesn't solve the SRP violation. Often derived types will still use the dependencies of the base class making them still violating the SRP and even if they don't, the base class itself will probably violate the SRP or have a high fan out.

Those base classes will often just be helpers to implement all kinds of cross-cutting concerns. Instead of using base classes, a better way to implementing cross-cutting concerns is through *decorators*.

When to Ignore Warnings

This warning can safely be ignored when the type in question does not violate the SRP and the number of dependencies is stable (does not change often).

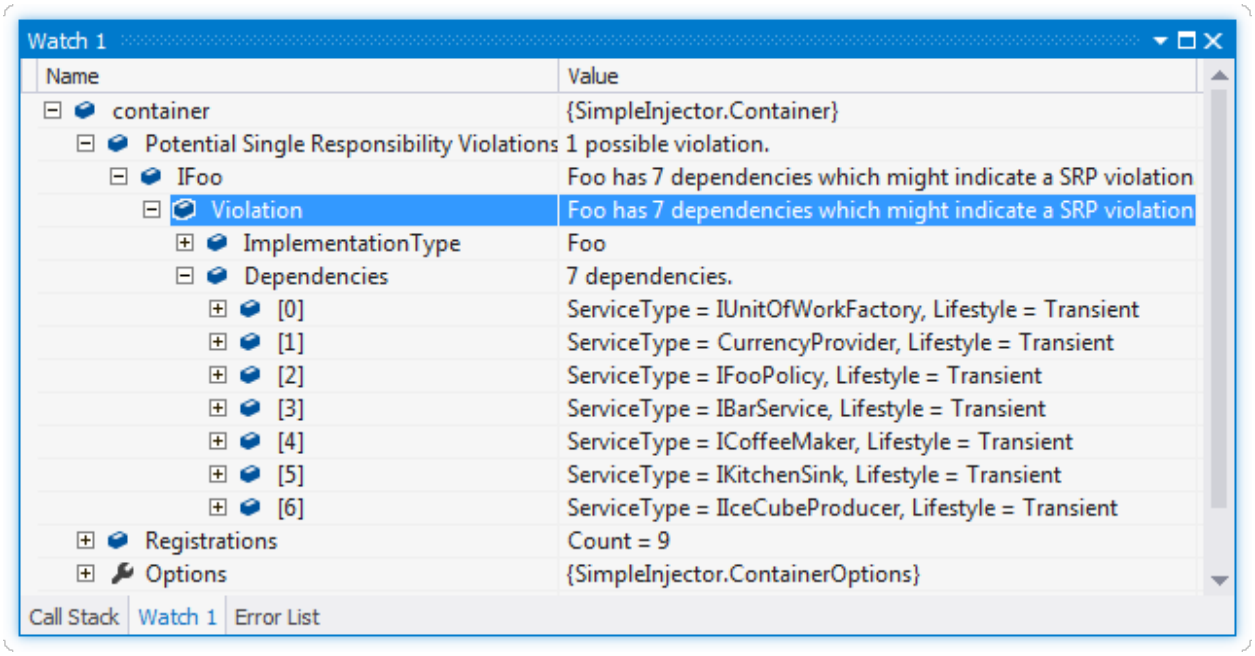
The warning can be suppressed on a per-registration basis as follows:

```
Registration registration = container.GetRegistration(typeof(IFoo)).Registration;

registration.SuppressDiagnosticWarning(DiagnosticType.SingleResponsibilityViolation);
```

Example

```
public class Foo : IFoo
{
    public Foo(
        IUnitOfWorkFactory uowFactory,
        CurrencyProvider currencyProvider,
        IFooPolicy fooPolicy,
        IBarService barService,
        ICoffeeMaker coffeeMaker,
        IKitchenSink kitchenSink,
        IIceCubeProducer iceCubeProducer,
        IWaterCooker waterCooker) {
    }
}
```

The **Foo** class has 8 dependencies and when it is registered in the container, it will result in the warning. Here is an example of this warning in the watch window:



The following example shows how to query the Diagnostic API for possible Single Responsibility Violations:

```
// using SimpleInjector.Diagnostics;

var container = /* get verified container */;

var results = Analyzer.Analyze(container)
    .OfType<SingleResponsibilityViolationDiagnosticResult>();

foreach (var result in results) {
    Console.WriteLine(result.ImplementationType.Name +
        " has " + result.Dependencies.Count + " dependencies.");
}
```

Diagnostic Warning - Container-registered Types

Severity

Information

Cause

A concrete type that was not registered explicitly and was not resolved using unregistered type resolution, but was created by the container using the default lifestyle.

Warning Description

The *Container-Registered Types* warning shows all concrete types that weren't registered explicitly, but registered by the container as transient for you, because they were referenced by another component's constructor or were resolved through a direct call to *container.GetInstance<T>()* (inside a RegisterInitializer[65] registered delegate for instance).

[65] https://simpleinjector.org/ReferenceLibrary/?topic=html/M_SimpleInjector_Container_RegisterInitializer__1.htm

This warning deserves your attention, since it might indicate that you program to implementations, instead of abstractions. Although the *Lifestyle Mismatches* and *Short Circuited Dependencies* warnings are a very strong signal of a configuration problem, this *Container-Registered Types* warnings is just a point of attention.

How to Fix Violations

Let components depend on an interface that describe the contract that this concrete type implements and register that concrete type in the container by that interface.

When to Ignore Warnings

If your intention is to resolve those types as transient and don't depend directly on their concrete types, this warning can in general be ignored safely.

The warning can be suppressed on a per-registration basis as follows:

```
var registration = container.GetRegistration(typeof(HomeController)).Registration;

registration.SuppressDiagnosticWarning(DiagnosticType.ContainerRegisteredComponent);
```

Example

```
var container = new Container();

container.Register<HomeController>();

container.Verify();

// Definition of HomeController
public class HomeController : Controller {
    private readonly SqlUserRepository repository;

    public HomeController(SqlUserRepository repository) {
        this.repository = repository;
    }
}
```

The given example registers a *HomeController* class that depends on an unregistered *SqlUserRepository* class. Injecting a concrete type can lead to problems, such as:

- It makes the *HomeController* hard to test, since concrete types are often hard to fake (or when using a mocking framework that fixes this, would still result in unit tests that contain a lot of configuration for the mocking framework, instead of pure test logic) making the unit tests hard to read and hard to maintain.
- It makes it harder to reuse the class, since it expects a certain implementation of its dependency.
- It makes it harder to add cross-cutting concerns (such as logging, audit trailing and authorization) to the system, because this must now either be added directly in the *SqlUserRepository* class (which will make this class hard to test and hard to maintain) or all constructors of classes that depend on *SqlUserRepository* must be changed to allow injecting a type that adds these cross-cutting concerns.

Instead of depending directly on *SqlUserRepository*, *HomeController* can better depend on an *IUserRepository* abstraction:

```
var container = new Container();

container.Register<IUserRepository, SqlUserRepository>();
container.Register<HomeController>();

container.Verify();

// Definition of HomeController
public class HomeController : Controller {
    private readonly IUserRepository repository;

    public HomeController(IUserRepository repository) {
        this.repository = repository;
    }
}
```

Tip: It would probably be better to define a generic *IRepository<T>* abstraction. This makes easy to *batch registration* implementations and allows cross-cutting concerns to be added using *decorators*.

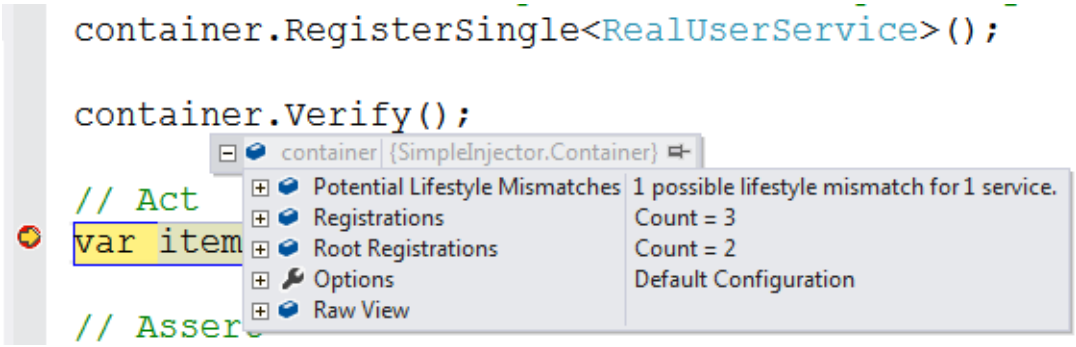
How to view diagnostic results

Note: In Simple Injector 3, the container now by default checks for problems when calling **Verify()**. In case of a diagnostic warning, the container's **Verify()** method will throw a **DiagnosticVerificationException**.

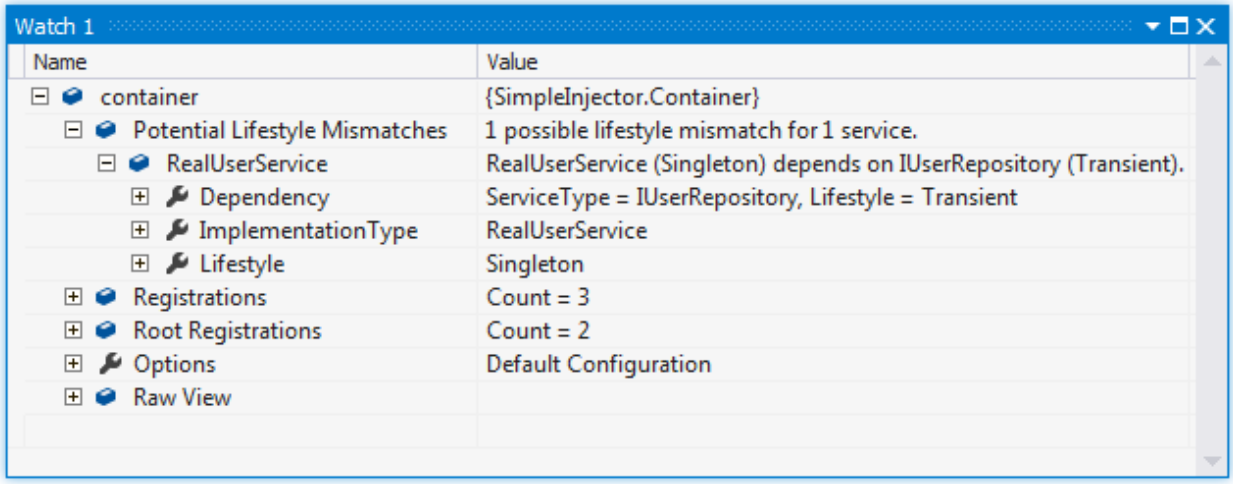
Tip: Always call **Verify()** when you're done configuring the **Container**. This allows your application to fail-fast in case of configuration problems.

There are two ways to view the diagnostic results - results can be viewed visually during debugging in Visual Studio and programmatically by calling the Diagnostic API.

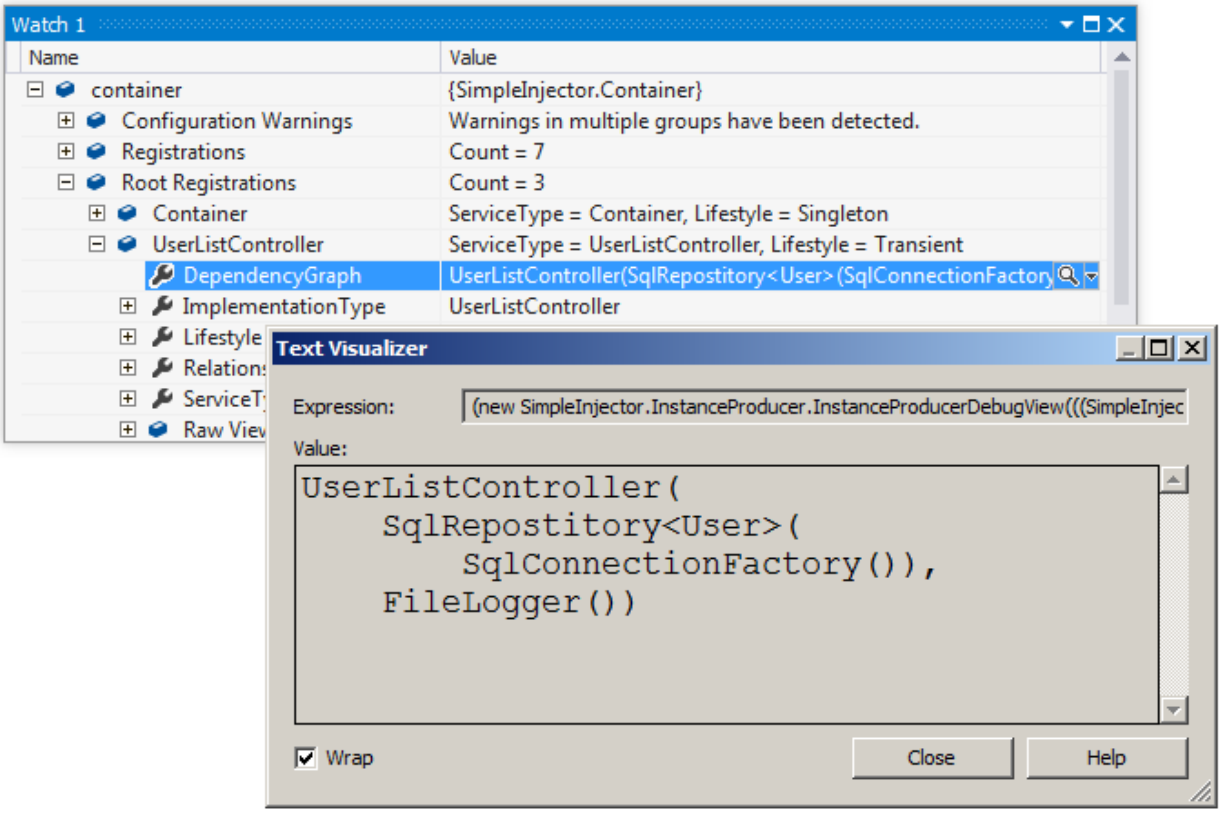
Diagnostic results are available during debugging in Visual Studio after calling **Container.Verify()**. Set a breakpoint after the line that calls **Verify()** and when the breakpoint hits, hover over the **Container** instance with the mouse. The debugger context menu will appear for the *container* variable which you can unfold to view the diagnostic results. This might look like this:



Another option is to add the *container* variable to the Visual Studio watch window by right clicking on the variable and selecting 'Add Watch' in the context menu:



The debugger views also allow visualizing your application's dependency graphs. This can give you a good view of what the end result of your DI configuration is. By drilling into the list of **Registrations** or **Root Registrations**, you can select the text visualizer (the magnifying glass icon) on the **DependencyGraph** property on any of the listed registrations:



Note: **Root Registrations** are registrations that are not depended upon by any other registration (or at least, not that Simple Injector can statically determine). They form the starting point of an object graph and are usually the types that are directly resolved from the container.

This same information can be requested programmatically by using the Diagnostic API. The Diagnostic API is located in the **SimpleInjector.Diagnostics** namespace. Interacting with the Diagnostic API is especially useful for automated testing. The following is an example of an integration test that checks whether the container is free of configuration

warnings and informational messages:

```
[TestMethod]
public void Container_Never_ContainsDiagnosticWarnings() {
    // Arrange
    var container = Bootstrapper.GetInitializedContainer();

    container.Verify(VerificationOption.VerifyOnly);

    // Assert
    var results = Analyzer.Analyze(container);

    Assert.IsFalse(results.Any(), Environment.NewLine +
        string.Join(Environment.NewLine,
            from result in results
            select result.Description));
}
```

Instead of interacting with the Diagnostic API directly, you can force the container to fail fast during verification in case one of the more severe warnings is detected:

```
var container = Bootstrapper.GetInitializedContainer();

container.Verify();
```

A call to **Verify()** defaults to **Verify(VerificationOption.VerifyAndDiagnose)**. When called, the container will check for all the warning types, since they are most likely to cause bugs in your application. By calling this overload during application startup, or inside an integration test, you'll keep the feedback cycle as short as possible, and you'll get notified about possible bugs that otherwise might have stayed undetected for much too long.

Suppressing warnings

There are rare cases that you want to ignore the warning system for specific registrations. There are scenarios where you are sure that the presented warning does not cause any harm in your case and changing the application's design is not feasible. In such situation you can suppress warnings on a case by case basis. This prevents a call to **Verify()** or **Verify(VerificationOption.VerifyAndDiagnose)** from throwing an exception, it prevents the warning from popping up in the debugger, and it prevents the *Analyzer.Analyze()* method from returning that warning.

A warning can be suppressed by disabling a specific warning type on a **Registration** object. Example:

```
var registration =
    Lifestyle.Transient.CreateRegistration(typeof(HomeController), container);

container.AddRegistration(typeof(HomeController), registration);

registration.SuppressDiagnosticWarning(DiagnosticType.DisposableTransientComponent);
```

In the previous code sample, a **Registration** instance for the *HomeController* type is created and registered in the container. This **Registration** instance is explicitly marked to suppress the diagnostic warning type **Disposable Transient Component**.

Suppressing this warning type for an MVC controller makes sense, because the MVC framework will ensure proper disposal of MVC controllers.

Alternatively, you can also request an already made registered and suppress a warning on that:

```
var registration = container.GetRegistration(typeof(HomeController)).Registration;

registration.SuppressDiagnosticWarning(DiagnosticType.DisposableTransientComponent);
```

Tip: The **RegisterMvcControllers** extension method of the **SimpleInjector.Integration.Web.Mvc.dll** will batch-register all MVC controllers and will automatically suppress the **Disposable Transient Component** warning on controller types.

Limitations

Warning: Although the *Container* can spot several configuration mistakes, be aware that there will always be ways to make configuration errors that the Diagnostic Services cannot identify. Wiring your dependencies is a delicate matter and should be done with care. Always follow best practices.

The **Diagnostic Services** work by analyzing all information that is known by the container. In general, only relationships between types that can be statically determined (such as constructor arguments) can be analyzed. The *Container* uses the following information for analysis:

- Constructor arguments of types that are created by the container (auto-wired types).
- Dependencies added by *Decorators*.
- Dependencies that are not registered explicitly but are referenced as constructor argument (this included types that got created through unregistered type resolution).

The Diagnostic Services **cannot** analyze the following:

- Types that are completely unknown, because these types are not registered explicitly and no registered type depends on them. In general you should register all root types (types that are requested directly by calling **GetInstance<T>()**, such as MVC Controllers) explicitly.
- Open-generic registrations that are resolved as root type (no registered type depends on them). Since the container uses unregistered type resolution, those registrations will be unknown until they are resolved. Prefer registering each closed-generic version explicitly, or add unit tests to verify that these root types can be resolved.
- Dependencies added using the RegisterInitializer[66] method:

```
container.RegisterInitializer<IService>(service => {
    // These dependencies will be unknown during diagnosis
    service.Dependency = new Dependency();
    service.TimeProvider = container.GetInstance<ITimeProvider>()
});
```

- Types that are created manually by registering a *Func<T>* delegate using one of the Register<TService>(Func<TService>)[67] overloads, for instance:

```
container.Register<IService>(() => new MyService(
    // These dependencies will be unknown during diagnosis
    container.GetInstance<ILogger>(),
    container.GetInstance<ITimeProvider>()));
```

- Dependencies that are resolved by requesting them manually from the *Container*, for instance by injecting the *Container* into a class and then calling *container.GetInstance<T>()* from within that class:

[66] https://simpleinjector.org/ReferenceLibrary/?topic=html/M_SimpleInjector_Container_RegisterInitializer__1.htm
[67] https://simpleinjector.org/ReferenceLibrary/?topic=html/M_SimpleInjector_Container_Register__1_2.htm

```
public class MyService : IService {
    private ITimeProvider provider;

    // Type depends on the container (don't do this)
    public MyService(Container container) {
        // This dependency will be unknown during diagnosis
        this.provider = container.GetInstance<ITimeProvider>();
    }
});
```

Tip: Try to prevent depending on any of the designs listed above because they all prevent you from having a *verifiable configuration* and trustworthy diagnostic results.

CHAPTER 6

How To

- How to *Register factory delegates*
- How to *Resolve instances by key*
- How to *Register multiple interfaces with the same implementation*
- How to *Override existing registrations*
- How to *Verify the container's configuration*
- How to *Work with dependency injection in multi-threaded applications*
- How to *Package registrations*

Register factory delegates

Simple Injector allows you to register a *Func<T>* delegate for the creation of an instance. This is especially useful in scenarios where it is impossible for the *Container* to create the instance. There are overloads of the **Register** method available that accept a *Func<T>* argument:

```
container.Register<IMyService>(() => SomeSubSystem.CreateMyService());
```

In situations where a service needs to create multiple instances of a certain component, or needs to explicitly control the lifetime of such component, abstract factories can be used. Instead of injecting an *IMyService*, you should inject an *IMyServiceFactory* that creates new instances of *IMyService*:

```
// Definition
public interface IMyServiceFactory {
    IMyService CreateNew();
}

// Implementation
sealed class ServiceFactory : IMyServiceFactory {
    public IMyService CreateNew() {
```

```
        return new MyServiceImpl();
    }
}

// Registration
container.RegisterSingleton<IMyServiceFactory>(new ServiceFactory());

// Usage
public class MyService {
    private readonly IMyServiceFactory factory;

    public MyService(IMyServiceFactory factory) {
        this.factory = factory;
    }

    public void SomeOperation() {
        using (var service1 = this.factory.CreateNew()) {
            // use service 1
        }

        using (var service2 = this.factory.CreateNew()) {
            // use service 2
        }
    }
}
```

Instead of creating specific interfaces for your factories, you can also choose to inject *Func<T>* delegates into your services:

```
// Registration
container.RegisterSingleton<Func<IMyService>>(() => new MyServiceImpl());

// Usage
public class MyService {
    private readonly Func<IMyService> factory;

    public MyService(Func<IMyService> factory) {
        this.factory = factory;
    }

    public void SomeOperation() {
        using (var service1 = this.factory.Invoke()) {
            // use service 1
        }
    }
}
```

This saves you from having to define a new interface and implementation per factory.

Note: On the downside however, this communicates less clearly the intent of your code and as a result might make your code harder to grasp.

When you choose *Func<T>* delegates over specific factory interfaces you can define the following extension method to simplify the registration of *Func<T>* factories:

```
// using System;
// using SimpleInjector;
// using SimpleInjector.Advanced;
```

```
public static void RegisterFuncFactory<TService, TImpl>(
    this Container container, Lifestyle lifestyle = null)
    where TService : class
    where TImpl : class, TService
{
    lifestyle = lifestyle ?? container.Options.DefaultLifestyle;
    var producer = lifestyle.CreateProducer<TService, TImpl>(container);
    container.RegisterSingleton<Func<TService>>(producer.GetInstance);
}

// Registration
container.RegisterFuncFactory<IMyService, RealService>();
```

The extension method allows registration of a single factory.

To take this one step further, the following extension method allows Simple Injector to resolve all types using a *Func<T>* delegate by default:

```
// using System;
// using System.Linq;
// using System.Linq.Expressions;
// using SimpleInjector;
public static void AllowResolvingFuncFactories(this ContainerOptions options) {
    options.Container.ResolveUnregisteredType += (s, e) => {
        var type = e.UnregisteredServiceType;

        if (!type.IsGenericType || type.GetGenericTypeDefinition() != typeof(Func<>))
↪{
            return;
        }

        Type serviceType = type.GetGenericArguments().First();

        InstanceProducer registration =
            options.Container.GetRegistration(serviceType, true);

        Type funcType = typeof(Func<>).MakeGenericType(serviceType);

        var factoryDelegate = Expression.Lambda(funcType,
            registration.BuildExpression()).Compile();

        e.Register(Expression.Constant(factoryDelegate));
    };
}

// Registration
container.Options.AllowResolvingFuncFactories();
```

After calling this *AllowResolvingFuncFactories* extension method, the container allows resolving *Func<T>* delegates.

Warning: We personally think that allowing to register *Func<T>* delegates by default is a design smell. The use of *Func<T>* delegates makes your design harder to follow and your system harder to maintain and test. Your system should only have a few of those factories at most. If you have many constructors in your system that depend on a *Func<T>*, please take a good look at your dependency strategy. The following article[68] goes into details about why Abstract Factories (such as *Func<T>*) are a design smell.

[68] https://www.cuttingedge.it/blogs/steven/pivot/entry.php?id=100

Lazy

Just like *Func<T>* delegates can be injected, *Lazy<T>* instances can also be injected into components. *Lazy<T>* is useful in situations where the creation of a component is time consuming and not always required. *Lazy<T>* enables you to postpone the creation of such a component until the moment it is actually required:

```
// Registration
container.Register<Lazy<IMyService>>(
    () => new Lazy<IMyService>(container.GetInstance<RealService>));

// Usage
public class SomeController {
    private readonly Lazy<IMyService> myService;

    public SomeController(Lazy<IMyService> myService) {
        this.myService = myService;
    }

    public void SomeOperation(bool someCondition) {
        if (someCondition) {
            this.myService.Value.Operate();
        }
    }
}
```

Tip: instead of polluting the API of your application with *Lazy<T>* dependencies, it is usually cleaner to hide the *Lazy<T>* behind a proxy, as shown in the following example.

```
// Proxy definition
public class LazyServiceProxy : IMyService {
    private readonly Lazy<IMyService> wrapped;

    public LazyServiceProxy(Lazy<IMyService> wrapped) {
        this.wrapped = wrapped;
    }

    public void Operate() => this.wrapped.Value.Operate();
}

// Registration
container.Register<IMyService>(() => new LazyServiceProxy(
    new Lazy<IMyService>(container.GetInstance<RealService>)));
```

This way the application can simply depend on *IMyService* instead of *Lazy<IMyService>*:

```
// Usage
public class SomeController {
    private readonly IMyService myService;

    public SomeController(IMyService myService) {
        this.myService = myService;
    }

    public void SomeOperation(bool someCondition) {
        if (someCondition) {
            this.myService.Operate();
        }
    }
```

```
}
```

Warning: The same warning applies to the use of *Lazy<T>* as it does for the use of *Func<T>* delegates. Further more, the constructors of your components should be simple, reliable and quick (as explained in this blog post[69] by Mark Seemann), and that would remove the need for lazy initialization. For more information about creating an application and container configuration that can be successfully verified, please read the *How To Verify the container's configuration*.

Resolve instances by key

Resolving instances by a key is a feature that is deliberately left out of Simple Injector, because it invariably leads to a design where the application tends to have numerous dependencies on the DI container itself. To resolve a keyed instance you will likely need to call directly into the *Container* instance and this leads to the Service Locator anti-pattern[70].

This doesn't mean that resolving instances by a key is never useful. Resolving instances by a key is normally a job for a specific factory rather than the *Container*. This approach makes the design much cleaner, saves you from having to take numerous dependencies on the DI library and enables many scenarios that the DI container authors simply didn't consider.

Note: The need for keyed registration can be an indication of ambiguity in the application design and a sign of a Liskov Substitution Principle[71] violation. Take a good look if each keyed registration shouldn't have its own unique interface, or perhaps each registration should implement its own version of a generic interface.

Take a look at the following scenario, where we want to retrieve instances of type *IRequestHandler* by a string key. There are of course several ways to achieve this, but here is a simple but effective way, by defining an *IRequestHandlerFactory*:

```
// Definition
public interface IRequestHandlerFactory
{
    IRequestHandler CreateNew(string name);
}

// Usage
var factory = container.GetInstance<IRequestHandlerFactory>();
var handler = factory.CreateNew("customers");
handler.Handle(requestContext);
```

By inheriting from the BCL's *Dictionary<TKey, TValue>*, creating an *IRequestHandlerFactory* implementation is almost a one-liner:

```
public class RequestHandlerFactory : Dictionary<string, Func<IRequestHandler>>,
    IRequestHandlerFactory {
    public IRequestHandler CreateNew(string name) => this[name]();
}
```

With this class, we can register *Func<IRequestHandler>* factory methods by a key. With this in place the registration of keyed instances is a breeze:

```
var container = new Container();
```

[69] http://blog.ploeh.dk/2011/03/03/InjectionConstructorsshouldbesimple/
[70] http://blog.ploeh.dk/2010/02/03/ServiceLocatorIsAnAntiPattern.aspx
[71] https://en.wikipedia.org/wiki/Liskov_substitution_principle

```
container.RegisterSingleton<IRequestHandlerFactory>(new RequestHandlerFactory {
    { "default", () => container.GetInstance<DefaultRequestHandler>() },
    { "orders", () => container.GetInstance<OrdersRequestHandler>() },
    { "customers", () => container.GetInstance<CustomersRequestHandler>() },
});
```

If you don't like a design that uses *Func<T>* delegates this way, it can easily be changed to be a *Dictionary<string, Type>* instead. The *RequestHandlerFactory* can be implemented as follows:

```
public class RequestHandlerFactory : Dictionary<string, Type>, IRequestHandlerFactory
↪{
    private readonly Container container;

    public RequestHandlerFactory(Container container) {
        this.container = container;
    }

    public IRequestHandler CreateNew(string name) =>
        (IRequestHandler)this.container.GetInstance(this[name]);
}
```

The registration will then look as follows:

```
var container = new Container();

container.RegisterSingleton<IRequestHandlerFactory>(new␣
↪RequestHandlerFactory(container) {
    { "default", typeof(DefaultRequestHandler) },
    { "orders", typeof(OrdersRequestHandler) },
    { "customers", typeof(CustomersRequestHandler) },
});
```

Note: Please remember the previous note about ambiguity in the application design. In the given example the design would probably be better of by using a generic *IRequestHandler<TRequest>* interface. This would allow the implementations to be *batch registered using a single line of code*, saves you from using keys, and results in a configuration that is *verifiable by the container*.

A final option for implementing keyed registrations is to manually create the registrations and store them in a dictionary. The following example shows the same *RequestHandlerFactory* using this approach:

```
public class RequestHandlerFactory : IRequestHandlerFactory {
    readonly Container container;
    readonly Dictionary<string, InstanceProducer<IRequestHandler>> producers =
        new Dictionary<string, InstanceProducer<IRequestHandler>>(
            StringComparer.OrdinalIgnoreCase);

    public RequestHandlerFactory(Container container) {
        this.container = container;
    }

    IRequestHandler IRequestHandlerFactory.CreateNew(string name) =>
        this.producers[name].GetInstance();

    public void Register<TImplementation>(string name, Lifestyle lifestyle = null)
        where TImplementation : class, IRequestHandler {
        var producer = (lifestyle ?? container.Options.DefaultLifestyle)
            .CreateProducer<IRequestHandler, TImplementation>(container);
```

```
        this.producers.Add(name, producer);
    }
}
```

The registration will then look as follows:

```
var container = new Container();

var factory = new RequestHandlerFactory(container);

factory.Register<DefaultRequestHandler>("default");
factory.Register<OrdersRequestHandler>("orders");
factory.Register<CustomersRequestHandler>("customers");

container.RegisterSingleton<IRequestHandlerFactory>(factory);
```

The advantage of this method is that it completely integrates with the *Container*. *Decorators* can be applied to individual returned instances, types can be registered multiple times and the registered handlers can be analyzed using the *Diagnostic Services*.

The previous examples showed how registrations could be requested based on a key. Another common use case is to have multiple consumers of a given abstraction, where each consumer requires a different implementation of that abstraction. In Simple Injector this can be achieved through *Context based injection*.

Register multiple interfaces with the same implementation

To adhere to the Interface Segregation Principle[72], it is important to keep interfaces narrow. Although in most situations implementations implement a single interface, it can sometimes be beneficial to have multiple interfaces on a single implementation. Here is an example of how to register this:

```
// Impl implements IInterface1, IInterface2 and IInterface3.
container.Register<IInterface1, Impl>(Lifestyle.Singleton);
container.Register<IInterface2, Impl>(Lifestyle.Singleton);
container.Register<IInterface3, Impl>(Lifestyle.Singleton);

var a = container.GetInstance<IInterface1>();
var b = container.GetInstance<IInterface2>();
var c = container.GetInstance<IInterface3>();

// Since Impl is a singleton, all requests return the same instance.
Assert.AreEqual(a, b);
Assert.AreEqual(b, c);
```

At first glance the previous example would seem to cause three instances of *Impl*, but Simple Injector 4 will ensure that all three registrations will get the same instance.

Override existing registrations

The default behavior of Simple Injector is to fail when a service is registered for a second time. Most of the time the developer didn't intend to override a previous registration and allowing this would lead to a configuration that would pass the container's verification, but doesn't behave as expected.

[72] https://en.wikipedia.org/wiki/Interface_segregation_principle

This design decision differs from most other DI libraries, where adding new registrations results in appending the collection of registrations for that abstraction. Registering collections in Simple Injector is an *explicit action* done using one of the RegisterCollection[73] method overloads.

There are certain scenarios however where overriding is useful. An example of such is a bootstrapper project for a business layer that is reused in multiple applications (in both a web application, web service, and Windows service for instance). Not having a business layer specific bootstrapper project would mean the complete DI configuration would be duplicated in the startup path of each application, which would lead to code duplication. In that situation the applications would roughly have the same configuration, with a few adjustments.

Best is to start of by configuring all possible dependencies in the BL bootstrapper and leave out the service registrations where the implementation differs for each application. In other words, the BL bootstrapper would result in an incomplete configuration. After that, each application can finish the configuration by registering the missing dependencies. This way you still don't need to override the existing configuration.

In certain scenarios it can be beneficial to allow an application override an existing configuration. The container can be configured to allow overriding as follows:

```
var container = new Container();

container.Options.AllowOverridingRegistrations = true;

// Register IUserService.
container.Register<IUserService, FakeUserService>();

// Replaces the previous registration
container.Register<IUserService, RealUserService>();
```

The previous example created a *Container* instance that allows overriding. It is also possible to enable overriding half way the registration process:

```
// Create a container with overriding disabled
var container = new Container();

// Pass container to the business layer.
BusinessLayer.Bootstrapper.Bootstrap(container);

// Enable overriding
container.Options.AllowOverridingRegistrations = true;

// Replaces the previous registration
container.Register<IUserService, RealUserService>();
```

Verify the container's configuration

Dependency Injection promotes the concept of programming against abstractions. This makes your code much easier to test, easier to change and maintain. However, since the code itself isn't responsible for maintaining the dependencies between implementations when using a DI library, the compiler will not be able to verify whether the dependency graph is correct.

When starting to use a Dependency Injection container, many developers see their application fail when it is deployed in staging or sometimes even production, because of container misconfigurations. This makes developers often conclude that dependency injection is bad, since the dependency graph cannot be verified. This conclusion however, is incorrect. First of all, the use of Dependency Injection doesn't require a DI library at all. The pattern is still valid, even

[73] https://simpleinjector.org/ReferenceLibrary/?topic=html/Overload_SimpleInjector_Container_RegisterCollection.htm

without the use of tooling that will wire everything together for you. For some types of applications Pure DI[74] is even advisable. Second, although it is impossible for the compiler to verify the dependency graph when using a DI library, verifying the dependency graph is still possible and advisable.

Simple Injector contains a **Verify()** method, that will iterate over all registrations and resolve an instance for each registration. Calling this method directly after configuring the container allows the application to fail during start-up if the configuration is invalid.

Calling the **Verify()** method however, is just part of the story. It is very easy to create a configuration that passes any verification, but still fails at runtime. Here are some tips to help building a verifiable configuration:

1. Stay away from *implicit property injection*, where the container is allowed to skip injecting the property if a corresponding or correctly registered dependency can't be found. This will disallow your application to fail fast and will result in *NullReferenceException*'s later on. Only use implicit property injection when the property is truly optional, omitting the dependency still keeps the configuration valid, and the application still runs correctly without that dependency. Truly optional dependencies should be very rare though, since most of the time you should prefer injecting empty implementations (a.k.a. the Null Object pattern[75]) instead of allowing dependencies to be a null reference. *Explicit property injection* on the other hand is better. With explicit property injection you force the container to inject a property and it will fail when it can't succeed. However, you should prefer constructor injection whenever possible. Note that the need for property injection is often an indication of problems in the design. If you revert to property injection because you otherwise have too many constructor arguments, you're probably violating the Single Responsibility Principle[76].

2. Register all root objects explicitly. For instance, register all ASP.NET MVC Controller instances explicitly in the container (Controller instances are requested directly and are therefore called 'root objects'). This way the container can check the complete dependency graph starting from the root object when you call **Verify()**. Prefer registering all root objects in an automated fashion, for instance by using reflection to find all root types. The Simple Injector ASP.NET MVC Integration NuGet Package[77] for instance, contains a RegisterMvcControllers[78] extension method that will do this for you and the WCF Integration NuGet Package[79] contains a similar RegisterWcfServices[80] extension method for this purpose.

3. If any of your root types are generic you should explicitly register each required closed-generic version of the type instead of making a single open-generic registration per generic type. Simple Injector will not be able to guess the closed types that could be resolved (root types are not referenced by other types and there can be endless permutations of closed-generic types) and as such open generic registrations are skipped by Simple Injector's verification system. Making an explicit registration for each closed-generic root type allows Simple Injector to verify and diagnose those registrations.

4. If registering root objects is not possible or feasible, test the creation of each root object manually during start-up. With ASP.NET Web Form Page classes for instance, you will probably call the container (directly or indirectly) from within their constructor (since Page classes must unfortunately have a default constructor). The key here again is finding them all in once using reflection. By finding all Page classes using reflection and instantiating them, you'll find out (during app start-up or through automated testing) whether there is a problem with your DI configuration or not. The *Web Forms Integration* guide contains an example of how to verify page classes.

5. There are scenarios where some dependencies cannot yet be created during application start-up. To ensure that the application can be started normally and the rest of the DI configuration can still be verified, abstract those dependencies behind a proxy or abstract factory. Try to keep those unverifiable dependencies to a minimum and keep good track of them, because you will probably have to test them manually or using an integration test.

6. But even when all registrations can be resolved successfully by the container, that still doesn't mean your configuration is correct. It is very easy to accidentally misconfigure the container in a way that only shows

[74] http://blog.ploeh.dk/2014/06/10/pure-di/
[75] https://en.wikipedia.org/wiki/Null_Object_pattern
[76] https://en.wikipedia.org/wiki/Single_responsibility_principle
[77] https://nuget.org/packages/SimpleInjector.Integration.Web.Mvc
[78] https://simpleinjector.org/ReferenceLibrary/?topic=html/M_SimpleInjector_SimpleInjectorMvcExtensions_RegisterMvcControllers.htm
[79] https://nuget.org/packages/SimpleInjector.Integration.Wcf
[80] https://simpleinjector.org/ReferenceLibrary.v2/?topic=html/M_SimpleInjector_SimpleInjectorWcfExtensions_RegisterWcfServices.htm

up late in the development process. Simple Injector contains *Diagnostics Services* to help you spot common configuration mistakes. To help you, all the diagnostic warnings are integrated into the verification mechanism. This means that a call to **Verify()** will also check for diagnostic warnings for you. It is advisable to analyze the container by calling **Verify** or by using the diagnostic services either during application startup or as part of an automated test that does this for you.

Work with dependency injection in multi-threaded applications

Note: Simple Injector is designed for use in highly-concurrent applications and the container is thread-safe. Its lock-free design allows it to scale linearly with the number of threads and processors in your system.

Many applications and application frameworks are inherently multi-threaded. Working in multi-threaded applications forces developers to take special care. It is easy for a less experienced developer to introduce a race condition in the code. Even although some frameworks such as ASP.NET make it easy to write thread-safe code, introducing a simple static field could break thread-safety.

This same holds when working with DI containers in multi-threaded applications. The developer that configures the container should be aware of the risks of shared state. **Not knowing which configured services are thread-safe is a sin.** Registering a service that is not thread-safe as singleton, will eventually lead to concurrency bugs, that usually only appear in production. Those bugs are often hard to reproduce and hard to find, making them costly to fix. And even when you correctly configured a service with the correct lifestyle, when another component that depends on it accidentally as a longer lifetime, the service might be kept alive much longer and might even be accessible from other threads.

Dependency injection however, can actually help in writing multi-threaded applications. Dependency injection forces you to wire all dependencies together in a single place in the application: the Composition Root[81]. This means that there is a single place in the application that knows about how services behave, whether they are thread-safe, and how they should be wired. Without this centralization, this knowledge would be scattered throughout the code base, making it very hard to change the behavior of a service.

Tip: Take a close look at the 'Lifestyle Mismatches' warnings in the *Diagnostic Services*. Lifestyle mismatches are a source of concurrency bugs.

Note: By default, Simple Injector will check for Lifestyle Mismatches for you when you resolve a service. In other words, Simple Injector will fail fast when there is a Lifestyle Mismatch in your configuration.

In a multi-threaded application, each thread should get its own object graph. This means that you should typically call **GetInstance<T>()** once at the beginning of the thread's execution to get the root object for processing that thread (or request). The container will build an object graph with all root object's dependencies. Some of those dependencies might be singletons; shared between all threads. Other dependencies might be transient; a new instance is created per dependency. Other dependencies might be thread-specific, request-specific, or with some other lifestyle. The application code itself is unaware of the way the dependencies are registered and that's the way it is supposed to be.

For web applications, you typically call **GetInstance<T>()** at the beginning of the web request. In an ASP.NET MVC application for instance, one Controller instance will be requested from the container (by the Controller Factory) per web request. When using one of the integration packages, such as the Simple Injector MVC Integration Quick Start NuGet package[82] for instance, you don't have to call **GetInstance<T>()** yourself, the package will ensure this is done for you. Still, **GetInstance<T>()** is typically called once per request.

The advice of building a new object graph (calling **GetInstance<T>()**) at the beginning of a thread, also holds when manually starting a new (background) thread. Although you can pass on data to other threads, you should not pass on container controlled dependencies to other threads. On each new thread, you should ask the container again for the dependencies. When you start passing dependencies from one thread to the other, those parts of the code have to know whether it is safe to pass those dependencies on. For instance, are those dependencies thread-safe? This might

[81] http://blog.ploeh.dk/2011/07/28/CompositionRoot/
[82] https://nuget.org/packages/SimpleInjector.MVC3

be trivial to analyze in some situations, but prevents you to change those dependencies with other implementations, since now you have to remember that there is a place in your code where this is happening and you need to know which dependencies are passed on. You are decentralizing this knowledge again, making it harder to reason about the correctness of your DI configuration and making it easier to misconfigure the container in a way that causes concurrency problems.

Running code on a new thread can be done by adding a little bit of infrastructural code. Take for instance the following example where we want to send e-mail messages asynchronously. Instead of letting the caller implement this logic, it is better to hide the logic for asynchronicity behind an abstraction; a proxy. This ensures that this logic is centralized to a single place, and by placing this proxy inside the composition root, we prevent the application code to take a dependency on the container itself (which should be prevented).

```
// Synchronous implementation of IMailSender
public sealed class RealMailSender : IMailSender {
    private readonly IMailFormatter formatter;

    public class RealMailSender(IMailFormatter formatter) {
        this.formatter = formatter;
    }

    void IMailSender.SendMail(string to, string message) {
        // format mail
        // send mail
    }
}

// Proxy for executing IMailSender asynchronously.
sealed class AsyncMailSenderProxy : IMailSender {
    private readonly ILogger logger;
    private readonly Func<IMailSender> mailSenderFactory;

    public AsyncMailSenderProxy(ILogger logger, Func<IMailSender> mailSenderFactory) {
        this.logger = logger;
        this.mailSenderFactory = mailSenderFactory;
    }

    void IMailSender.SendMail(string to, string message) {
        // Run on a new thread
        Task.Factory.StartNew(() => {
            this.SendMailAsync(to, message);
        });
    }

    private void SendMailAsync(string to, string message) {
        // Here we run on a different thread and the
        // services should be requested on this thread.
        var mailSender = this.mailSenderFactory();

        try {
            mailSender.SendMail(to, message);
        }
        catch (Exception ex) {
            // logging is important, since we run on a
            // different thread.
            this.logger.Log(ex);
        }
    }
}
```

In the Composition Root, instead of registering the *MailSender*, we register the *AsyncMailSenderProxy* as follows:

```
container.Register<ILogger, FileLogger>(Lifestyle.Singleton);
container.Register<IMailSender, RealMailSender>();
container.RegisterDecorator<IMailSender, AsyncMailSenderProxy>(Lifestyle.Singleton);
```

In this case the container will ensure that when an *IMailSender* is requested, a single *AsyncMailSenderProxy* is returned with a *Func<IMailSender>* delegate that will create a new *RealMailSender* when requested. The RegisterDecorator[83] overloads natively understand how to handle *Func<Decoratee>* dependencies. The *Decorators* section explains more about registering decorators.

Warning: Please note that the previous example is just meant for educational purposes. In practice, you want the sending of e-mails to go through a durable queue or outbox to prevent loss of e-mails. Loss can occur when the mail server is unavailable, which is something that is guaranteed to happen at some point in time, even when the mail server is running locally.

Package registrations

Simple Injector has the notion of 'packages'. A package is a group of container registrations packed into a class that implements the **IPackage** interface. This feature is similar to what other containers call Installers, Modules or Registries.

To use this feature, you need to install the SimpleInjector.Packaging NuGet package[84].

SimpleInjector.Packaging exists to accommodate applications that require plug-in like modularization, where parts of the application, packed with their own container registrations, can be independently compiled into a dll and 'dropped' into a folder, where the main application can pick them up, without the need for the main application to be recompiled and redeployed.

To accommodate this, those independent application parts can create a package by defining a class that implements the **IPackage** interface:

```
public class ModuleXPackage : IPackage
{
    public void RegisterServices(Container container)
    {
        container.Register<IService1, Service1Impl>();
        container.Register<IService2, Service2Impl>();
    }
}
```

After doing so, the main application can dynamically load these application modules, and make sure their packages are ran:

```
var assemblies =
    from file in new DirectoryInfo(pluginDirectory).GetFiles()
    where file.Extension.ToLower() == ".dll"
    select Assembly.Load(AssemblyName.GetAssemblyName(file.FullName));

container.RegisterPackages(assemblies);
```

As explained above, **SimpleInjector.Packaging** is specifically designed for loading configurations from assemblies that are loaded dynamically. In other scenarios the use of Packaging is discouraged.

[83] https://simpleinjector.org/ReferenceLibrary/?topic=html/Overload_SimpleInjector_Extensions_DecoratorExtensions_RegisterDecorator.htm
[84] https://www.nuget.org/packages/SimpleInjector.Packaging/

For non-plug-in scenario's, all container registrations should be located as close as possible to the application's entry point. This location is commonly referred to as the Composition Root[85].

Although even inside the Composition Root it might make sense to split the registration into multiple functions or even classes, as long as those registrations are available to the entry-point at compile time, it makes more sense to call them statically instead of by the use of reflection, as can be seen in the following example:

```
public void App_Start()
{
    var container = new Container();
    container.Options.DefaultScopedLifestyle = new WebRequestLifestyle();
    BusinessLayerBootstrapper.Bootstrap(container);
    PresentationLayerBootstrapper.Bootstrap(container);

    // add missing registrations here.

    container.Verify();
}

class BusinessLayerBootstrapper {
    public static void Bootstrap(Container container) { ... }
}

class PresentationLayerBootstrapper {
    public static void Bootstrap(Container container) { ... }
}
```

The previous example gives the same amount of componentization, while everything is visibly referenced from within the start-up path. In other words, you can use your IDE's *go-to reference* feature to jump directly to that code, while still being able to group things together.

On top of this, switching on or off groups of registrations based on configuration settings becomes simpler, as can be seen in the following example:

```
if (ConfigurationManager.AppSettings["environment"] != "production")
     MockedExternalServicesPackage.Bootstrap(container);
else
     ProductionExternalServicesPackage.Bootstrap(container);
```

[85] http://blog.ploeh.dk/2011/07/28/CompositionRoot/

CHAPTER 7

Advanced Scenarios

Although its name may not imply it, Simple Injector is capable of handling many advanced scenarios.

This chapter discusses the following subjects:

- *Generics*
- *Batch registration / Automatic registration*
- *Registration of open generic types*
- *Mixing collections of open-generic and non-generic components*
- *Unregistered type resolution*
- *Context based injection / Contextual binding*
- *Property injection*
- *Covariance and Contravariance*
- *Registering plugins dynamically*

Generics

.NET has superior support for generic programming and Simple Injector has been designed to make full use of it. Simple Injector arguably has the most advanced support for generics of all DI libraries. Simple Injector can handle any generic type and implementing patterns such as decorator, mediator, strategy and chain of responsibility is simple.

Aspect-Oriented Programming is easy with Simple Injector's advanced support for generics. Generic decorators with generic type constraints can be registered with a single line of code and can be applied conditionally using predicates. Simple Injector can handle open generic types, closed generic types and partially-closed generic types. The sections below provides more detail on Simple Injector's support for generic typing:

- *Batch registration of non-generic types based on an open-generic interface*
- *Registering open generic types and working with partially-closed types*

- *Mixing collections of open-generic and non-generic components*
- *Resolving Covariant/Contravariant types*
- *Registration of generic decorators*

Batch / Automatic registration

Batch or automatic registration is a way of registering a set of (related) types in one go based on some convention. This feature removes the need to constantly update the container's configuration each and every time a new type is added. The following example show a series of manually registered repositories:

```
container.Register<IUserRepository, SqlUserRepository>();
container.Register<ICustomerRepository, SqlCustomerRepository>();
container.Register<IOrderRepository, SqlOrderRepository>();
container.Register<IProductRepository, SqlProductRepository>();
// and the list goes on...
```

To prevent having to change the container for each new repository we can use the non-generic registration overloads in combination with a simple LINQ query:

```
var repositoryAssembly = typeof(SqlUserRepository).Assembly;

var registrations =
    from type in repositoryAssembly.GetExportedTypes()
    where type.Namespace == "MyComp.MyProd.BL.SqlRepositories"
    where type.GetInterfaces().Any()
    select new { Service = type.GetInterfaces().Single(), Implementation = type };

foreach (var reg in registrations) {
    container.Register(reg.Service, reg.Implementation, Lifestyle.Transient);
}
```

Although many other DI libraries contain an advanced API for doing convention based registration, we found that doing this with custom LINQ queries is easier to write, more understandable, and can often prove to be more flexible than using a predefined and restrictive API.

Another interesting scenario is registering multiple implementations of a generic interface. Say for instance your application contains the following interface:

```
public interface IValidator<T> {
    ValidationResults Validate(T instance);
}
```

Your application might contain many implementations of this interface for validating Customers, Employees, Products, Orders, etc. Without batch registration you would probably end up with a set registrations similar to those we've already seen:

```
container.Register<IValidator<Customer>, CustomerValidator>();
container.Register<IValidator<Employee>, EmployeeValidator>();
container.Register<IValidator<Order>, OrderValidator>();
container.Register<IValidator<Product>, ProductValidator>();
// and the list goes on...
```

By using the **Register** overload for batch registration, the same registrations can be made in a single line of code:

```
container.Register(typeof(IValidator<>), new[] { typeof(IValidator<>).Assembly });
```

By default **Register** searches the supplied assemblies for all types that implement the *IValidator<T>* interface and registers each type by their specific (closed generic) interface. It even works for types that implement multiple closed versions of the given interface.

Note: There is a **Register** overload available that takes a list of *System.Type* instances, instead a list of *Assembly* instances and there is a **GetTypesToRegister** method that allows retrieving a list of types based on a given service type for a set of given assemblies.

Above are a couple of examples of the things you can do with batch registration. A more advanced scenario could be the registration of multiple implementations of the same closed generic type to a common interface, i.e. a set of types that all implement the same interface.

As an example, imagine the scenario where you have a *CustomerValidator* type and a *GoldCustomerValidator* type and they both implement *IValidator<Customer>* and you want to register them both at the same time. The earlier registration methods would throw an exception alerting you to the fact that you have multiple types implementing the same closed generic type. The following registration however, does enable this scenario:

```
var assemblies = new[] { typeof(IValidator<>).Assembly };
container.RegisterCollection(typeof(IValidator<>), assemblies);
```

The code snippet registers all types from the given assembly that implement *IValidator<T>*. As we now have multiple implementations the container cannot inject a single instance of *IValidator<T>* and because of this, we need to register collections. Because we register a collection, we can no longer call **container.GetInstance<IValidator<T>>()**. Instead instances can be retrieved by having an *IEnumerable<IValidator<T>>* constructor argument or by calling **container.GetAllInstances<IValidator<T>>()**.

It is not generally regarded as best practice to have an *IEnumerable<IValidator<T>>* dependency in multiple class constructors (or accessed from the container directly). Depending on a set of types complicates your application design, can lead to code duplication. This can often be simplified with an alternate configuration. A better way is to have a single composite type that wraps *IEnumerable<IValidator<T>>* and presents it to the consumer as a single instance, in this case a *CompositeValidator<T>*:

```
public class CompositeValidator<T> : IValidator<T> {
    private readonly IEnumerable<IValidator<T>> validators;

    public CompositeValidator(IEnumerable<IValidator<T>> validators) {
        this.validators = validators;
    }

    public ValidationResults Validate(T instance) {
        var allResults = ValidationResults.Valid;

        foreach (var validator in this.validators) {
            var results = validator.Validate(instance);
            allResults = ValidationResults.Join(allResults, results);
        }

        return allResults;
    }
}
```

This *CompositeValidator<T>* can be registered as follows:

```
container.Register(typeof(IValidate<>), typeof(CompositeValidator<>),
    Lifestyle.Singleton);
```

This registration maps the open generic *IValidator<T>* interface to the open generic *CompositeValidator<T>* implementation. Because the *CompositeValidator<T>* contains an *IEnumerable<IValidator<T>>* dependency, the registered types will be injected into its constructor. This allows you to let the rest of the application simply depend on the *IValidator<T>*, while registering a collection of *IValidator<T>* implementations under the covers.

Note: Simple Injector preserves the lifestyle of instances that are returned from an injected *IEnumerable<T>* instance. In reality you should not see the the injected *IEnumerable<IValidator<T>>* as a collection of implementations, you should consider it a **stream** of instances. Simple Injector will always inject a reference to the same stream (the *IEnumerable<T>* itself is a singleton) and each time you iterate the *IEnumerable<T>*, for each individual component, the container is asked to resolve the instance based on the lifestyle of that component. Regardless of the fact that the *CompositeValidator<T>* is registered as singleton the validators it wraps will each have their own specific lifestyle.

The next section will explain mapping of open generic types (just like the *CompositeValidator<T>* as seen above).

Registration of open generic types

When working with generic interfaces, we will often see numerous implementations of that interface being registered:

```
container.Register<IValidate<Customer>, CustomerValidator>();
container.Register<IValidate<Employee>, EmployeeValidator>();
container.Register<IValidate<Order>, OrderValidator>();
container.Register<IValidate<Product>, ProductValidator>();
// and the list goes on...
```

As the previous section explained, this can be rewritten to the following one-liner:

```
container.Register(typeof(IValidate<>), new[] { typeof(IValidate<>).Assembly });
```

Sometimes you'll find that many implementations of the given generic interface are no-ops or need the same standard implementation. The *IValidate<T>* is a good example. It is very likely that not all entities will need validation but your solution would like to treat all entities the same and not need to know whether any particular type has validation or not (having to write a specific empty validation for each type would be a horrible task). In a situation such as this we would ideally like to use the registration as described above, and have some way to fallback to some default implementation when no explicit registration exist for a given type. Such a default implementation could look like this:

```
// Implementation of the Null Object pattern.
public sealed class NullValidator<T> : IValidate<T> {
    public ValidationResults Validate(T instance) => ValidationResults.Valid;
}
```

We could configure the container to use this *NullValidator<T>* for any entity that does not need validation:

```
container.Register<IValidate<OrderLine>, NullValidator<OrderLine>>();
container.Register<IValidate<Address>, NullValidator<Address>>();
container.Register<IValidate<UploadImage>, NullValidator<UploadImage>>();
container.Register<IValidate<Mothership>, NullValidator<Mothership>>();
// and the list goes on...
```

This repeated registration is, of course, not very practical. We might be tempted to again fix this as follows:

```
container.Register(typeof(IValidate<>), typeof(NullValidator<>));
```

This willl however not work, because this registration will try to map any closed *IValidate<T>* abstraction to the *NullValidator<T>* implementation, but other registrations (such as *ProductValidator* and *OrderValidator*) already exist.

What we need here is to make *NullValidator<T>* as fallback registration and Simple Injector allows this using the **RegisterConditional** method overloads:

```
container.RegisterConditional(typeof(IValidate<>), typeof(NullValidator<>),
    c => !c.Handled);
```

The result of this registration is exactly as you would have expected to see from the individual registrations above. Each request for *IValidate<Department>*, for example, will return a *NullValidator<Department>* instance each time. The **RegisterConditional** is supplied with a predicate. In this case the predicate checks whether there already is a different registration that handles the requested service type. In that case the predicate returns *false* and the registration is not applied.

This predicate can also be used to apply types conditionally based on a number of contextual arguments. Here's an example:

```
container.RegisterConditional(typeof(IValidator<>), typeof(LeftValidator<>),
    c => c.ServiceType.GetGenericArguments().Single().Namespace.Contains("Left"));

container.RegisterConditional(typeof(IValidator<>), typeof(RightValidator<>),
    c => c.ServiceType.GetGenericArguments().Single().Namespace.Contains("Right"));
```

Simple Injector protects you from defining invalid registrations by ensuring that given the registrations do not overlap. Building on the last code snippet, imagine accidentally defining a type in the namespace "MyCompany.LeftRight". In this case both open-generic implementations would apply, but Simple Injector will never silently pick one. It will throw an exception instead.

As discussed before, the **PredicateContext.Handled** property can be used to implement a fallback mechanism. A more complex example is given below:

```
container.RegisterConditional(typeof(IRepository<>), typeof(ReadOnlyRepository<>),
    c => typeof(IReadOnlyEntity).IsAssignableFrom(
        c.ServiceType.GetGenericArguments()[0]));

container.RegisterConditional(typeof(IRepository<>), typeof(ReadWriteRepository<>),
    c => !c.Handled);
```

In the case above we tell Simple Injector to only apply the *ReadOnlyRepository<T>* registration in case the given *T* implements *IReadOnlyEntity*. Although applying the predicate can be useful, in this particular case it's better to apply a generic type constraint to *ReadOnlyRepository<T>*. Simple Injector will automatically apply the registered type conditionally based on it generic type constraints. So if we apply the generic type constraint to the *ReadOnlyRepository<T>* we can remove the predicate:

```
class ReadOnlyRepository<T> : IRepository<T> where T : IReadOnlyEntity { }

container.Register(typeof(IRepository<>), typeof(ReadOnlyRepository<>));
container.RegisterConditional(typeof(IRepository<>), typeof(ReadWriteRepository<>),
    c => !c.Handled);
```

The final option in Simple Injector is to supply the **Register** or **RegisterConditional** methods with a partially-closed generic type:

```
// SomeValidator<List<T>>
var partiallyClosedType = typeof(SomeValidator<>).MakeGenericType(typeof(List<>));
container.Register(typeof(IValidator<>), partiallyClosedType);
```

The type *SomeValidator<List<T>>* is called *partially-closed*, since although its generic type argument has been filled in with a type, it still contains a generic type argument. Simple Injector will be able to apply these constraints, just as it handles any other generic type constraints.

Mixing collections of open-generic and non-generic components

The **Register** overload that take in a list of assemblies only select non-generic implementations of the given open-generic type. Open-generic implementations are skipped, because they often need special attention.

To register collections that contain both non-generic and open-generic components a **RegisterCollection** overload is available that accept a list of Type instances. For instance:

```
container.RegisterCollection(typeof(IValidator<>), new[] {
    typeof(DataAnnotationsValidator<>), // open generic
    typeof(CustomerValidator), // implements IValidator<Customer>
    typeof(GoldCustomerValidator), // implements IValidator<Customer>
    typeof(EmployeeValidator), // implements IValidator<Employee>
    typeof(OrderValidator) // implements IValidator<Order>
});
```

In the previous example a set of *IValidator<T>* implementations is supplied to the **RegisterCollection** overload. This list contains one generic implementation, namely *DataAnnotationsValidator<T>*. This leads to a registration that is equivalent to the following manual registration:

```
container.RegisterCollection<IValidator<Customer>>(
    typeof(DataAnnotationsValidator<Customer>),
    typeof(CustomerValidator),
    typeof(GoldCustomerValidator));

container.RegisterCollection<IValidator<Employee>>(
    typeof(DataAnnotationsValidator<Employee>),
    typeof(EmployeeValidator));

container.RegisterCollection<IValidator<Order>>(
    typeof(DataAnnotationsValidator<Order>),
    typeof(OrderValidator));
```

In other words, the supplied non-generic types are grouped by their closed *IValidator<T>* interface and the *DataAnnotationsValidator<T>* is applied to every group. This leads to three separate *IEnumerable<IValidator<T>>* registrations. One for each closed-generic *IValidator<T>* type.

Note: **RegisterCollection** is guaranteed to preserve the order of the types that you supply.

But besides these three *IEnumerable<IValidator<T>>* registrations, an invisible fourth registration is made. This is a registration that hooks onto the **unregistered type resolution** event and this will ensure that any time an *IEnumerable<IValidator<T>>* for a *T* that is anything other than *Customer*, *Employee* and *Order*, an *IEnumerable<IValidator<T>>* is returned that contains the closed-generic versions of the supplied open-generic types; *DataAnnotationsValidator<T>* in the given example.

Note: This will work equally well when the open generic types contain type constraints. In that case those types will be applied conditionally to the collections based on their generic type constraints.

In most cases however, manually supplying the **RegisterCollection** with a list of types leads to hard to maintain configurations, since the registration needs to be changed for each new validator we add to the system. Instead we can make use of one of the **RegisterCollection** overloads that accepts a list of assemblies and append the open generic type separately:

```
// Extension method from the SimpleInjector.Advanced namespace.
container.AppendToCollection(typeof(IValidator<>), typeof(DataAnnotationsValidator<>
↪));
```

```
container.RegisterCollection(typeof(IValidator<>),
    new[] { typeof(IValidator<>).Assembly });
```

Warning: This **RegisterCollection** overload will request all the types from the supplied *Assembly* instances. The CLR however does not give *any* guarantees what so ever about the order in which these types are returned. Don't be surprised if the order of these types in the collection change after a recompile or an application restart. In case strict ordering is required, use the **GetTypesToRegister** method (as explained below) and order types manually.

Alternatively, we can make use of the Container's **GetTypesToRegister** to find the types for us:

```
var typesToRegister = container.GetTypesToRegister(
    typeof(IValidator<>),
    new[] { typeof(IValidator<>).Assembly) },
    new TypesToRegisterOptions {
        IncludeGenericTypeDefinitions = true,
        IncludeComposites = false,
    });

container.RegisterCollection(typeof(IValidator<>), typesToRegister);
```

The **Register** and **RegisterCollection** overloads that accept a list of assemblies use this **GetTypesToRegister** method internally as well. Each however use their own **TypesToRegisterOptions** configuration.

Unregistered type resolution

Unregistered type resolution is the ability to get notified by the container when a type that is currently unregistered in the container, is requested for the first time. This gives the user (or extension point) the chance of registering that type. Simple Injector supports this scenario with the ResolveUnregisteredType[86] event. Unregistered type resolution enables many advanced scenarios.

For more information about how to use this event, please take a look at the ResolveUnregisteredType event documentation[87] in the reference library[88].

Context based injection

Context based injection is the ability to inject a particular dependency based on the context it lives in (or change the implementation based on the type it is injected into). Simple Injector contains the **RegisterConditional** method overloads that enable context based injection.

Note: In many cases context based injection is not the best solution, and the design should be reevaluated. In some narrow cases however it can make sense.

One of the simplest use cases for **RegisterConditional** is to select an implementation depending on the consumer a dependency is injected into. Take a look at the following registrations for instance:

```
container.RegisterConditional<ILogger, NullLogger>(
    c => c.Consumer.ImplementationType == typeof(HomeController));
container.RegisterConditional<ILogger, FileLogger>(
    c => c.Consumer.ImplementationType == typeof(UsersController));
container.RegisterConditional<ILogger, DatabaseLogger>(c => !c.Handled);
```

[86] https://simpleinjector.org/ReferenceLibrary/?topic=html/E_SimpleInjector_Container_ResolveUnregisteredType.htm
[87] https://simpleinjector.org/ReferenceLibrary/?topic=html/E_SimpleInjector_Container_ResolveUnregisteredType.htm
[88] https://simpleinjector.org/ReferenceLibrary/

Here we register three implementations, namely *NullLogger*, *FileLogger* and *DatabaseLogger*, all of which implement *ILogger*. The registrations are made using a predicate (lambda) describing for which condition they hold. The *NullLogger* will only be injected into the *HomeController* and the *FileLogger* will only be injected into the *UsersController*. The *DatabaseLogger* on the other hand is configured as fallback registration and will be injected in all other consumers.

Simple Injector will process conditional registrations in the order in which they are made. This means that fallback registrations, such as for the previous *DatabaseLogger*, should be made last. Simple Injector will always call all predicates to ensure no overlapping registrations are made. In case there are multiple conditional registrations that can be applied, Simple Injector will throw an exception.

Note: The predicates are only used during object graph compilation and the predicate's result is burned in the structure of returned object graph. For a requested type, the exact same graph will be created on every subsequent call. This disallows changing the graph based on runtime conditions.

A very common scenario is to base the type of the injected dependency on the type of the consumer. Take for instance the following *ILogger* interface with a generic *Logger<T>* class that needs to be injected into several consumers.

```
public interface ILogger { }

public class Logger<T> : ILogger { }

public class Consumer1 {
    public Consumer1(ILogger logger) { }
}

public class Consumer2 {
    public Consumer2(ILogger logger) { }
}
```

In this case we want to inject a *Logger<Consumer1>* into *Consumer1* and a *Logger<Consumer2>* into *Consumer2*. By using the **RegisterConditional** overload that accepts a *implementation type factory delegate*, we can accomplish this as follows:

```
container.RegisterConditional(
    typeof(ILogger),
    c => typeof(Logger<>).MakeGenericType(c.Consumer.ImplementationType),
    Lifestyle.Singleton,
    c => true);
```

In the previous code snippet we supply the **RegisterConditional** method with a lambda presenting a *Func<TypeFactoryContext, Type>* delegate that allows building the exact implementation type based on contextual information. In this case we use the implementation type of the consuming component to build the correct closed *Logger<T>* type. We also supply the method with a predicate, but in this case we make the registration unconditional by returning *true* from the predicate, meaning that this is the only registration for *ILogger*.

Note: Although building a generic type using MakeGenericType is relatively slow, the call to the *Func<TypeFactoryContext, Type>* delegate itself has a one-time cost. The factory delegate will only be called a finite number of times. After an object graph has been built, the delegate will not be called again when that same object graph is resolved.

Note: Even though the use of a generic *Logger<T>* is a common design (with log4net as the grand godfather of this design), doesn't always make it a good design. The need for having the logger contain information about its parent type, might indicate design problems. If you're doing this, please take a look at this Stackoverflow answer[89]. It talks about logging in conjunction with the SOLID design principles.

[89] https://stackoverflow.com/a/9915056/264697

Property injection

Simple Injector does not inject any properties into types that get resolved by the container. In general there are two ways of doing property injection, and both are not enabled by default for reasons explained below.

Implicit property injection

Some containers (such as Castle Windsor) implicitly inject public writable properties by default for any instance you resolve. They do this by mapping those properties to configured types. When no such registration exists, or when the property doesn't have a public setter, the property will be skipped. Simple Injector does not do implicit property injection, and for good reason. We think that **implicit property injection** is simply too uuhh... implicit :-). Silently skipping properties that can't be mapped can lead to a DI configuration that can't be easily verified and can therefore result in an application that fails at runtime instead of failing when the container is verified.

Explicit property injection

We strongly feel that explicit property injection is a much better way to go. With explicit property injection the container is forced to inject a property and the process will fail immediately when a property can't be mapped or injected. Some containers (such as Unity and Ninject) allow explicit property injection by allowing properties to be marked with attributes that are defined by the DI library. Problem with this is that this forces the application to take a dependency on the library, which is something that should be prevented.

Because Simple Injector does not encourage its users to take a dependency on the container (except for the startup path of course), Simple Injector does not contain any attributes that allow explicit property injection and it can therefore not explicitly inject properties out-of-the-box.

Besides this, the use of property injection should be very exceptional and in general constructor injection should be used in the majority of cases. If a constructor gets too many parameters (constructor over-injection anti-pattern), it is an indication of a violation of the Single Responsibility Principle[90] (SRP). SRP violations often lead to maintainability issues. So instead of patching constructor over-injection with property injection, the root cause should be analyzed and the type should be refactored, probably with Facade Services[91]. Another common reason to use properties is because those dependencies are optional. Instead of using optional property dependencies, best practice is to inject empty implementations (a.k.a. Null Object pattern[92]) into the constructor.

Enabling property injection

Simple Injector contains two ways to enable property injection. First of all the *RegisterInitializer<T>* method can be used to inject properties (especially configuration values) on a per-type basis. Take for instance the following code snippet:

```
container.RegisterInitializer<HandlerBase>(handlerToInitialize => {
    handlerToInitialize.ExecuteAsynchronously = true;
});
```

In the previous example an *Action<T>* delegate is registered that will be called every time the container creates a type that inherits from *HandlerBase*. In this case, the handler will set a configuration value on that class.

Note: although this method can also be used injecting services, please note that the *Diagnostic Services* will be unable to see and analyze that dependency.

[90] https://en.wikipedia.org/wiki/Single_responsibility_principle
[91] http://blog.ploeh.dk/2010/02/02/RefactoringtoAggregateServices/
[92] https://en.wikipedia.org/wiki/Null_Object_pattern

IPropertySelectionBehavior

The second way to inject properties is by implementing a custom **IPropertySelectionBehavior**. The *property selection behavior* is a general extension point provided by the container, to override the library's default behavior (which is to *not* inject properties). The following example enables explicit property injection using attributes, using the *ImportAttribute* from the *System.ComponentModel.Composition.dll*:

```
using System;
using System.ComponentModel.Composition;
using System.Linq;
using System.Reflection;
using SimpleInjector.Advanced;

class ImportPropertySelectionBehavior : IPropertySelectionBehavior {
    public bool SelectProperty(Type implementationType, PropertyInfo prop) =>
        prop.GetCustomAttributes(typeof(ImportAttribute)).Any();
}
```

The previous class can be registered as follows:

```
var container = new Container();
container.Options.PropertySelectionBehavior = new ImportPropertySelectionBehavior();
```

This enables explicit property injection on all properties that are marked with the [Import] attribute and an exception will be thrown when the property cannot be injected for whatever reason.

Tip: Properties injected by the container through the **IPropertySelectionBehavior** will be analyzed by the *Diagnostic Services*.

Note: The **IPropertySelectionBehavior** extension mechanism can also be used to implement implicit property injection. There's an example of this[93] in the source code. Doing so however is not advised because of the reasons given above.

Covariance and Contravariance

Since version 4.0 of the .NET framework, the type system allows Covariance and Contravariance in Generics[94] (especially interfaces and delegates). This allows for instance, to use a *IEnumerable<string>* as an *IEnumerable<object>* (covariance), or to use an *Action<object>* as an *Action<string>* (contravariance).

In some circumstances, the application design can benefit from the use of covariance and contravariance (or variance for short) and it would be beneficial if the container returned services that were 'compatible' with the requested service, even when the requested service type itself is not explicitly registered. To stick with the previous example, the container could return an *IEnumerable<string>* even when an *IEnumerable<object>* is requested.

When resolving a collection, Simple Injector will resolve all assignable (variant) implementations of the requested service type as part of the requested collection.

Take a look at the following application design around the *IEventHandler<in TEvent>* interface:

```
public interface IEventHandler<in TEvent> {
    void Handle(TEvent e);
}

public class CustomerMovedEvent {
```

[93] https://github.com/simpleinjector/SimpleInjector/blob/master/SimpleInjector.CodeSamples/ImplicitPropertyInjectionExtensions.cs
[94] https://msdn.microsoft.com/en-us/library/dd799517.aspx

```
    public readonly Guid CustomerId;
    public CustomerMovedEvent(Guid customerId) {
        this.CustomerId = customerId;
    }
}

public class CustomerMovedAbroadEvent : CustomerMovedEvent {
    public CustomerMovedEvent(Guid customerId) : base(customerId) { }
}

public class SendFlowersToMovedCustomer : IEventHandler<CustomerMovedEvent> {
    public void Handle(CustomerMovedEvent e) { ... }
}

public class WarnShippingDepartmentAboutMove : IEventHandler<CustomerMovedAbroadEvent>
↪ {
    public void Handle(CustomerMovedAbroadEvent e) { ... }
}
```

The design contains two event classes *CustomerMovedEvent* and *CustomerMovedAbroadEvent* (where *CustomerMovedAbroadEvent* inherits from *CustomerMovedEvent*) and two concrete event handlers *SendFlowersToMovedCustomer* and *WarnShippingDepartmentAboutMove*. These classes can be registered using the following registration:

```
// Configuration
container.RegisterCollection(typeof(IEventHandler<>),
    new[] { typeof(IEventHandler<>).Assembly });

// Usage
var handlers = container.GetAllInstances<IEventHandler<CustomerMovedAbroadEvent>>();

foreach (var handler in handlers) {
    Console.WriteLine(handler.GetType().Name);
}
```

With the given classes, the code snippet above will give the following output:

```
SendFlowersToMovedCustomer
WarnShippingDepartmentAboutMove
```

Although we requested all registrations for *IEventHandler<CustomerMovedAbroadEvent>*, the container returned *IEventHandler<CustomerMovedEvent>* and *IEventHandler<CustomerMovedAbroadEvent>*. Simple Injector did this because the *IEventHandler<in TEvent>* interface was defined with the ***in*** keyword, which makes *IEventHandler<SendFlowerToMovedCustomer>* assignable to *IEventHandler<CustomerMovedAbroadEvent>* (since *CustomerMovedAbroadEvent* inherits from *CustomerMovedEvent*, *SendFlowerToMovedCustomer* can also process *CustomerMovedAbroadEvent* events).

Tip: If you don't want Simple Injector to resolve variant registrations remove the **in** and **out** keywords from the interface definition. i.e. the **in** and **out** keywords are the trigger for Simple Injector to apply variance.

Tip: Don't mark generic type arguments with **in** and **out** keywords by default, even if Resharper tells you to. Most of the generic abstractions you define will always have exactly one non-generic implementation but marking the interface with **in** and **out** keywords communicates that covariance and contravariance is expected and there could therefore be multiple applicable implementations. This will confuse the reader of your code. Only apply these keywords if variance is actually required. You should typically not use variance when defining *ICommandHandler<TCommand>* or *IQueryHandler<TQuery, TResult>*, but it might make sense for *IEventHandler<in TEvent>* and *IValidator<in T>*.

Note: Simple Injector only resolves variant implementations for collections that are registered using the *RegisterCollection* overloads. In the screnario you are resolving a single instance using *GetInstance<T>* then Simple Injector

will not return an assignable type, even if the exact type is not registered, because this could easily lead to ambiguity; Simple Injector will not know which implementation to select.

Registering plugins dynamically

Applications with a plugin architecture often allow special plugin assemblies to be dropped in a special folder and to be picked up by the application, without the need of a recompile. Although Simple Injector has no out of the box support for this, registering plugins from dynamically loaded assemblies can be implemented in a few lines of code. Here is an example:

```
string pluginDirectory =
    Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "Plugins");

var pluginAssemblies =
    from file in new DirectoryInfo(pluginDirectory).GetFiles()
    where file.Extension.ToLower() == ".dll"
    select Assembly.Load(AssemblyName.GetAssemblyName(file.FullName));

container.RegisterCollection<IPlugin>(pluginAssemblies);
```

The given example makes use of an *IPlugin* interface that is known to the application, and probably located in a shared assembly. The dynamically loaded plugin .dll files can contain multiple classes that implement *IPlugin*, and all publicly exposed concrete types that implement *IPlugin* will be registered using the **RegisterCollection** method and can get resolved using the default auto-wiring behavior of the container, meaning that the plugin must have a single public constructor and all constructor arguments must be resolvable by the container. The plugins can get resolved using *container.GetAllInstances<IPlugin>()* or by adding an *IEnumerable<IPlugin>* argument to a constructor.

CHAPTER 8

Aspect-Oriented Programming

The SOLID[95] principles give us important guidance when it comes to writing maintainable software. The 'O' of the 'SOLID' acronym stands for the Open/closed Principle[96] which states that classes should be open for extension, but closed for modification. Designing systems around the Open/closed principle means that new behavior can be plugged into the system, without the need to change any existing parts, making the chance of breaking existing code much smaller and prevent having to make sweeping changes throughout the code base. This way of working is often referred to as Aspect-oriented programming.

Aspect-oriented programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It allows new behavior to be plugged in or changed, without having to change or even recompile existing code. Simple Injector's main support for AOP is by the use of decorators. Besides decorators, one can also plugin interception using a dynamic proxy framework.

This chapter discusses the following subjects:

- *Decoration*
- *Interception using Dynamic Proxies*

Decoration

The best way to add new functionality (such as cross-cutting concerns[97]) to classes is by the use of the decorator pattern[98]. The decorator pattern can be used to extend (decorate) the functionality of a certain object at run-time. Especially when using generic interfaces, the concept of decorators gets really powerful. Take for instance the examples given in the *Registration of open generic types* section or for instance the use of an generic *ICommandHandler<TCommand>* interface.

Tip: This article[99] describes an architecture based on the use of the *ICommandHandler<TCommand>* interface.

[95] https://en.wikipedia.org/wiki/SOLID
[96] https://en.wikipedia.org/wiki/Open/closed_principle
[97] https://en.wikipedia.org/wiki/Cross-cutting_concern
[98] https://en.wikipedia.org/wiki/Decorator_pattern
[99] https://cuttingedge.it/blogs/steven/pivot/entry.php?id=91

Take the plausible scenario where we want to validate all commands that get executed by an *ICommandHandler<TCommand>* implementation. The Open/Closed principle states that we want to do this, without having to alter each and every implementation. We can do this using a (single) decorator:

```
public class ValidationCommandHandlerDecorator<TCommand> : ICommandHandler<TCommand> {
    private readonly IValidator validator;
    private readonly ICommandHandler<TCommand> decoratee;

    public ValidationCommandHandlerDecorator(IValidator validator,
        ICommandHandler<TCommand> decoratee) {
        this.validator = validator;
        this.decoratee = decoratee;
    }

    void ICommandHandler<TCommand>.Handle(TCommand command) {
        // validate the supplied command (throws when invalid).
        this.validator.ValidateObject(command);

        // forward the (valid) command to the real command handler.
        this.decoratee.Handle(command);
    }
}
```

The *ValidationCommandHandlerDecorator<TCommand>* class is an implementation of the *ICommandHandler<TCommand>* interface, but it also wraps / decorates an *ICommandHandler<TCommand>* instance. Instead of injecting the real implementation directly into a consumer, we can (let Simple Injector) inject a validator decorator that wraps the real implementation.

The *ValidationCommandHandlerDecorator<TCommand>* depends on an *IValidator* interface. An implementation that used Microsoft Data Annotations might look like this:

```
using System.ComponentModel.DataAnnotations;

public class DataAnnotationsValidator : IValidator {
    void IValidator.ValidateObject(object instance) {
        var context = new ValidationContext(instance, null, null);

        // Throws an exception when instance is invalid.
        Validator.ValidateObject(instance, context, validateAllProperties: true);
    }
}
```

The implementations of the *ICommandHandler<T>* interface can be registered using the **Register** method overload that takes in a list of assemblies:

```
container.Register(
    typeof(ICommandHandler<>),
    new[] { typeof(ICommandHandler<>).Assembly });
```

By using the following method, you can wrap the *ValidationCommandHandlerDecorator<TCommand>* around each and every *ICommandHandler<TCommand>* implementation:

```
container.RegisterDecorator(
    typeof(ICommandHandler<>),
    typeof(ValidationCommandHandlerDecorator<>));
```

Multiple decorators can be wrapped by calling the **RegisterDecorator** method multiple times, as the following registration shows:

```
container.Register(
    typeof(ICommandHandler<>),
    new[] { typeof(ICommandHandler<>).Assembly });

container.RegisterDecorator(
    typeof(ICommandHandler<>),
    typeof(TransactionCommandHandlerDecorator<>));

container.RegisterDecorator(
    typeof(ICommandHandler<>),
    typeof(DeadlockRetryCommandHandlerDecorator<>));

container.RegisterDecorator(
    typeof(ICommandHandler<>),
    typeof(ValidationCommandHandlerDecorator<>));
```

The decorators are applied in the order in which they are registered, which means that the first decorator (*TransactionCommandHandlerDecorator<T>* in this case) wraps the real instance, the second decorator (*DeadlockRetryCommandHandlerDecorator<T>* in this case) wraps the first decorator, and so on.

Applying Decorators conditionally

There's an overload of the **RegisterDecorator** available that allows you to supply a predicate to determine whether that decorator should be applied to a specific service type. Using a given context you can determine whether the decorator should be applied. Here is an example:

```
container.RegisterDecorator(
    typeof(ICommandHandler<>),
    typeof(AccessValidationCommandHandlerDecorator<>),
    context => typeof(IAccessRestricted).IsAssignableFrom(
        context.ServiceType.GetGenericArguments()[0]));
```

The given context contains several properties that allows you to analyze whether a decorator should be applied to a given service type, such as the current closed generic service type (using the *ServiceType* property) and the concrete type that will be created (using the *ImplementationType* property). The predicate will (under normal circumstances) be called only once per closed generic type, so there is no performance penalty for using it.

Applying Decorators conditionally using type constraints

The previous example shows the conditional registration of the *AccessValidationCommandHandlerDecorator<T>* decorator. It is applied in case the closed *TCommand* type (of *ICommandHandler<TCommand>*) implements the *IAccessRestricted* interface.

Simple Injector will automatically apply decorators conditionally based on defined generic type constraints[100]. We can therefore define the *AccessValidationCommandHandlerDecorator<T>* with a generic type constraint, as follows:

```
class AccessValidationCommandHandlerDecorator<TCommand> : ICommandHandler<TCommand>
    where TCommand : IAccessRestricted
{
    private readonly ICommandHandler<TCommand> decoratee;

    public AccessValidationCommandHandlerDecorator(ICommandHandler<TCommand>␣
↪decoratee){
```

[100] https://msdn.microsoft.com/en-us/library/d5x73970.aspx

```
        this.decoratee = decoratee;
    }

    void ICommandHandler<TCommand>.Handle(TCommand command) {
        // Do access validation
        this.decoratee.Handle(command);
    }
}
```

Since Simple Injector natively understands generic type constraints, we can reduce the previous registration to the following:

```
container.RegisterDecorator(
    typeof(ICommandHandler<>),
    typeof(AccessValidationCommandHandlerDecorator<>));
```

The use of generic type constraints has many advantages:

- It allows constraints to be specified exactly once, in the place it often makes most obvious, i.e. the decorator itself.
- It allows constraints to be specified in the syntax you are used to the most, i.e. C#.
- It allows constraints to be specified in a very succinct manner compared to the verbose, error prone and often hard to read syntax of the reflection API (the previous examples already shown this).
- It allows decorator to be simplified, because of the added compile time support.

Obviously there are cases where these conditions can't or shouldn't be defined using generic type constraints. The following code example shows a registration that can't be expressed using generic type constraints:

```
container.RegisterDecorator(
    typeof(ICommandHandler<>),
    typeof(AccessValidationCommandHandlerDecorator<>),
    c => c.ImplementationType.GetCustomAttributes(typeof(AccessAttribute)).Any());
```

This registration applies the decorator conditionally based on an attribute on the (initially) decorated handler type. There is obviously no way to express this using generic type constraints, so we will have to fallback to the predicate syntax.

Decorators with Func<T> decoratee factories

There are certain scenarios where it is necessary to postpone the building of part of an object graph. For instance when a service needs to control the lifetime of a dependency, needs multiple instances, when instances need to be *executed on a different thread*, or when instances need to be created within a certain *scope* or context (e.g. security).

You can easily delay the building of part of the graph by depending on a factory; the factory allows building that part of the object graph to be postponed until the moment the type is actually required. However, when working with decorators, injecting a factory to postpone the creation of the decorated instance will not work. This is best demonstrated with an example.

Take for instance an *AsyncCommandHandlerDecorator<T>* that executes a command handler on a different thread. We could let the *AsyncCommandHandlerDecorator<T>* depend on a *CommandHandlerFactory<T>*, and let this factory call back into the container to retrieve a new *ICommandHandler<T>* but this would fail, since requesting an *ICommandHandler<T>* would again wrap the new instance with a *AsyncCommandHandlerDecorator<T>* and we'd end up recursively creating the same instance type again and again resulting in an endless loop.

This particular scenario is really hard to solve without library support and as such Simple Injector allows injecting a *Func<T>* delegate into registered decorators. This delegate functions as a factory for the creation of the decorated instance and avoids the recursive decoration explained above.

Taking the same *AsyncCommandHandlerDecorator<T>* as an example, it could be implemented as follows:

```
public class AsyncCommandHandlerDecorator<T> : ICommandHandler<T> {
    private readonly Func<ICommandHandler<T>> decorateeFactory;

    public AsyncCommandHandlerDecorator(Func<ICommandHandler<T>> decorateeFactory) {
        this.decorateeFactory = decorateeFactory;
    }

    public void Handle(T command) {
        // Execute on different thread.
        ThreadPool.QueueUserWorkItem(state => {
            try {
                // Create new handler in this thread.
                ICommandHandler<T> handler = this.decorateeFactory.Invoke();
                handler.Handle(command);
            } catch (Exception ex) {
                // log the exception
            }
        });
    }
}
```

This special decorator is registered just as any other decorator:

```
container.RegisterDecorator(
    typeof(ICommandHandler<>),
    typeof(AsyncCommandHandlerDecorator<>),
    c => c.ImplementationType.Name.StartsWith("Async"));
```

The *AsyncCommandHandlerDecorator<T>* however, has only singleton dependencies (the *Func<T>* is a singleton) and the *Func<ICommandHandler<T>>* factory always calls back into the container to register a decorated instance conforming the decoratee's lifestyle, each time it's called. If for instance the decoratee is registered as transient, each call to the factory will result in a new instance. It is therefore safe to register this decorator as a singleton:

```
container.RegisterDecorator(
    typeof(ICommandHandler<>),
    typeof(AsyncCommandHandlerDecorator<>),
    Lifestyle.Singleton,
    c => c.ImplementationType.Name.StartsWith("Async"));
```

When mixing this decorator with other (synchronous) decorators, you'll get an extremely powerful and pluggable system:

```
container.Register(
    typeof(ICommandHandler<>),
    new[] { typeof(ICommandHandler<>).Assembly });

container.RegisterDecorator(
    typeof(ICommandHandler<>),
    typeof(TransactionCommandHandlerDecorator<>));

container.RegisterDecorator(
    typeof(ICommandHandler<>),
```

```
    typeof(DeadlockRetryCommandHandlerDecorator<>));

container.RegisterDecorator(
    typeof(ICommandHandler<>),
    typeof(AsyncCommandHandlerDecorator<>),
    Lifestyle.Singleton,
    c => c.ImplementationType.Name.StartsWith("Async"));

container.RegisterDecorator(
    typeof(ICommandHandler<>),
    typeof(ValidationCommandHandlerDecorator<>));
```

This configuration has an interesting mix of decorator registrations.

1. The registration of the *AsyncCommandHandlerDecorator<T>* allows (a subset of) command handlers to be executed in the background (while any command handler with a name that does not start with 'Async' will execute synchronously)
2. Prior to this point all commands are validated synchronously (to allow communicating validation errors to the caller)
3. All handlers (sync and async) are executed in a transaction and the operation is retried when the database rolled back because of a deadlock

Warning: Please note that the previous example is just meant for educational purposes. In practice, you don't want your commands to be processed this way, since it could lead to message loss. Instead you want to use a durable queue.

Another useful application for *Func<T>* decoratee factories is when a command needs to be executed in an isolated fashion, e.g. to prevent sharing the unit of work with the request that triggered the execution of that command. This can be achieved by creating a proxy that starts a new thread-specific scope, as follows:

```
using SimpleInjector.Lifestyles;

public class ThreadScopedCommandHandlerProxy<T> : ICommandHandler<T> {
    private readonly Container container;
    private readonly Func<ICommandHandler<T>> decorateeFactory;

    public ThreadScopedCommandHandlerProxy(Container container,
        Func<ICommandHandler<T>> decorateeFactory) {
        this.container = container;
        this.decorateeFactory = decorateeFactory;
    }

    public void Handle(T command) {
        // Start a new scope.
        using (ThreadScopedLifestyle.BeginScope(container)) {
            // Create the decorateeFactory within the scope.
            ICommandHandler<T> handler = this.decorateeFactory.Invoke();
            handler.Handle(command);
        };
    }
}
```

This proxy class starts a new *thread scoped lifestyle* and resolves the decoratee within that new scope using the factory. The use of the factory ensures that the decoratee is resolved according to its lifestyle, independent of the lifestyle of our proxy class. The proxy can be registered as follows:

```
container.RegisterDecorator(
    typeof(ICommandHandler<>),
```

```
    typeof(ThreadScopedCommandHandlerProxy<>),
    Lifestyle.Singleton);
```

Note: Since the *ThreadScopedCommandHandlerProxy<T>* only depends on singletons (both the *Container* and the *Func<ICommandHandler<T>>* are singletons), it too can safely be registered as singleton.

Since a typical application will not use the thread scoped lifestyle, but would prefer a scope specific to the application type, a special *hybrid lifestyle* needs to be defined that allows object graphs to be resolved in this mixed-request scenario:

```
container.Options.DefaultScopedLifestyle = Lifestyle.CreateHybrid(
    defaultLifestyle = new ThreadScopedLifestyle(),
    fallbackLifestyle: new WebRequestLifestyle());

container.Register<IUnitOfWork, DbUnitOfWork>(Lifestyle.Scoped);
```

Obviously, if you run (part of) your commands on a background thread and also use registrations with a *scoped lifestyle* you will have a use both the *ThreadScopedCommandHandlerProxy<T>* and *AsyncCommandHandlerDecorator<T>* together which can be seen in the following configuration:

```
container.Options.DefaultScopedLifestyle = Lifestyle.CreateHybrid(
    defaultLifestyle = new ThreadScopedLifestyle(),
    fallbackLifestyle: new WebRequestLifestyle());

container.Options.DefaultScopedLifestyle = scopedLifestyle;

container.Register<IUnitOfWork, DbUnitOfWork>(Lifestyle.Scoped);
container.Register<IRepository<User>, UserRepository>(Lifestyle.Scoped);

container.Register(
    typeof(ICommandHandler<>),
    new[] { typeof(ICommandHandler<>).Assembly });

container.RegisterDecorator(
    typeof(ICommandHandler<>),
    typeof(ThreadScopedCommandHandlerProxy<>),
    Lifestyle.Singleton);

container.RegisterDecorator(
    typeof(ICommandHandler<>),
    typeof(AsyncCommandHandlerDecorator<>),
    Lifestyle.Singleton,
    c => c.ImplementationType.Name.StartsWith("Async"));
```

With this configuration all commands are executed in an isolated context and some are also executed on a background thread.

Decorated collections

When registering a decorator, Simple Injector will automatically decorate any collection with elements of that service type:

```
container.RegisterCollection<IEventHandler<CustomerMovedEvent>>(new[] {
    typeof(CustomerMovedEventHandler),
    typeof(NotifyStaffWhenCustomerMovedEventHandler)
});
```

```
container.RegisterDecorator(
    typeof(IEventHandler<>),
    typeof(TransactionEventHandlerDecorator<>),
    c => SomeCondition);
```

The previous registration registers a collection of *IEventHandler<CustomerMovedEvent>* services. Those services are decorated with a *TransactionEventHandlerDecorator<TEvent>* when the supplied predicate holds.

For collections of elements that are created by the container (container controlled), the predicate is checked for each element in the collection. For collections of uncontrolled elements (a list of items that is not created by the container), the predicate is checked once for the whole collection. This means that controlled collections can be partially decorated. Taking the previous example for instance, you could let the *CustomerMovedEventHandler* be decorated, while leaving the *NotifyStaffWhenCustomerMovedEventHandler* undecorated (determined by the supplied predicate).

When a collection is uncontrolled, it means that the lifetime of its elements are unknown to the container. The following registration is an example of an uncontrolled collection:

```
IEnumerable<IEventHandler<CustomerMovedEvent>> handlers =
    new IEventHandler<CustomerMovedEvent>[] {
        new CustomerMovedEventHandler(),
        new NotifyStaffWhenCustomerMovedEventHandler(),
    };

container.RegisterCollection<IEventHandler<CustomerMovedEvent>>(handlers);
```

Although this registration contains a list of singletons, the container has no way of knowing this. The collection could easily have been a dynamic (an ever changing) collection. In this case, the container calls the registered predicate once (and supplies the predicate with the *IEventHandler<CusotmerMovedEvent>* type) and if the predicate returns true, each element in the collection is decorated with a decorator instance.

Warning: In general you should prevent registering uncontrolled collections. The container knows nothing about them, and can't help you in doing *diagnostics*. Since the lifetime of those items is unknown, the container will be unable to wrap a decorator with a lifestyle other than transient. Best practice is to register container-controlled collections which is done by using one of the **RegisterCollection** overloads that take a collection of *System.Type* instances.

Using contextual information inside decorators

As we shown before, you can apply a decorator conditionally based on a predicate you can supply to the **RegisterDecorator** overloads:

```
container.RegisterDecorator(
    typeof(ICommandHandler<>),
    typeof(AsyncCommandHandlerDecorator<>),
    c => c.ImplementationType.Name.StartsWith("Async"));
```

Sometimes however you might want to apply a decorator unconditionally, but let the decorator act at runtime based on this contextual information. You can do this by injecting the **DecoratorContext** into the decorator's constructor as can be seem in the following example:

```
public class TransactionCommandHandlerDecorator<T> : ICommandHandler<T> {
    private readonly ITransactionBuilder builder;
    private readonly ICommandHandler<T> decoratee;
    private readonly TransactionType transactionType;
```

```
    public TransactionCommandHandlerDecorator(DecoratorContext decoratorContext,
        ITransactionBuilder builder, ICommandHandler<T> decoratee) {
        this.builder = builder;
        this.decoratee = decoratee;
        this.transactionType = decoratorContext.ImplementationType
            .GetCustomAttribute<TransactionAttribute>()
            .TransactionType;
    }

    public void Handle(T command) {
        using (var ta = this.builder.BeginTransaction(this.transactionType)) {
            this.decoratee.Handle(command);
            ta.Complete();
        }
    }
}
```

The previous code snippet shows a decorator that applies a transaction behavior to command handlers. The decorator is injected with the **DecoratorContext** class which supplies the decorator with contextual information about the other decorators in the chain and the actual implementation type. In this example the decorator expects a *TransactionAttribute* to be applied to the wrapped command handler implementation and it starts the correct transaction type based on this information. The following code snippet shows a possible command handler implementation:

```
[Transaction(TransactionType.ReadCommitted)]
public class ShipOrderHandler : ICommandHandler<ShipOrder> {
    public void Handle(ShipOrder command) {
        // Business logic here
    }
}
```

If the attribute was applied to the command class instead of the command handler, this decorator would been able to gather this information without the use of the **DecoratorContext**. This would however leak implementation details into the command, since which type of transaction a handler should run is clearly an implementation detail and is of no concern to the consumer of that command. Placing that attribute on the handler instead of the command is therefore a much more reasonable thing to do.

The decorator would also be able to get the attribute by using the injected decoratee, but this would only work when the decorator would directly wrap the handler. This would make the system quite fragile, since it would break once you start placing other decorator in between this decorator and the handler, which is a very likely thing to happen.

Applying decorators conditionally based on consumer

The previous examples showed how to apply a decorator conditionally based on information about its dependencies, such as the decorators that it wraps and the wrapped real implementation. Another option is to make decisions based on the consuming components; the components the decorator is injected into.

Although the **RegisterDecorator** methods don't have any built-in support for this, this behavior can be achieved by using the **RegisterConditional** methods. For instance:

```
container.RegisterConditional<IMailSender, AsyncMailSenderDecorator>(
    c => c.Consumer.ImplementationType == typeof(UserController));
container.RegisterConditional<IMailSender, BufferedMailSenderDecorator>(
    c => c.Consumer.ImplementationType == typeof(EmailBatchProcessor));

container.RegisterConditional<IMailSender, SmtpMailSender>(c => !c.Handled);
```

Here we use **RegisterConditional** to register two decorators. Both decorator will wrap the *SmtpMailSender* that is registered last. The *AsyncMailSenderDecorator* is wrapped around the *SmtpMailSender* in case it is injected into the *UserController*, while the *BufferedMailSenderDecorator* is wrapped when injected into the *EmailBatchProcessor*. Note that the *SmtpMailSender* is registered as conditional as well, and is registered as fallback registration using **!c.Handled**, which basically means that in case no other registration applies, that registration is used.

Decorator registration factories

In some advanced scenarios, it can be useful to depend the actual decorator type based on some contextual information. There is a **RegisterDecorator** overload that accepts a factory delegate that allows building the exact decorator type based on the actual type being decorated.

Take the following registration for instance:

```
container.RegisterDecorator(
    typeof(IEventHandler<>),
    factoryContext => typeof(LoggingEventHandlerDecorator<,>).MakeGenericType(
        typeof(LoggingEventHandler<,>).GetGenericArguments().First(),
        factoryContext.ImplementationType),
    Lifestyle.Transient,
    predicateContext => true);
```

This example registers the *LoggingEventHandlerDecorator<TEvent, TLogTarget>* decorator for the *IEventHandler<TEvent>* abstraction. The supplied factory delegate builds up a partially-closed generic type by filling in the *TLogTarget* argument, where the *TEvent* is left 'open'. This is done by requesting the first generic type argument (the *TEvent*) from the open-generic *LoggingEventHandler<,>* type itself and using the **ImplementationType** as second argument. This means that when this decorator is wrapped around a type called *CustomerMovedEventHandler*, the factory method will create the type *LoggingEventHandler<TEvent, CustomerMovedEventHandler>*. In other words, the second argument is a concrete type (and thus closed), while the first argument is still a blank.

When a closed version of *IEventHandler<TEvent>* is requested later on, Simple Injector will know how to fill in the blank with the correct type for this *TEvent* argument.

Tip: Simple Injector doesn't care in which order you define your generic type arguments, nor how you name them; it will be able to figure out the correct type to build any way.

Note: The type factory delegate is typically called once per closed-type and the result is burned in the compiled object graph. You can't use this delegate to make runtime decisions.

Interception using Dynamic Proxies

Interception is the ability to intercept a call from a consumer to a service, and add or change behavior. The decorator pattern[101] describes a form of interception, but when it comes to applying cross-cutting concerns, you might end up writing decorators for many service interfaces, but with the exact same code. If this is happening, it's time to take a close look at your design. If for what ever reason, it's impossible for you to make the required improvements to your design, your second best bet is to explore the possibilities of interception through dynamic proxies.

Warning: Simple Injector has *no out-of-the-box support for interception* because the use of interception is an indication of a sub optimal design and we are keen on pushing developers into best practices. Whenever possible, choose to improve your design to make decoration possible.

Using the *Interception extensions* code snippets, you can add the ability to do interception with Simple Injector. Using the given code, you can for instance define a *MonitoringInterceptor* that allows logging the execution time of the called service method:

[101] https://en.wikipedia.org/wiki/Decorator_pattern

```
private class MonitoringInterceptor : IInterceptor {
    private readonly ILogger logger;

    // Using constructor injection on the interceptor
    public MonitoringInterceptor(ILogger logger) {
        this.logger = logger;
    }

    public void Intercept(IInvocation invocation) {
        var watch = Stopwatch.StartNew();

        // Calls the decorated instance.
        invocation.Proceed();

        var decoratedType = invocation.InvocationTarget.GetType();

        this.logger.Log(string.Format("{0} executed in {1} ms.",
            decoratedType.Name, watch.ElapsedMiliseconds));
    }
}
```

This interceptor can be registered to be wrapped around a concrete implementation. Using the given extension methods, this can be done as follows:

```
container.InterceptWith<MonitoringInterceptor>(type => type ==␣
↪typeof(IUserRepository));
```

This registration ensures that every time an *IUserRepository* interface is requested, an interception proxy is returned that wraps that instance and uses the *MonitoringInterceptor* to extend the behavior.

The current example doesn't add much compared to simply using a decorator. When having many interface service types that need to be decorated with the same behavior however, it gets different:

```
container.InterceptWith<MonitoringInterceptor>(t => t.Name.EndsWith("Repository"));
```

Note: The *Interception extensions* code snippets use .NET's *System.Runtime.Remoting.Proxies.RealProxy* class to generate interception proxies. The *RealProxy* only allows to proxy interfaces.

Note: the interfaces in the given *Interception extensions* code snippets are a simplified version of the Castle Project interception facility. If you need to create lots different interceptors, you might benefit from using the interception abilities of the Castle Project. Also please note that the given snippets use dynamic proxies to do the interception, while Castle uses lightweight code generation (LCG). LCG allows much better performance than the use of dynamic proxies. Please see this stackoverflow q/a[102] for an implementation for Castle Windsor.

Note: Don't use interception for intercepting types that all implement the same generic interface, such as *ICommandHandler<T>* or *IValidator<T>*. Try using decorator classes instead, as shown in the *Decoration* section on this page.

[102] https://stackoverflow.com/questions/24513530/using-simple-injector-with-castle-proxy-interceptor

CHAPTER 9

Extensibility Points

Simple Injector allows much of its default behavior to be changed or extended. This chapter describes the available extension points and shows examples of how to use them. Do note that in most cases we advice developers to stick with the default behavior, because this behavior is based on best practices.

- *Overriding Constructor Resolution Behavior*
- *Property Injection*
- *Overriding Parameter Injection Behavior*
- *Resolving Unregistered Types*
- *Overriding Lifestyle Selection Behavior*
- *Intercepting the Creation of Types*
- *Building up external instances*
- *Interception of Resolved Object Graphs*

Overriding Constructor Resolution Behavior

Out of the box, Simple Injector only allows the creation of classes that contain a single public constructor. This behavior is chosen deliberately because having multiple constructors is an anti-pattern[103].

There are some exceptional circumstances though, where we don't control the amount of public constructors a type has. Code generators for instance, can have this annoying side effect. Earlier versions of the T4MVC[104] template for instance did this.

In these rare cases we need to override the way Simple Injector does its constructor overload resolution. This can be done by creating custom implementation of **IConstructorResolutionBehavior**. The default behavior can be replaced by setting the *Container.Options.ConstructorResolutionBehavior* property.

[103] https://cuttingedge.it/blogs/steven/pivot/entry.php?id=97
[104] https://github.com/T4MVC/T4MVC

```
public interface IConstructorResolutionBehavior {
    ConstructorInfo GetConstructor(Type implementationType);
}
```

Simple Injector will call into the registered **IConstructorResolutionBehavior** when the type is registered to allow the **IConstructorResolutionBehavior** implementation to verify the type. The implementation is called again when the registered type is resolved for the first time.

The following example changes the constructor resolution behavior to always select the constructor with the most parameters (the greediest constructor):

```
// Custom constructor resolution behavior
public class GreediestConstructorBehavior : IConstructorResolutionBehavior {
    public ConstructorInfo GetConstructor(Type implementationType) => (
        from ctor in implementationType.GetConstructors()
        orderby ctor.GetParameters().Length descending
        select ctor)
        .First();
}

// Usage
var container = new Container();
container.Options.ConstructorResolutionBehavior = new GreediestConstructorBehavior();
```

The following bit more advanced example changes the constructor resolution behavior to always select the constructor with the most parameters from the list of constructors with only resolvable parameters:

```
public class MostResolvableParametersConstructorResolutionBehavior
    : IConstructorResolutionBehavior {
    private readonly Container container;

    public MostResolvableParametersConstructorResolutionBehavior(Container container)
↪{
        this.container = container;
    }

    private bool IsCalledDuringRegistrationPhase => !this.container.IsLocked();

    [DebuggerStepThrough]
    public ConstructorInfo GetConstructor(Type implementationType) {
        var constructor = this.GetConstructors(implementationType).FirstOrDefault();
        if (constructor != null) return constructor;
        throw new ActivationException(BuildExceptionMessage(implementationType));
    }

    private IEnumerable<ConstructorInfo> GetConstructors(Type implementation) =>
        from ctor in implementation.GetConstructors()
        let parameters = ctor.GetParameters()
        where this.IsCalledDuringRegistrationPhase
            || implementation.GetConstructors().Length == 1
            || ctor.GetParameters().All(this.CanBeResolved)
        orderby parameters.Length descending
        select ctor;

    private bool CanBeResolved(ParameterInfo parameter) =>
        this.GetInstanceProducerFor(new InjectionConsumerInfo(parameter)) != null;

    private InstanceProducer GetInstanceProducerFor(InjectionConsumerInfo i) =>
```

```
        this.container.Options.DependencyInjectionBehavior.GetInstanceProducer(i,␣
↪false);

    private static string BuildExceptionMessage(Type type) =>
        !type.GetConstructors().Any()
            ? TypeShouldHaveAtLeastOnePublicConstructor(type)
            : TypeShouldHaveConstructorWithResolvableTypes(type);

    private static string TypeShouldHaveAtLeastOnePublicConstructor(Type type) =>
        string.Format(CultureInfo.InvariantCulture,
            "For the container to be able to create {0}, it should contain at least "␣
↪+
            "one public constructor.", type.ToFriendlyName());

    private static string TypeShouldHaveConstructorWithResolvableTypes(Type type) =>
        string.Format(CultureInfo.InvariantCulture,
            "For the container to be able to create {0}, it should contain a public "␣
↪+
            "constructor that only contains parameters that can be resolved.",
            type.ToFriendlyName());
}

// Usage
var container = new Container();
container.Options.ConstructorResolutionBehavior =
    new MostResolvableConstructorBehavior(container);
```

The previous examples changed the constructor overload resolution for all registered types. This is usually not the best approach, since this promotes ambiguity in design of our classes. Since ambiguity is usually only a problem in code generation scenarios, it's best to only override the behavior for types that are affected by the code generator.

Overriding Property Injection Behavior

Attribute based property injection and implicit property injection are not supported by Simple Injector out of the box. With attribute based property injection the container injects properties that are decorated with an attribute. With implicit property injection the container automatically injects all properties that can be mapped to a registration, but silently skips other properties. An extension point is provided to change the library's default behavior, which is to **not** inject any property at all.

Out of the box, Simple Injector does allow explicit property injection based on registration of delegates using the **RegisterInitializer** method:

```
container.Register<ILogger, FileLogger>();
container.RegisterInitializer<FileLogger>(instance => {
    instance.Path = "c:\logs\log.txt";
});
```

This enables property injection on a per-type basis and it allows configuration errors to be spot by the C# compiler and is especially suited for injection of configuration values. Downside of this approach is that the *Diagnostic Services* will not be able to analyze properties injected this way and although the **RegisterInitializer** can be called on base types and interfaces, it is cumbersome when applying property injection on a larger scale.

The Simple Injector API exposes the **IPropertySelectionBehavior** interface to change the way the library does property injection. The example below shows a custom **IPropertySelectionBehavior** implementation that enables attribute based property injection using any custom attribute:

```
using System;
using System.Linq;
using System.Reflection;
using SimpleInjector.Advanced;

class PropertySelectionBehavior<T> : IPropertySelectionBehavior where T : Attribute {
    public bool SelectProperty(PropertyInfo prop) =>
        prop.GetCustomAttributes(typeof(T)).Any();
}

// Usage:
var container = new Container();
container.Options.PropertySelectionBehavior =
    new PropertySelectionBehavior<MyInjectAttribute>();
```

This enables explicit property injection on all properties that are marked with the supplied attribute (in this case **MyInjectAttribute**). In case a property is decorated that can't be injected, the container will throw an exception.

Tip: Dependencies injected by the container through the **IPropertySelectionBehavior** will be analyzed by the *Diagnostic*, just like any constructor dependency is analyzed.

Implicit property injection can be enabled by creating an **IPropertySelectionBehavior** implementation that queries the container to check whether the property's type to be registered in the container:

```
public class ImplicitPropertyInjectionBehavior : IPropertySelectionBehavior {
    private readonly IPropertySelectionBehavior original;
    private readonly ContainerOptions options;

    internal ImplicitPropertyInjectionBehavior(Container container) {
        this.options = container.Options;
        this.original = container.Options.PropertySelectionBehavior;
    }

    public bool SelectProperty(Type t, PropertyInfo p) =>
        this.IsImplicitInjectable(t, p) || this.original.SelectProperty(t, p);

    private bool IsImplicitInjectable(Type t, PropertyInfo p) =>
        IsInjectableProperty(p) && this.CanBeResolved(t, p);

    private static bool IsInjectableProperty(PropertyInfo property) =>
        property.CanWrite && property.GetSetMethod(nonPublic: false)?.IsStatic ==␣
↪false;

    private bool CanBeResolved(Type t, PropertyInfo property) =>
        this.GetProducer(new InjectionConsumerInfo(t, property)) != null;

    private InstanceProducer GetProducer(InjectionConsumerInfo info) =>
        this.options.DependencyInjectionBehavior.GetInstanceProducer(info, false);
}

// Usage:
var container = new Container();
container.Options.PropertySelectionBehavior =
    new ImplicitPropertyInjectionBehavior(container);
```

Warning: Silently skipping properties that can't be mapped can lead to a DI configuration that can't be easily verified and can therefore result in an application that fails at runtime instead of failing when the container is verified. Prefer explicit property injection -or better- constructor injection whenever possible.

Overriding Parameter Injection Behavior

Simple Injector does not allow injecting primitive types (such as integers and string) into constructors. The **IDependencyInjectionBehavior** interface is defined by the library to change this default behavior.

The following article contains more information about changing the library's default behavior: Primitive Dependencies with Simple Injector[105].

Resolving Unregistered Types

Unregistered type resolution is the ability to get notified by the container when a type is requested that is currently unregistered in the container. This gives you the change of registering that type. Simple Injector supports this scenario with the ResolveUnregisteredType[106] event. Unregistered type resolution enables many advanced scenarios. The library itself uses this event for implementing enabling support for *decorators*.

For more information about how to use this event, please look at the ResolveUnregisteredType event documentation[107] in the reference library[108].

Overriding Lifestyle Selection Behavior

By default, when registering a type without explicitly specifying a lifestyle, that type is registered using the **Transient** lifestyle. This behavior can be overridden and this is especially useful in batch-registration scenarios.

Here are some examples of registration calls that all register types as *Transient*:

```
container.Register<IUserContext, AspNetUserContext>();
container.Register<ITimeProvider>(() => new RealTimeProvider());
container.RegisterCollection<ILogger>(new[] { typeof(SqlLogger), typeof(FileLogger) }
↪);
container.Register(typeof(IHandler<>), new[] { typeof(IHandler<>).Assembly });
container.RegisterDecorator(typeof(IHandler<>), typeof(LoggingHandlerDecorator<>));
container.RegisterConditional(typeof(IValidator<>), typeof(NullVal<>), c => !c.
↪Handled);
container.RegisterMvcControllers();
container.RegisterWcfServices();
container.RegisterWebApiControllers(GlobalConfiguration.Configuration);
```

Most of these methods have overloads that allow supplying a different lifestyle. This works great in situations where you register a single type (using one of the **Register** method overloads for instance), and when all registrations need the same lifestyle. This is less suitable for cases where you batch-register a set of types where each type needs a different lifestyle.

In this case we need to override the way Simple Injector does lifestyle selection. There are two ways of overriding the lifestyle selection.

Overriding the lifestyle selection can done globally by changing the **Container.Options.DefaultLifestyle** property, as shown in the following example:

```
container.Options.DefaultLifestyle = Lifestyle.Singleton;
```

[105] https://cuttingedge.it/blogs/steven/pivot/entry.php?id=94
[106] https://simpleinjector.org/ReferenceLibrary/?topic=html/E_SimpleInjector_Container_ResolveUnregisteredType.htm
[107] https://simpleinjector.org/ReferenceLibrary/?topic=html/E_SimpleInjector_Container_ResolveUnregisteredType.htm
[108] https://simpleinjector.org/ReferenceLibrary/

Any registration that's not explicitly supplied with a lifestyle, will get this lifestyle. In this case all registrations will be made as **Singleton**.

A more common need is to select the lifestyle based on some context. This can be done by creating custom implementation of **ILifestyleSelectionBehavior**.

```
public interface ILifestyleSelectionBehavior {
    Lifestyle SelectLifestyle(Type implementationType);
}
```

When no lifestyle is explicitly supplied by the user, Simple Injector will call into the registered **ILifestyleSelectionBehavior** when the type is registered to allow the **ILifestyleSelectionBehavior** implementation to select the proper lifestyle. The default behavior can be replaced by setting the **Container.Options.LifestyleSelectionBehavior** property.

Simple Injector's default **ILifestyleSelectionBehavior** implementation simply forwards the call to **Container.Options.DefaultLifestyle**.

The following example changes the lifestyle selection behavior to always register those instances as singleton:

```
using System;
using SimpleInjector;
using SimpleInjector.Advanced;

// Custom lifestyle selection behavior
public class SingletonLifestyleSelectionBehavior : ILifestyleSelectionBehavior {
    public Lifestyle SelectLifestyle(Type implementationType) => Lifestyle.Singleton;
}

// Usage
var container = new Container();
container.Options.LifestyleSelectionBehavior = new␣
↪SingletonLifestyleSelectionBehavior();
```

In case there is always a single default lifestyle, a much easier to set the **Container.Options.DefaultLifestyle** property:

```
container.Options.DefaultLifestyle = Lifestyle.Singleton;
```

The default **Container.Options.LifestyleSelectionBehavior** implementation simply returns the configured **Container.Options.DefaultLifestyle**.

It gets more interesting when the lifestyle changes on the given type. The following example changes the lifestyle selection behavior to pick the lifestyle based on an attribute:

```
using System;
using System.Reflection;
using SimpleInjector.Advanced;

// Attribute for use by the application
public enum CreationPolicy { Transient, Scoped, Singleton }

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface,
    Inherited = false, AllowMultiple = false)]
public sealed class CreationPolicyAttribute : Attribute {
    public CreationPolicyAttribute(CreationPolicy policy) {
        this.Policy = policy;
    }

    public CreationPolicy Policy { get; }
```

```
}

// Custom lifestyle selection behavior
public class AttributeBasedLifestyleSelectionBehavior : ILifestyleSelectionBehavior {
    private const CreationPolicy DefaultPolicy = CreationPolicy.Transient;

    public Lifestyle SelectLifestyle(Type type) => ToLifestyle(GetPolicy(type));

    private static Lifestyle ToLifestyle(CreationPolicy policy) =>
        policy == CreationPolicy.Singleton ? Lifestyle.Singleton :
        policy == CreationPolicy.Scoped ? Lifestyle.Scoped :
        Lifestyle.Transient;

    private static CreationPolicy GetPolicy(Type type) =>
        type.GetCustomAttribute<CreationPolicyAttribute>()?.Policy ?? DefaultPolicy;
}

// Usage
var container = new Container();
container.Options.DefaultScopedLifestyle = new WebRequestLifestyle();

container.Options.LifestyleSelectionBehavior =
    new AttributeBasedLifestyleSelectionBehavior();

container.Register<IUserContext, AspNetUserContext>();

// Usage in application
[CreationPolicy(CreationPolicy.Scoped)]
public class AspNetUserContext : IUserContext {
    // etc
}
```

Intercepting the Creation of Types

Intercepting the creation of types allows registrations to be modified. This enables all sorts of advanced scenarios where the creation of a single type or whole object graphs gets altered. Simple Injector contains two events that allow altering the type's creation: ExpressionBuilding[109] and ExpressionBuilt[110]. Both events are quite similar but are called in different stages of the *building pipeline*.

The **ExpressionBuilding** event gets called just after the registration's expression has been created that new up a new instance of that type, but before any lifestyle caching has been applied. This event can for instance be used for *Context based injection*.

The **ExpressionBuilt** event gets called after the lifestyle caching has been applied. After lifestyle caching is applied much of the information that was available about the creation of that registration during the time **ExpressionBuilding** was called, is gone. While **ExpressionBuilding** is especially suited for changing the relationship between the resolved type and its dependencies, **ExpressionBuilt** is especially useful for applying decorators or *applying interceptors*.

Note that Simple Injector has built-in support for *applying decorators* using the RegisterDecorator[111] extension methods. These methods internally use the **ExpressionBuilt** event.

[109] https://simpleinjector.org/ReferenceLibrary/?topic=html/E_SimpleInjector_Container_ExpressionBuilding.htm
[110] https://simpleinjector.org/ReferenceLibrary/?topic=html/E_SimpleInjector_Container_ExpressionBuilding.htm
[111] https://simpleinjector.org/ReferenceLibrary/?topic=html/Overload_SimpleInjector_Extensions_DecoratorExtensions_RegisterDecorator.htm

Building up External Instances

Some frameworks insist in creating some of the classes we write and want to manage their lifetime. A notorious example of this is ASP.NET Web Forms. One of the symptoms we often see with those frameworks is that the classes that the framework creates need to have a default constructor.

This disallows Simple Injector to create those instances and inject dependencies into their constructor. But Simple Injector can still be asked to initialize such instance according the container's configuration. This is especially useful when overriding the default *property injection behavior*.

The following code snippet shows how an external instance can be initialized:

```
public static BuildUp(Page page) {
    InstanceProducer producer =
        container.GetRegistration(page.GetType(), throwOnFailure: true);
    Registration registration = producer.Registration;
    registration.InitializeInstance(page);
}
```

This allows any properties and initializers to be applied, but obviously doesn't allow the lifestyle to be changed, or any decorators to be applied.

By calling the **GetRegistration** method, the container will create and cache an *InstanceProducer* instance that is normally used to create the instance. Note however, that the **GetRegistration** method restricts the shape of the type to initialize. Since **GetRegistration** is used in cases where Simple Injector creates types for you, Simple Injector will therefore check whether it can create that type. This means that if this type has a constructor with arguments that Simple Injector can't inject (for instance because there are primitive type arguments in there), an exception will be thrown.

In that particular case, instead of requesting an *InstanceProducer* from the container, you need to create a *Registration* class using the *Lifestyle* class:

```
Registration registration =
    Lifestyle.Transient.CreateRegistration(page.GetType(), container);
registration.InitializeInstance(page);
```

Do note however that if you create *Registration* instances manually, make sure you cache them. *Registration* instances generate expression trees and compile them down to a delegate. This is a time -and memory- consuming operation. But every second time you call **InitializeInstance** on the same *Registration* instance, it will be fast as hell.

Interception of Resolved Object Graphs

Simple Injector allows registering a delegate that will be called every time an instance is resolved directly from the container. This allows executing code just before and after an object graph gets resolved. This allows plugging in monitoring or diagnosing the container.

The Glimpse plugin for Simple Injector[112] for instance, makes use of this hook to allow displaying information about which objects where resolved during a web request.

The following example shows the **Options.RegisterResolveInterceptor** method in action:

```
container.Options.RegisterResolveInterceptor(CollectResolvedInstance, c => true);

private static object CollectResolvedInstance(InitializationContext context,
    Func<object> instanceProducer)
```

[112] https://www.nuget.org/packages/Glimpse.SimpleInjector/

```
{
    // Invoke the delegate that calls into Simple Injector to get the requested␣
↪service.
    object instance = instanceProducer();

    // Collect request specific data for display to the user.
    List<InstanceInitializationData> list =␣
↪GetListForCurrentRequest(ResolvedInstances);
    list.Add(new InstanceInitializationData(context, instance));

    // Return the resolve instance.
    return instance;
}
```

The example above shows the registration code from the Glimpse plugin component. It registers an interception delegate to the *CollectResolvedInstance* method by calling *container.Options.RegisterResolveInterceptor*. The *c => true* lambda informs Simple Injector that the *CollectResolvedInstance* method should always be applied for every service that is being resolved. This makes sense for the Glimpse plugin, because the user would want to get a complete view of what is being resolved during that request.

When a user calls **Container.GetInstance** or **InstanceProducer.GetInstance**, instead of creating the requested instance, Simple Injector will call the *CollectResolvedInstance* method and supplies to that method:

1. An **InitializationContext** that contains information about the service that is requested.
2. An *Func<object>* delegate that allows the requested instance to be created.

The **InitializationContext** allows access to the **InstanceProducer** and **Registration** instances that describe the service's registration. These two types enable detailed analysis of the resolved service, if required.

An **InstanceProducer** instance is responsible of caching the compiled factory delegate that allows the creation of new instances (according to their lifestyle) that is created. This factory delegate is a *Func<object>*. In case a *resolve interceptor* gets applied to an **InstanceProducer**, instead of calling that *Func<object>*, the **InstanceProducer** will call the resolve interceptor, while supplying that original *Func<object>* to the interceptor.

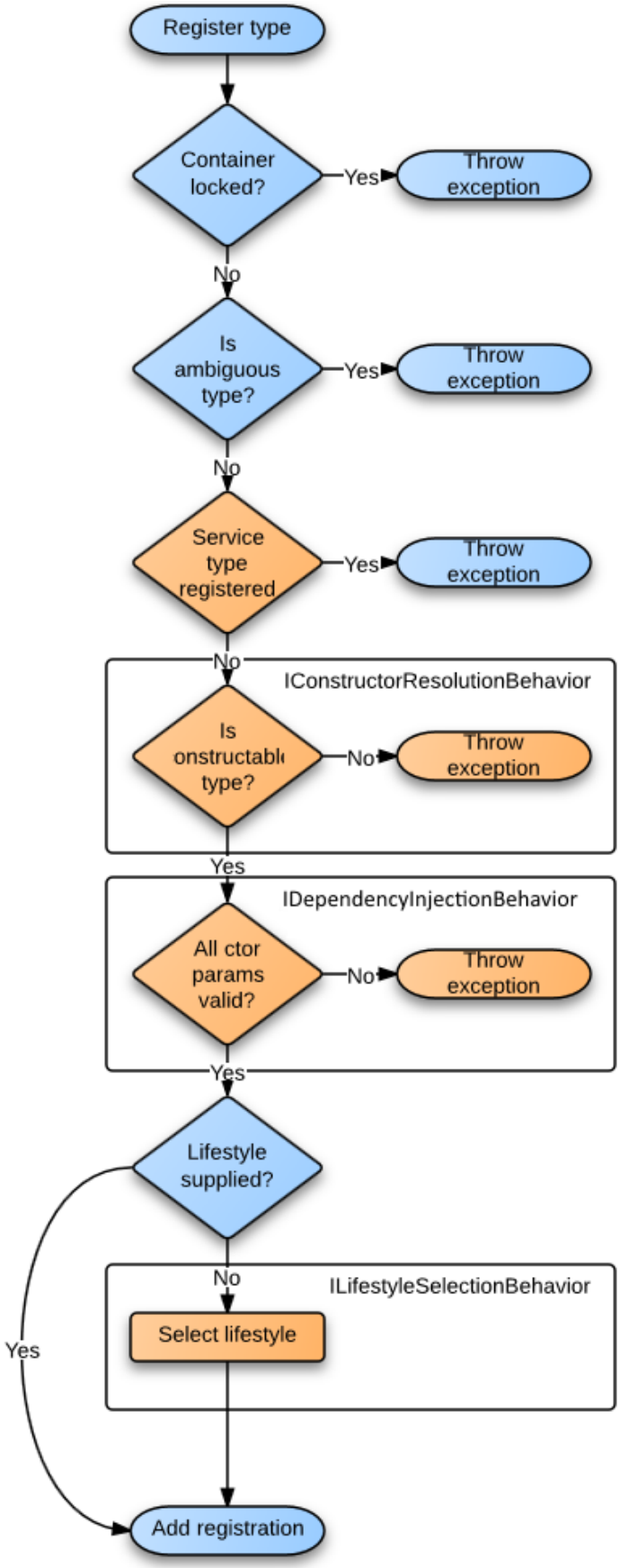
CHAPTER 10

Simple Injector Pipeline

The pipeline is a series of steps that the Simple Injector container follows when registering and resolving each type. Simple Injector supports customizing certain steps in the pipeline to affect the default behavior. This chapter describes these steps and how the pipeline can be customized.

Registration Pipeline

The registration pipeline is the set of steps Simple Injector takes when making a registration in the container. This pipeline mainly consists of a set of validations that is performed. The process is rather straightforward and looks like this:


Register type
Container locked?
Yes
Throw exception
No
Is ambiguous type?
Yes
Throw exception
No
Service type registered
Yes
Throw exception
No
IConstructorResolutionBehavior
Is onstructabl type?
No
Throw exception
Yes
IDependencyInjectionBehavior
All ctor params valid?
No
Throw exception
Yes
Lifestyle supplied?
No
ILifestyleSelectionBehavior
Select lifestyle
Yes
Add registration

Warning: The order in which those validations are performed is undocumented and might change from version to version.

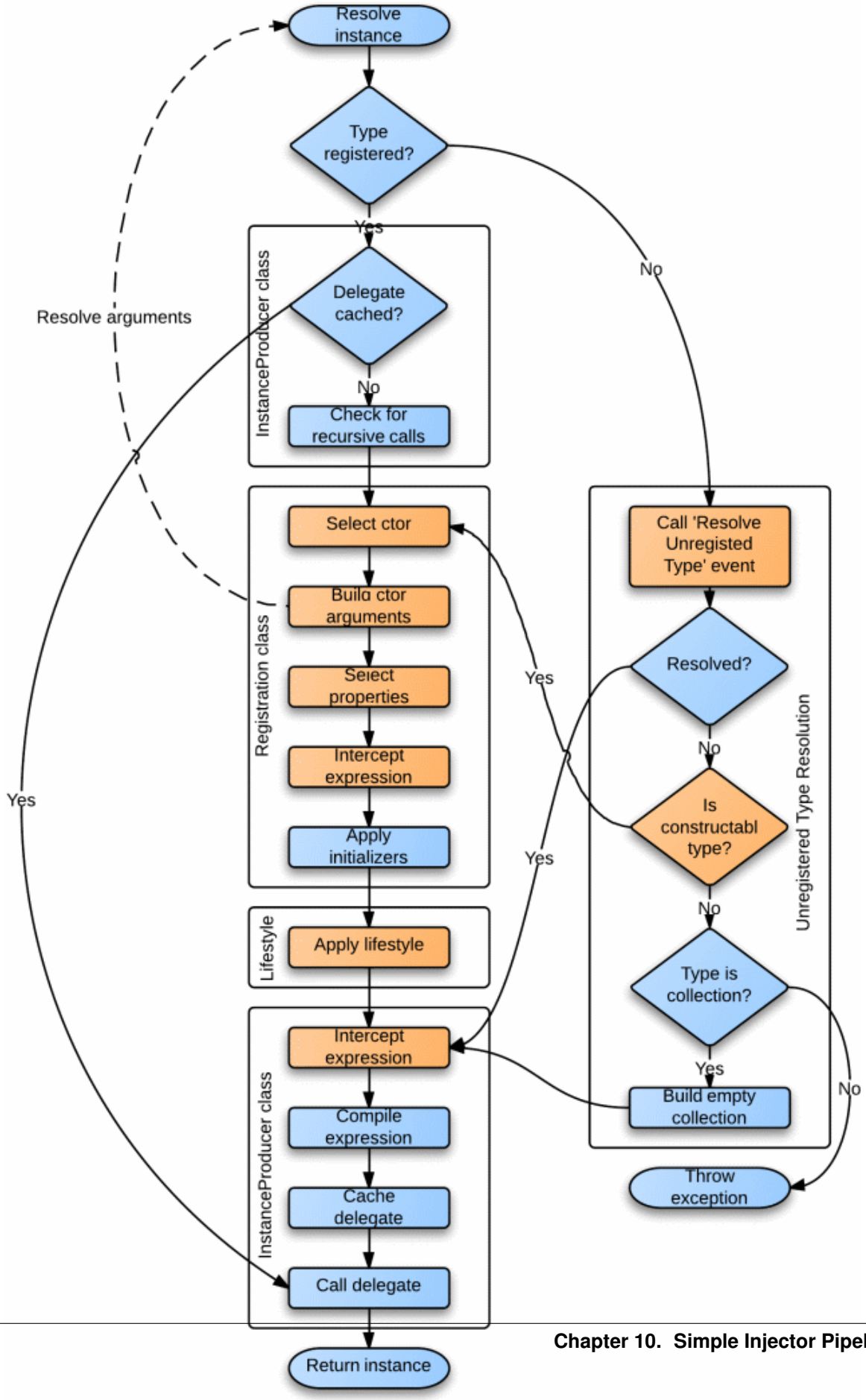
Steps:

- **Container locked?**: When the first type is resolved from the container, the container is locked for further changes. When a call to one of the registration methods is made after that, the container will throw an exception. The container can't be unlocked. This behavior is fixed and can't be changed. For a thorough explanation about this design, please read the *design principles documentation*. That documentation section also explains how to add new registrations after this point.
- **Is ambiguous type?**: When a type is registered that is considered to be ambiguous, the container will throw an exception. Types such as *System.Type* and *System.String* are considered ambiguous, since they have value type semantics and its unlikely that the configuration only contains a single definition of such type. For instance, when components take a string parameter in their constructor, it's very unlikely that all components need that same value. Instead some components need a connection string, others a path to the file system, etc. To prevent this ambiguity in the configuration, Simple Injector blocks these registrations. This behavior is fixed and can't be changed.
- **Service type registered?**: When a service type has already been registered, the container throws an exception. This prevents any accidental misconfigurations. This behavior can be overridden. See the *How to override existing registrations* for more information.
- **Is constructable type?**: When the registration is supplied with an implementation type that the container must create and auto-wire (using **Register<TService, TImplementation>()** for instance), the container checks if the implementation type is constructable. A type is considered to be constructable when it is a concrete type and has a single public constructor. An exception is thrown when these conditions are not met. This behavior can be overridden by implementing a custom **IConstructorResolutionBehavior**.
- **All ctor params valid?**: The constructor that has been selected in the previous step will be analyzed for invalid constructor parameters. Ambiguous types such as *System.String* and value types such as *Int32* and *Guid* are not allowed. This behavior can be overridden by implementing a custom **IDependencyInjectionBehavior**. Take a look at this blog post[113] for an elaborate example.
- **Lifestyle supplied?**: If the user explicitly supplied a **Lifestyle** to its registration, the registration is done using that lifestyle, i.e. the registration is made using one of the **Register** method overload that accepts a **Lifestyle** instance. Otherwise, the configured **ILifestyleSelectionBehavior** is queried to get the proper lifestyle for the given implementation type. The default **ILifestyleSelectionBehavior** simply returns the value of **Container.Options.DefaultLifestyle**. By default this means that the registration is made using the **Transient** lifestyle, but this behavior can be overridden by either changing the **Container.Options.DefaultLifestyle** or by replacing the default **ILifestyleSelectionBehavior**.
- **Add registration**: When all previous validations succeeded, the registration is added to the container. Although the type may be registered successfully, this still doesn't mean it can always be resolved. This depends on several other factors such as whether all dependencies can be resolved correctly. These checks cannot be performed during registration, and they are performed during the *Resolve Pipeline*.

Resolve Pipeline

The resolve pipeline is the set of steps Simple Injector takes when resolving an instance. Many steps in the pipeline can be replaced (orange in the diagram below) to change the default behavior of the container. The following diagram visualizes this pipeline:

[113] http://www.cuttingedge.it/blogs/steven/pivot/entry.php?id=94


Resolve instance
Type registered?
Yes
No
Delegate cached?
No
Check for recursive calls
InstanceProducer class
Resolve arguments
Select ctor
Build ctor arguments
Select properties
Intercept expression
Apply initializers
Registration class
Yes
Yes
Yes
Lifestyle
Apply lifestyle
Intercept expression
Compile expression
Cache delegate
Call delegate
InstanceProducer class
Return instance
Call 'Resolve Unregisted Type' event
Resolved?
No
Is constructabl type?
No
Type is collection?
Yes
No
Build empty collection
Throw exception
Unregistered Type Resolution

Note: The order in which those steps are performed is *deterministic* and can be safely depended upon.

Steps:

- **Type registered?**: If a type is requested that is not yet registered, the container falls back to unregistered type resolution.
- **Delegate cached?**: At the end of the pipeline, the compiled delegate is cached during the lifetime of the *Container* instance. Executing the pipeline steps is expensive and caching the delegate is crucial for performance. This does mean though, that once compiled, the way a type is created, cannot be changed.
- **Check for recursive calls**: The container checks if a type indirectly depends on itself and throw a descriptive exception in this case. This prevents any hard to debug StackOverflowException that might otherwise occur.
- **Select ctor**: The container selects the single public constructor of the concrete type. This behavior can be overridden by implementing a custom **IConstructorResolutionBehavior**. When the registration is made using a *Func<T>* delegate, this and the following steps are skipped.
- **Build ctor arguments**: The container will call back into itself to resolve all constructor arguments. This results in a recursive call into the container which will trigger building a complete object graph. The container will throw an exception when one of the parameters cannot be resolved. This behavior can be overridden by implementing a custom **IDependencyInjectionBehavior**. Take a look at this blog post[114] for an elaborate example. The result of this step is an *Expression* that describes the invocation of the constructor with its arguments, e.g. "*new MyService(new DbLogger(), new MyRepository(new DbFactory()))*".
- **Select properties**: The default behavior of the container is to not inject any properties and without any customization this step will be skipped. This behavior can be changed by implementing a custom **IPropertySelectionBehavior**. This custom behavior can decide how to handle each property (both public and non-public) of the given implementation. **Note that read-only properties (without a setter) and static properties will be queried as well, although they can never be injected**. It is the responsibility of the implementation to decide what to do with those properties. Note that the container will **not** silently skip any properties. If the custom property selection behavior returns true for a given property, the container throws an exception when the property cannot be injected. For instance, because the dependency can't be resolved or when the application's sandbox does not permit accessing internal types. When this step resulted in any properties being injected, it results in an *Expression* that describes the invocation of a delegate that injects the properties into the type that was created in the previous step, e.g. *"injectProperties(new PropertyDependency1(), new PropertyDependency2(), new ServiceToInjectInto(new DbLogger()))"*. The 'injectProperties' in this case is a compiled delegate that takes in the created instance as last element and returns that same instance. The other arguments passed into this delegate are the properties that must be injected. Note that although this *Expression* calls a delegate, the delegate only sets the type's properties based on method arguments. The *Expression* still contains all dependencies of the type (both constructor and property). It is important to note that the structure of this expression might change from version to version, but the fact that the expression holds all dependency information will not (and the service to inject the properties into will always be the last argument, since the framework has to ensure that the type's dependencies are created first). By building this structure with all information available, we allow the following step to have complete control over the expression. Note that in case the registration is made using a *Func<T>* delegate, only the properties of the supplied *TService* will be queried and not the properties of the actually returned type (which might be a sub type of *TService*). For more information about changing the default behavior, see the *Property Injection* section on the *Advanced Scenarios* page.
- **Intercept expression** (1): By default the container skips this step. Users can hook a delegate onto the **ExpressionBuilding** event. This event allows molding and changing the expression built in the previous step. Please take a look at the *Context Based Injection* section in the *Advanced scenarios* wiki page for an example of what you can achieve by hooking onto this event. Note that there is a restriction to the changes you can make to the expression. Although the Expression can be changed completely, you have to make sure that any replaced expression returns the same implementation type (or a subtype).

[114] http://www.cuttingedge.it/blogs/steven/pivot/entry.php?id=94

- **Apply initializers**: Any applicable *Action* delegates that are registered using **RegisterInitializer<T>(Action<T>)**, will be applied to the expression at this point. When one or more initializers are applied, it results in the creation of an *Expression* that wraps the original expression and invokes a delegate that calls the *Action* delegates, i.e. *"applyInitializers(new MyService())"*.
- **Apply lifestyle**: Until this point in the pipeline, the expression that has been built describes the creation of a new instance (transient). This step applies caching to this instance. Lifestyles are applied by **Lifestyle** implementations. They use the expression that was built up using the previous steps and they are allowed to compile this expression to a delegate, before applying the caching. This means that the expressiveness about all the type's dependencies can be embedded in the compiled delegate and is unavailable for analysis and interception when the next step is applied.
- **Intercept expression** (2): The container's **ExpressionBuilt** event gets triggered after the lifestyle has been applied to an expression. The container's *RegisterDecorator* extension methods internally make use of this event to decorate any type while preserving the lifestyle of that type. Multiple **ExpressionBuilt** events could handle the same type and they are all applied in the order in which they are registered.
- **Compile expression**: In this step, the expression that is the result of the previous step is compiled to a *Func<object>* delegate. Several optimizations are applied. This step cannot be customized.
- **Cache delegate**: The compiled delegate is stored for reuse. This step cannot be customized.
- **Call delegate**: The cached delegate is called to resolve an instance of the registered type. This step cannot be customized.
- **Call 'Resolve Unregistered Type' event**: When a type is requested that is not registered, the container will call the *ResolveUnregisteredType* event. Users can hook onto this event to make a last-minute registration in the container, even after the container has been locked down.
- **Resolved?**: When there was a registered **ResolveUnregisteredType** event that responded to the unregistered type, it is assumed that it has a lifestyle applied. It therefore makes a jump through the pipeline and continues right after the *Apply lifestyle* step. This allows any post lifestyle interception (such as decorators) to still be applied to types that are resolved using unregistered type resolution.
- **Is constructable type?**: When no **ResolveUnregisteredType** handled the registration of the given type, the container will check if the type is constructable. This is done by querying the *IConstructorResolutionBehavior* and **IDependencyInjectionBehavior** implementations. By default, this means that the type should have a single public constructor, that the constructor arguments should not be ambiguous types (such as String or a value type) and that it can be resolved by the container. This behavior can be customized. If a type is constructable according to these rules, the type is created by running it through the pipeline starting at *Select ctor* step with the transient lifetime. In other words, concrete types that are not registered explicitly, will by default get resolved with the transient lifestyle.
- **Type is collection?**: When the requested type is an *IEnumerable<T>*, *ICollection<T>*, *IList<T>*, *IReadOnlyCollection<T>* or *IReadOnlyList<T>*, and **Container.Options.ResolveUnregisteredCollections** is set, the container will build an empty list that will be used as singleton. This collection will be passed on to the *Intercept expression* step after *Apply lifestyle* to allow this empty list to still be intercepted and decorated. If the type is not an *IEnumerable<T>*, the type can't be created by the container and an exception is thrown. Note that the default value for **Container.Options.ResolveUnregisteredCollections** is **false**, which means that the container will throw an exception rather than building an empty list.

CHAPTER 11

Design Principles

While designing Simple Injector we defined a set of rules that formed the foundation for development. These rules still keep us focused today and continue to help us decide how to implement a feature and which features **not** to implement. In the section below you'll find details of the design principles of Simple Injector.

The design principles:

- *Make simple use cases easy, make complex use cases possible*
- *Push developers into best practices*
- *Fast by default*
- *Don't force vendor lock-in*
- *Never fail silently*
- *Features should be intuitive*
- *Communicate errors clearly and describe how to solve them*

Make simple use cases easy, make complex use cases possible

This guideline comes directly from the Framework Design Guidelines[115] and is an important guidance for us. Commonly used features should be easy to implement, even for a new user, but the library must be flexible and extensible enough to support complex scenarios.

Push developers into best practices

We believe in good design and best practices. When it comes to Dependency Injection, we believe that we know quite a bit about applying design patterns correctly and also how to prevent applying patterns incorrectly. We have designed Simple Injector in a way that promotes these best practices. Occasional we may explicitly choose to **not** implement

[115] http://www.amazon.com/Framework-Design-Guidelines-Conventions-Libraries/dp/0321545613

certain features because they don't steer the developer in the right direction. Our intention has always been to build a library that makes it difficult to shoot yourself in the foot!

Fast by default

For most applications the performance of the DI library is not an issue; I/O is usually the bottleneck. You will find, however, that certain DI libraries are very sensitive to different configurations and you will need to monitor the container for potential performance problems. Most performance problems can be fixed by changing the configuration (changing registrations to singleton, adding caching, etc), no matter which library you use. At that point however it can get really complicated to configure certain libraries.

Making Simple Injector fast by default removes any concerns regarding the performance of the construction of object graphs. Instead of having to monitor Simple Injector's performance and make ugly tweaks to the configuration when object construction is too slow, the developer is free to worry about more important things.

Fast by default means that the performance of object instantiation from any of the registration features that the library supplies out-of-the-box will be comparable to the performance of hard-wired object instantiation.

Don't force vendor lock-in

The Dependency injection pattern promotes the use of loosely coupled components. -When done right- loosely coupled code can be much more maintainable. When building a library that promotes this pattern it would be ironic if that same library was to ask you to take a dependency on the library. The truth is many 3rd party library providers do want you to use certain abstractions and attributes from their offering and thereby force you to create a hard dependency to their code.

When we build applications ourselves, we try to prevent any vendor lock-in (even to Simple Injector), so why should we force you to get locked into Simple Injector? We don't want to do this. We want you to get hooked to Simple Injector, but we want this to be through the compelling vision and competing features; not by vendor lock-in. If Simple Injector doesn't suit your needs you should be able to easily swap it for another competing product, just as you would want to replace your logging library without it affecting your entire code base.

Never fail silently

We all hate the hunt for bugs in our code. It can be made even worse when we discover a library or framework we have chosen to use is hiding these bugs by ignoring them and failing to report them to us. A good example is logging libraries - many of us have been frustrated when we discover our logging libraries continue to run without logging, because we misconfigured it, but didn't bother to inform us. This frustration can lead to real world costs and a lack of trust in the tools we use.

We decided that Simple Injector should by default never fail silently. If you make a configuration error then Simple Injector should tell you as soon as reasonably possible. We want Simple Injector to fail fast!

Features should be intuitive

This means that features should be easy to use and do the right thing by default.

Communicate errors clearly and describe how to solve them

In our day jobs we regularly encounter exception messages that aren't helpful or, even worse, are misleading (we have all seen the *NullReferenceException*). It frustrates us, takes time to track down and therefore costs money. We don't want to put any developer in that position and therefore defined an explicit design rule stating that Simple Injector should always communicate errors as clearly as possible. And, not only should it describe the problem, it should offer details on the options for solving the problem.

If you encounter a scenario where we fail to do this, please let us know. We are serious about this and we will fix it!

CHAPTER 12

Design Decisions

Our *design principles* have influenced the direction of the development of features in Simple Injector. In this section we would like to explain some of the design decisions.

- *The container is locked after the first call to resolve*
- *The API clearly separates registration of collections from other registrations*
- *No support for XML based configuration*
- *Never force users to release what they resolve*
- *Don't allow resolving scoped instances outside an active scope*
- *No out-of-the-box support for property injection*
- *No out-of-the-box support for interception*
- *Limited batch-registration API*
- *No per-thread lifestyle*
- *Allow only a single constructor*

The container is locked after the first call to resolve

When an application makes its first call to **GetInstance**, **GetAllIntances** or **Verify**, the container locks itself to prevent any future changes being made by explicit registrations. This strictly separates the configuration of the container from its use and forces the user to configure the container in one single place. This design decision is inspired by the following design principle:

- *Push developers into best practices*

In most situations it makes little sense to change the configuration once the application is running. This will make the application much more complex, whereas dependency injection as a pattern is meant to lower the total complexity of a system. By strictly separating the registration/startup phase from the phase where the application is in a running state, it is much easier to determine how the container will behave and it is much easier to verify the container's configuration.

The locking behavior of Simple Injector exists to protect the user from defining invalid and/or confusing combinations of registrations.

Allowing the ability to alter the DI configuration while the application is running could easily cause strange, hard to debug, and hard to verify behavior. This may also mean the application has numerous hard references to the container and this is something we work hard to prevent. Attempting to alter the configuration when running a multi-threaded application would lead to very un-deterministic behavior, even if the container itself is thread-safe.

Imagine the scenario where you want to replace some *FileLogger* component for a different implementation with the same *ILogger* interface. If there's a component that directly or indirectly depends on *ILogger*, replacing the *ILogger* implementation might not work as you would expect. If the consuming component is registered as singleton, for example, the container should guarantee that only one instance of this component will be created. When you are allowed to change the implementation of *ILogger* after a singleton instance already holds a reference to the "old" registered implementation the container has 2 choices - neither of which are correct:

1. Return the cached instance of the consuming component that has a reference to the "wrong" *ILogger* implementation.
2. Create and cache a new instance of that component and, in doing so, break the promise of the type being registered as a singleton and the guarantee that the container will always return the same instance.

The description above is a simple to grasp example of dealing with the runtime replacement of services. But adding new registrations can also cause things to go wrong in unexpected ways. A simple example would be where the container has previously supplied the object graph with a default implementation resolved through unregistered type resolution.

Problems with thread-safety can easily emerge when the user changes a registration during a web request. If the container allowed such registration changes during a request, other requests could directly be impacted by those changes (since in general there should only be one *Container* instance per AppDomain). Depending on things such as the lifestyle of the registration; the use of factories and how the object graph is structured, it could be a real possibility that another request gets both the old and the new registration. Take for instance a transient registration that is replaced with a different one. If this is done while an object graph for a different thread is being resolved while the service is injected into multiple points within the graph - the graph would contain different instance of that abstraction with different lifetimes at the same time in the same request - and this is bad.

Since we consider it good practice to lock the container, we were able to greatly optimize performance of the container and adhere to the *Fast by default* principle.

Do note that container lock-down still allows runtime registrations. A few common ways to add registrations to the container are:

1. Resolving an unregistered concrete type from the container. The container will auto-register that type for you as transient registration.
2. Using *unregistered type resolution* the container will be able to at a later time resolve new types.
3. The Lifestyle.CreateProducer[116] overloads can be called at any point in time to create new **InstanceProducer** instances that allow building new registrations.

All these options provide users with a safe way to add registrations at a later point in time, without the risks described above.

The API clearly differentiates the registration of collections

When designing Simple Injector, we made a very explicit design decision to define a separate **RegisterCollection** method for registering a collection of services for an abstraction. This design adheres to the following principles:

[116] https://simpleinjector.org/ReferenceLibrary/?topic=html/Overload_SimpleInjector_Lifestyle_CreateProducer.htm

- *Never fail silently*
- *Features should be intuitive*

This design differentiates vastly from how other DI libraries work. Most libraries provide the same API for single registrations and collections. Registering a collection of some abstraction in that case means that you call the **Register** method multiple times with the same abstraction but with different implementations. There are some clear downsides to such an approach.

- There's a big difference between having a collection of services and a single service. For many of the services you register, you will have one implementation and it doesn't make sense for there to be multiple implementations. For other services you will always expect a collection of them (even if you have one or no implementations). In the majority -if not all- of cases you wouldn't expect to switch dynamically between one and multiple implementations.
- An API that mixes these concepts will be unable to warn you if you accidentally add a second registration for the same service. Those APIs will 'fail silently' and simply return one of the items you registered. Simple Injector will throw an exception when you call **Register<T>** for the same T and will describe that collections should be registered using **RegisterCollection**.
- None of the APIs that mix these concepts make it clear which of the registered services is returned if you resolve one of them. Some libraries will return the first registered element, while others return the last. Although all of them describe this behavior in their documentation it's not clear from the API itself i.e. it is not discoverable. An API design like this is unintuitive. A design that separates **Register** from **RegisterCollection** on the other hand, makes the intention of the code very clear to anyone who reads it.

In general, your services should not depend on an *IEnumerable<ISomeService>*, especially when your application has multiple services that need to work with *ISomeService*. The problem with injecting *IEnumerable<T>* into multiple consumers is that you will have to iterate that collection in multiple places. This forces the consumers to know about having multiple implementations and how to iterate and process that collection. As far as the consumer is concerned this should be an implementation detail. If you ever need to change the way a collection is processed you will have to go through the application, since this logic will have be duplicated throughout the system.

Instead of injecting an *IEnumerable<T>*, a consumer should instead depend on a single abstraction and you can achieve this using a Composite[117] Implementation that wraps the actual collection and contains the logic of processing the collection. Registering composite implementation is so much easier with Simple Injector because of the clear separation between a single implementation and a collection of implementations. Take the following configuration for example, where we register a collection of *ILogger* implementations and a single composite implementation for use in the rest of our code:

```
container.RegisterCollection<ILogger>(new[] {
    typeof(FileLogger),
    typeof(SqlLogger),
    typeof(EventLogLogger)
});

container.Register<ILogger, CompositeLogger>(Lifestyle.Singleton);
```

In case the unusual scenario that you need both a default registration and list of registrations, this is still easy to configure in Simple Injector. Take a look at the following example:

```
container.Register<ILogger, FileLogger>();

container.RegisterCollection<ILogger>(new[] {
    typeof(ILogger),
    typeof(SqlLogger),
```

[117] https://en.wikipedia.org/wiki/Composite_pattern

```
    typeof(EventLogLogger)
});
```

The previous example registers both a *FileLogger* as one-to-one registration for *ILogger* and a collection of *ILogger* instances. The first registration in the collection itself is *ILogger* which means that it points back to the one-to-one mapping using *FileLogger*.

This way you have full control over which registration is the default one (in this case the first), since ordering of the collection is guaranteed to be the order of registration.

No support for XML based configuration

While designing Simple Injector, we decided to *not* provide an XML based configuration API, since we want to:

- *Push developers into best practices*

Having a XML centered configuration however is *not* best practice.

XML based configuration is brittle, error prone and always provides a subset of what you can achieve with code based configuration. General consensus is to use code based configuration as much as possible and only fall back to file based configuration for the parts of the configuration that really need to be customizable after deployment. These are normally just a few registrations since the majority of changes would still require developer interaction (write unit tests or recompile for instance). Even for those few lines that do need to be configurable, it's a bad idea to require the fully qualified type name in a configuration file. A configuration switch (true/false or simple enum) is in most cases a better option. You can read the configured value in your code based configuration, this allows you to keep the type names in your code. This allows you to refactor easily, gives you compile-time support and is much more friendly to the person having to change this configuration file.

Putting fully qualified type names in your configuration files is only encouraged when a plugin architecture is required that allows special plugin assemblies to be dropped in a special folder and to be picked up by the application, without the need of a recompile. But even in that case the number of type names in the configuration should be reduced to the bare minimum, where most types are registered using batch-registration in code.

Never force users to release what they resolve

The Register Resolve Release[118] (RRR) pattern is a common pattern that DI containers implement. In general terms the pattern describes that you should tell the container how to build each object graph (Register) during application start-up, ask the container for an object graph (Resolve) at the beginning of a request, and tell the container when you're done with that object graph (Release) after the request.

Although this pattern applies to Simple Injector, we never force users to have to explicitly release any service once they have finished with it. With Simple Injector your components are automatically released when the web request finishes, or when you dispose of your *Thread Scope* or *Async Scope*. By not forcing users to release what they resolve, we adhere to the following design principles:

- *Never fail silently*
- *Features should be intuitive*

A container that expects the user to release the instances they resolve will fail silently when a user forgets to release, because forgetting to release is a failure and the container doesn't know when the user is done with the object graph. Forgetting to release can sometimes leads to out of memory exceptions that are often hard to trace back and are

[118] http://blog.ploeh.dk/2010/09/29/TheRegisterResolveReleasepattern/

therefore costly to fix. The need to release explicitly is far from intuitive and is therefore not needed when working with Simple Injector.

Don't allow resolving scoped instances outside an active scope

When you register a component in Simple Injector with a *scoped lifestyle*, you can only resolve an instance when there is an active instance of that specified scope. For instance, when you register your *DbContext* per Web Request Lifestyle, resolving that instance on a background thread will fail in Simple Injector. This design is chosen because we want to:

- *Never fail silently*

The reason is simple - resolving an instance outside of the context of a scope is a bug. The container could decide to return a singleton or transient for you (as other DI libraries do), but neither of these cases is usually what you would expect. Take a look at the *DbContext* example for instance, the class is normally registered as Per Web Request lifestyle for a reason, probably because you want to reuse one instance for the whole request. Not reusing an instance, but instead injecting a new instance (transient) would most likely not give the expected results. Returning a single instance (singleton) when outside of a scope, i.e. reusing a single *DbContext* over multiple requests/threads will sooner or later lead you down the path of failure.

Because there is not a standard logical default for Simple Injector to return when you request an instance outside of the context of an active scope, the right thing to do is throwing an exception. Returning a transient or singleton is a form of failing silently.

That doesn't mean that you're lost when you really need the option of per request and transient or singleton, you are required to configure such a scope explicitly by defining a *Hybrid* lifestyle. We *Make simple use cases easy, and complex use cases possible*.

No out-of-the-box support for property injection

Simple Injector has no out-of-the-box support for property injection, to adhere to the following principles:

- *Don't force vendor lock-in*
- *Never fail silently*

In general there are two ways of implementing property injection: Implicit and Explicit property injection. With implicit property injection, the container injects any public writable property by default for any instance you resolve. This is done by mapping those properties to configured types. When no such registration exists, or when the property doesn't have a public setter, the property will be skipped. Simple Injector does not do implicit property injection, and for good reason. We think that implicit property injection is simply too uuhh... implicit :-). There are many reasons for a container to skip a property, but in none of the cases does the container know if skipping the property is really what the user wants, or whether it was a bug. In other words, the container is forced to fail silently.

With explicit property injection, the container is forced to inject a property and the process will fail immediately when a property can't be mapped or injected. The common way containers allow you to specify whether a property should be injected or not is by the use of library-defined attributes. As previously discussed, this would force the application to take a dependency on the library, which causes a vendor lock-in.

The use of property injection should be non-standard; constructor injection should be used in the majority -if not all- of cases. If a constructor gets too many parameters (the constructor over-injection anti-pattern), it is an indication of a violation of the Single Responsibility Principle[119] (SRP). SRP violations often lead to maintainability issues. Instead

[119] https://en.wikipedia.org/wiki/Single_responsibility_principle

of fixing constructor over-injection with property injection the root cause should be analyzed and the type should be refactored, probably with Facade Services[120].

Another common reason developers start using properties is because they think their dependencies are optional. Instead of using optional property dependencies, best practice is to inject empty implementations (a.k.a. Null Object pattern[121]) into the constructor; Dependencies should rarely be optional.

This doesn't mean that you can't do property injection with Simple Injector, but with Simple Injector this will have to be *explicitly configured*.

No out-of-the-box support for interception

Simple Injector does not support interception out of the box, because we want to:

- *Push developers into best practices*
- *Fast by default*
- *Don't force vendor lock-in*

Simple Injector tries to push developers into good design, and the use of interception is often an indication of a suboptimal design. We prefer to promote the use of decorators. If you can't apply a decorator around a group of related types, you are probably missing a common (generic) abstraction.

Simple Injector is designed to be fast by default. Applying decorators in Simple Injector is just as fast as normal injection, while applying interceptors has a much higher cost, since it involves the use of reflection.

To be able to intercept, you will need to take a dependency on your interception library, since this library defines an *IInterceptor* interface or something similar (such as Castle's *IInterceptor* or Unity's *ICallHandler*). Decorators on the other hand can be created without asking you to take a dependency on an external library. Since vendor lock-in should be avoided, Simple Injector doesn't define any interfaces or attributes to be used at the application level.

Limited batch-registration API

Most DI libraries have a large and advanced batch-registration API that often allow specifying registrations in a fluent way. The downside of these APIs is that developers will struggle to use them correctly; they are often far from intuitive and the library's documentation needs to be repeatedly consulted.

Instead of creating our own API that would fall into the same trap as all the others, we decided not to have such elaborate API, because:

- *Features should be intuitive*

In most cases we found it much easier to write batch registrations using LINQ; a language that many developers are already familiar with. Specifying your registrations in LINQ reduces the need to learn yet another (domain specific) language (with all its quirks).

When it comes to batch-registering generic-types things are different. Batch-registering generic types can be very complex without tool support. We have defined a clear API consisting of a few **Register** and **RegisterCollection** overloads that covers the majority of the cases.

[120] http://blog.ploeh.dk/2010/02/02/RefactoringtoAggregateServices/
[121] https://en.wikipedia.org/wiki/Null_Object_pattern

No per-thread lifestyle

A per-thread lifestyle caches instances for as long as the thread lives and stores that instance in thread-static storage, in such way that any calls to **GetInstance** that are executed on that thread, will get that same instance.

Note: This makes a per-thread lifestyle very different from the *Thread Scoped* lifestyle, as the lifetime of an instance is limited to a very clearly defined scope and usually a very short period of time, whereas a per-thread instance will live for the duration of the thread.

While designing Simple Injector, we explicitly decided not to include a Per Thread lifestyle out-of-the-box, because we want to:

- *Push developers into best practices*

The Per Thread lifestyle is very dangerous and in general you should not use it in your application, especially web applications.

This lifestyle should be considered dangerous, because it is very hard to predict what the actual lifespan of a thread is. When you create and start a thread using *new Thread().Start()*, you'll get a fresh block of thread-static memory, which means the container will create a new per-threaded instance for you. When starting threads from the thread pool using *ThreadPool.QueueUserWorkItem* however, you may get an existing thread from the pool. The same holds when running in frameworks like ASP.NET. ASP.NET pools threads to increase performance.

All this means that a thread will almost certainly outlive a web request. ASP.NET and other frameworks can run requests asynchronously meaning that a web request can be finished on a different thread to the thread the request started executing on. These are some of the problems you can encounter when working with a Per Thread lifestyle.

A web request will typically begin with a call to **GetInstance** which will load the complete object graph including any services registered with the Per Thread lifestyle. At some point during the operation the call is postponed (due to the asynchronous nature of the ASP.NET framework). At some future moment in time ASP.NET will resume processing this call on a different thread and at this point we have a problem - some of the objects in our object graph are tied up on another thread, possibly doing something else for another operation. What a mess!

Since these instances are registered as Per Thread, they are probably not suited to be used in another thread. They are almost certainly not thread-safe (otherwise they would be registered as Singleton). Since the first thread that initially started the request is already free to pick up new requests, we can run into the situation where two threads access those Per Thread instances simultaneously. This will lead to race conditions and bugs that are hard to diagnose and find.

So in general, using Per Thread is a bad idea and that's why Simple Injector does not support it. If you wish, you can always shoot yourself in the foot by implementing such a custom lifestyle, but don't blame us :-)

For registrations with thread-affinity, we advise the use of the *Thread Scoped* lifestyle.

Allow only a single constructor

Out of the box, Simple Injector only allows building up types that contain a single public constructor, because we want to adhere to the following principles:

- *Push developers into best practices*
- *Never fail silently*

Having multiple public constructors on the components that you resolve is an anti-pattern. This anti-pattern is described in more detail here[122]

This doesn't mean that it is impossible to do auto-wiring on types with multiple public constructors, but with Simple Injector this behavior will have to be *explicitly configured*.

[122] https://www.cuttingedge.it/blogs/steven/pivot/entry.php?id=97

CHAPTER 13

Legal stuff

Simple Injector License

Simple Injector is published under the MIT-license. Go here[123] to read our license. The MIT License is a free software license originating at the Massachusetts Institute of Technology (MIT). It is a permissive free software license, meaning that it permits reuse within proprietary software provided all copies of the licensed software include a copy of the MIT License terms. Such proprietary software retains its proprietary nature even though it incorporates software under the MIT License.

Contributions

Simple Injector has a strict policy for accepting contributions. Contributions are only accepted by developers who have signed our comprehensive Contributors License Agreement. This agreement helps to ensure that all of the code contributed to the Simple Injector project cannot later be claimed as belonging to any individual or group. This protects the Simple Injector project, its members and its users from any IP[124] claims.

If you would like to learn more on how to become a contributor and how to contribute code to the project, please read the *How To Contribute Code* page.

Simple Injector Trademark Policy

We, the Simple Injector Contributors, love it when people talk about Simple Injector, build businesses around Simple Injector and produce products that make life better for Simple Injector users and developers. We do, however, have a trademark, which we are obliged to protect. The trademark gives us the exclusive right to use the term to promote websites, services, businesses and products. Although those rights are exclusively ours, we are happy to give people permission to use the term under most circumstances.

[123] https://github.com/simpleinjector/SimpleInjector/blob/master/licence.txt
[124] https://en.wikipedia.org/wiki/Intellectual_property

The following is a general policy that tells you when you can refer to the Simple Injector name and logo without need of any specific permission from Simple Injector:

First, you must make clear that you are not Simple Injector and that you do not represent Simple Injector. A simple disclaimer on your home page is an excellent way of doing that.

Second, you may not incorporate the Simple Injector name or logo into the name or logo of your website, product, business or service.

Third, you may use the Simple Injector name (but not the Simple Injector logo) only in descriptions of your website, product, NuGet package, business or service to provide accurate information to the public about yourself.

Fourth, you may not use the Simple Injector graphical logo.

If you would like to use the Simple Injector name or logo for any other use, please contact us and we'll discuss a way to make that happen. We don't have strong objections to people using the name for their websites and businesses, but we do need the chance to review such use. Generally, we approve your use if you agree to a few things, mainly: (1) our rights to the Simple Injector trademark are valid and superior to yours and (2) you'll take appropriate steps to make sure people don't confuse your website, package, or product for ours. In other words, it's not a big deal, and a short conversation (usually done via email) should clear everything up in short order.

If you currently have a website that is using the Simple Injector name and you have not gotten permission from us, don't panic. Let us know, and we'll work it out, as described above.

CHAPTER 14

How to Contribute

For any contributions to be accepted you first need to print, sign, scan and email a copy of the CLA[125] to mailto:cla@simpleinjector.org[126]

For the moment we request that changes are only made after a discussion[127] and that each change has a related and assigned issue. Changes that do not relate to an approved issue may not be accepted.

Once you have completed your changes:

1. Make sure all existing and new unit tests pass.
2. Unit tests must conform to Roy Osherove's Naming Standards for Unit Tests[128] and the AAA pattern[129] must be documented explicitly in each test.
3. Make sure it compiles in both Debug and Release mode (xml comments are only checked in the release build).
4. Make sure there are no StyleCop[130] warnings {Ctrl + Shift + Y}
5. Make sure the project can be built using the **build.bat**.
6. Submit a pull request

[125] https://github.com/simpleinjector/SimpleInjector/raw/master/Simple%20Injector%20Contributor%20License%20Agreement.pdf
[126] cla@simpleinjector.org
[127] https://simpleinjector.org/community
[128] http://osherove.com/blog/2005/4/3/naming-standards-for-unit-tests.html
[129] http://c2.com/cgi/wiki?ArrangeActAssert
[130] https://visualstudiogallery.msdn.microsoft.com/cac2a05b-6eb6-4fa2-95b9-1f8d011e6cae

CHAPTER 15

Appendix

Interception Extensions

Adding interception abilities to Simple Injector.

```
using System;
using System.Diagnostics;
using System.Linq;
using System.Linq.Expressions;
using System.Reflection;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Proxies;

using SimpleInjector;

public interface IInterceptor {
    void Intercept(IInvocation invocation);
}

public interface IInvocation {
    object InvocationTarget { get; }
    object ReturnValue { get; set; }
    object[] Arguments { get; }
    void Proceed();
    MethodBase GetConcreteMethod();
}

// Extension methods for interceptor registration
// NOTE: These extension methods can only intercept interfaces, not abstract types.
public static class InterceptorExtensions {
    public static void InterceptWith<TInterceptor>(this Container container,
        Func<Type, bool> predicate)
        where TInterceptor : class, IInterceptor {
        container.Options.ConstructorResolutionBehavior.
↪GetConstructor(typeof(TInterceptor));
```

```
        var interceptWith = new InterceptionHelper() {
            BuildInterceptorExpression =
                e => BuildInterceptorExpression<TInterceptor>(container),
            Predicate = type => predicate(type)
        };

        container.ExpressionBuilt += interceptWith.OnExpressionBuilt;
    }

    public static void InterceptWith(this Container container,
        Func<IInterceptor> interceptorCreator, Func<Type, bool> predicate) {
        var interceptWith = new InterceptionHelper() {
            BuildInterceptorExpression =
                e => Expression.Invoke(Expression.Constant(interceptorCreator)),
            Predicate = type => predicate(type)
        };

        container.ExpressionBuilt += interceptWith.OnExpressionBuilt;
    }

    public static void InterceptWith(this Container container,
        Func<ExpressionBuiltEventArgs, IInterceptor> interceptorCreator,
        Func<Type, bool> predicate) {
        var interceptWith = new InterceptionHelper() {
            BuildInterceptorExpression = e => Expression.Invoke(
                Expression.Constant(interceptorCreator),
                Expression.Constant(e)),
            Predicate = type => predicate(type)
        };

        container.ExpressionBuilt += interceptWith.OnExpressionBuilt;
    }

    public static void InterceptWith(this Container container,
        IInterceptor interceptor, Func<Type, bool> predicate) {
        var interceptWith = new InterceptionHelper() {
            BuildInterceptorExpression = e => Expression.Constant(interceptor),
            Predicate = predicate
        };

        container.ExpressionBuilt += interceptWith.OnExpressionBuilt;
    }

    [DebuggerStepThrough]
    private static Expression BuildInterceptorExpression<TInterceptor>(
        Container container)
        where TInterceptor : class
    {
        var interceptorRegistration = container.GetRegistration(typeof(TInterceptor));

        if (interceptorRegistration == null) {
            // This will throw an ActivationException
            container.GetInstance<TInterceptor>();
        }

        return interceptorRegistration.BuildExpression();
    }
```

```
    private class InterceptionHelper {
        private static readonly MethodInfo NonGenericInterceptorCreateProxyMethod = (
            from method in typeof(Interceptor).GetMethods()
            where method.Name == "CreateProxy"
            where method.GetParameters().Length == 3
            select method)
            .Single();

        internal Func<ExpressionBuiltEventArgs, Expression>␣
↪BuildInterceptorExpression;
        internal Func<Type, bool> Predicate;

        [DebuggerStepThrough]
        public void OnExpressionBuilt(object sender, ExpressionBuiltEventArgs e) {
            if (this.Predicate(e.RegisteredServiceType)) {
                ThrowIfServiceTypeNotInterface(e);
                e.Expression = this.BuildProxyExpression(e);
            }
        }

        [DebuggerStepThrough]
        private static void ThrowIfServiceTypeNotInterface(ExpressionBuiltEventArgs␣
↪e) {
            // NOTE: We can only handle interfaces, because
            // System.Runtime.Remoting.Proxies.RealProxy only supports interfaces.
            if (!e.RegisteredServiceType.IsInterface) {
                throw new NotSupportedException("Can't intercept type " +
                    e.RegisteredServiceType.Name + " because it is not an interface.
↪");
            }
        }

        [DebuggerStepThrough]
        private Expression BuildProxyExpression(ExpressionBuiltEventArgs e) {
            var expr = this.BuildInterceptorExpression(e);

            // Create call to
            // (ServiceType)Interceptor.CreateProxy(Type, IInterceptor, object)
            var proxyExpression =
                Expression.Convert(
                    Expression.Call(NonGenericInterceptorCreateProxyMethod,
                        Expression.Constant(e.RegisteredServiceType, typeof(Type)),
                        expr,
                        e.Expression),
                    e.RegisteredServiceType);

            if (e.Expression is ConstantExpression && expr is ConstantExpression) {
                return Expression.Constant(CreateInstance(proxyExpression),
                    e.RegisteredServiceType);
            }

            return proxyExpression;
        }

        [DebuggerStepThrough]
        private static object CreateInstance(Expression expression) {
            var instanceCreator = Expression.Lambda<Func<object>>(expression,
```

```
                new ParameterExpression[0])
                .Compile();

            return instanceCreator();
        }
    }
}

public static class Interceptor
{
    public static T CreateProxy<T>(IInterceptor interceptor, T realInstance) =>
        (T)CreateProxy(typeof(T), interceptor, realInstance);

    [DebuggerStepThrough]
    public static object CreateProxy(Type serviceType, IInterceptor interceptor,
        object realInstance) {
        var proxy = new InterceptorProxy(serviceType, realInstance, interceptor);
        return proxy.GetTransparentProxy();
    }

    private sealed class InterceptorProxy : RealProxy {
        private static MethodBase GetTypeMethod = typeof(object).GetMethod("GetType");

        private object realInstance;
        private IInterceptor interceptor;

        [DebuggerStepThrough]
        public InterceptorProxy(Type classToProxy, object obj, IInterceptor␣
↪interceptor)
            : base(classToProxy) {
            this.realInstance = obj;
            this.interceptor = interceptor;
        }

        public override IMessage Invoke(IMessage msg) {
            if (msg is IMethodCallMessage) {
                var message = (IMethodCallMessage)msg;
                return object.ReferenceEquals(message.MethodBase, GetTypeMethod)
                    ? this.Bypass(message)
                    : this.InvokeMethodCall(message);
            }

            return msg;
        }

        private IMessage InvokeMethodCall(IMethodCallMessage msg) {
            var i = new Invocation { Proxy = this, Message = msg, Arguments = msg.
↪Args };
            i.Proceeding = () =>
                i.ReturnValue = msg.MethodBase.Invoke(this.realInstance, i.Arguments);
            this.interceptor.Intercept(i);
            return new ReturnMessage(i.ReturnValue, i.Arguments,
                i.Arguments.Length, null, msg);
        }

        private IMessage Bypass(IMethodCallMessage msg) {
            object value = msg.MethodBase.Invoke(this.realInstance, msg.Args);
            return new ReturnMessage(value, msg.Args, msg.Args.Length, null, msg);
```

```
        }

        private class Invocation : IInvocation {
            public Action Proceeding;
            public InterceptorProxy Proxy { get; set; }
            public object[] Arguments { get; set; }
            public IMethodCallMessage Message { get; set; }
            public object ReturnValue { get; set; }
            public object InvocationTarget => this.Proxy.realInstance;
            public void Proceed() => this.Proceeding();
            public MethodBase GetConcreteMethod() => this.Message.MethodBase;
        }
    }
}
```

After copying the previous code snippet to your project, you can add interception using the following lines of code:

```
// Register a MonitoringInterceptor to intercept all interface
// service types, which type name end with the text 'Repository'.
container.InterceptWith<MonitoringInterceptor>(
    serviceType => serviceType.Name.EndsWith("Repository"));

// When the interceptor (and its dependencies) are thread-safe,
// it can be registered as singleton to prevent a new instance
// from being created and each call. When the intercepted service
// and both the interceptor are both singletons, the returned
// (proxy) instance will be a singleton as well.
container.RegisterSingle<MonitoringInterceptor>();

// Here is an example of an interceptor implementation.
// NOTE: Interceptors must implement the IInterceptor interface:
private class MonitoringInterceptor : IInterceptor {
    private readonly ILogger logger;

    public MonitoringInterceptor(ILogger logger) {
        this.logger = logger;
    }

    public void Intercept(IInvocation invocation) {
        var watch = Stopwatch.StartNew();

        // Calls the decorated instance.
        invocation.Proceed();

        var decoratedType = invocation.InvocationTarget.GetType();

        this.logger.Log(string.Format("{0} executed in {1} ms.",
            decoratedType.Name, watch.ElapsedMilliseconds));
    }
}
```

Variance Extensions

Allowing Simple Injector to resolve a variant registration.

The following code snippet adds the ability to Simple Injector to resolve an assignable variant implementation, in case

the exact requested type is not registered.

```
using System;
using System.Linq;
using SimpleInjector;

public static class VarianceExtensions
{
    /// <summary>
    /// When this method is called on a container, it allows the container to map an
    /// unregistered requested (interface or delegate) type to an assignable and
    /// (interface or delegate) type that has been registered in the container. When
    /// there are multiple compatible types, an <see cref="ActivationException"/> will
    /// be thrown.
    /// </summary>
    /// <param name="ContainerOptions">The options to make the registrations in.</
↪param>
    public static void AllowToResolveVariantTypes(this ContainerOptions options) {
        var container = options.Container;
        container.ResolveUnregisteredType += (sender, e) => {
            Type serviceType = e.UnregisteredServiceType;

            if (!serviceType.IsGenericType || e.Handled) return;

            var registrations = FindAssignableRegistrations(container, serviceType);

            if (!registrations.Any()) {
                // No registration found. We're done.
            }
            else if (registrations.Length == 1) {
                // Exactly one registration. Let's map the registration to the
                // unregistered service type.
                e.Register(registrations[0].Registration);
            } else {
                var names = string.Join(", ",
                    registrations.Select(r => r.ServiceType.ToFriendlyName()));

                throw new ActivationException(string.Format(
                    "There is an error in the container's configuration. It is impos"␣
↪+
                    "sible to resolve type {0}, because there are {1} registrations "␣
↪+
                    "that are applicable. Ambiguous registrations: {2}.",
                    serviceType.ToFriendlyName(), registrations.Length, names));
            }
        };
    }

    private static InstanceProducer[] FindAssignableRegistrations(Container container,
        Type serviceType) {
        Type serviceTypeDefinition = serviceType.GetGenericTypeDefinition();

        return (
            from reg in container.GetCurrentRegistrations()
            where reg.ServiceType.IsGenericType
            where reg.ServiceType.GetGenericTypeDefinition() == serviceTypeDefinition
            where serviceType.IsAssignableFrom(reg.ServiceType)
            select reg)
            .ToArray();
```

```
    }
}
```

After copying the previous code snippet to your project, you can use the extension method as follows:

```
var container = new Container();

container.Options.AllowToResolveVariantTypes();

// IEventHandler<in TEvent>
this.container.Register<IEventHandler<CustomerMovedEvent>, CustomerMovedEventHandler>
↪();

// MoveCustomerAbroadEvent inherits from MoveCustomerEvent
var handler = this.container.GetInstance<IEventHandler<CustomerMovedAbroadEvent>>();

// Assert
Assert.IsInstanceOfType(handler, typeof(CustomerMovedEventHandler));
```

Resolution conflicts caused by dynamic assembly loading

Simple Injector can sometimes throw an exception message similar to the following when batch-registration is mixed with dynamic loading of assemblies:

Type {type name} is a member of the assembly {assembly name} which seems to have been loaded more than once. The CLR believes the second instance of the assembly is a different assembly to the first. It is this multiple loading of assemblies that is causing this issue. The most likely cause is that the same assembly has been loaded from different locations within different contexts.

What's the problem?

This exception is thrown by Simple Injector when trying to resolve a type for which it does not have an exact registration, but finds a registration for a different type with the exact same type name.

The problem is usually caused by incorrect assembly loading: the same assembly has been loaded multiple times, leading the common language runtime to believe that the multiple instances of the assembly are different assemblies. Although both assemblies contain the same type, as far as the runtime is concerned there are two different types, and they are completely unrelated. Simple Injector cannot automatically fix this problem - the runtime sees the types as unrelated and Simple Injector cannot implement an automatic conversion. Instead, Simple Injector will halt the resolution process and throw an exception.

A common reason leading to this situation is through the use of *Assembly.LoadFile*, instead of using Assembly.Load[131] or in an ASP.NET environment BuildManager.GetReferencedAssemblies()[132]. *Assembly.Load* returns an already loaded assembly even if a different path is specified, *Assembly.LoadFile* does not (depending on the current assembly loading context), it loads a duplicate instance.

So what do you need to do?

Avoid using *Assembly.LoadFile* and use *Assembly.Load* as shown in the following example:

[131] https://msdn.microsoft.com/en-us/library/x4cw969y(v=vs.110).aspx
[132] https://msdn.microsoft.com/en-us/library/system.web.compilation.buildmanager.getreferencedassemblies(v=vs.110).aspx

```
Assembly assembly = Assembly.Load(AssemblyName.GetAssemblyName("c:\..."));
```

For an ASP.NET application use *BuildManager.GetReferencedAssemblies* as shown in the following example:

```
var assemblies = BuildManager.GetReferencedAssemblies().Cast<Assembly>();
```

The *GetReferencedAssemblies* method will load all referenced assemblies located within the web application's /bin folder of the ASP.NET temp directory where the referenced assemblies are shadow copied by default.

Compiler Error: 'InjectionConsumerInfo.ServiceType' is deprecated

With Simple Injector v4, the properties **InjectionConsumerInfo.ServiceType** and **ExpressionBuildingEventArgs.RegisteredServiceType** have been marked obsolete and an exception will be thrown when these properties are called at runtime.

When using these properties the C# compiler will emit one of the following compiler errors:

Error CS0619 'InjectionConsumerInfo.ServiceType' is obsolete: 'This property has been removed. Please use ImplementationType instead.

Error CS0619 'ExpressionBuildingEventArgs.RegisteredServiceType' is obsolete: 'This property has been removed. Please use KnownImplementationType instead.

What's the problem?

The **InjectionConsumerInfo** and **ExpressionBuildingEventArgs** classes are used at a point in the pipeline where only the given implementation type is known. Although in some cases the actual service type could be determined correctly, in other cases it just contained a wrong value (of a different registration).

InjectionConsumerInfo and **ExpressionBuildingEventArgs** are used by **Registration** instances. A **Registration** instance is responsible for creating and caching a specific implementation, but it has no notion of the abstraction/service for which it is registered. This is deliberate, since a **Registration** can be reused by multiple **InstanceProducer** instances. **InstanceProducer** instances are responsible of the mapping from an abstraction to the implementation provided by a **Registration**.

The following example shows multiple registrations for the same implementation:

```
container.Register<IFoo, FooBar>(Lifestyle.Singleton);
container.Register<IBar, FooBar>(Lifestyle.Singleton);
```

These registrations result in two separate **InstanceProducer** instances (one for *IFoo* and one for *IBar*), but both use the same **Registration** instance for *FooBar*. This ensures that a single instance of *FooBar* will exist within that **Container** instance.

The result of this is that when *FooBar* is built, there are multiple possible abstractions. However, since a **Registration** is only built once, it could only be aware of the first service type it is built for, which could be either *IFoo* or *IBar*.

Since the use of these properties was ambiguous and could lead to fragile configurations, they needed to be removed.

So what should I do instead?

In general, try to use the **ImplementationType** or **KnownImplementationType** properties instead. If required, you can extract the actual service type from the implementation type using the new **IsClosedTypeOf**, **GetClosedTypeOf** and **GetClosedTypesOf** extension methods.

The following snippet shows how to use the **GetClosedTypeOf** method:

```
container.RegisterConditional(typeof(IRepository<,>), typeof(ReadOnlyRepo<>), context␣
↪=>
{
    var implementation = context.Consumer.ImplementationType;
    var serviceType = implementation.GetClosedTypeOf(typeof(IRepository<>));
    var entityType = serviceType.GetGenericArguments()[0];
    return typeof(IReadOnlyEntity).IsAssignableFrom(entityType);
});
```

The **GetClosedTypeOf** will throw an exception when the **ImplementationType** implements more than one closed-generic version of the supplied open-generic type. In case multiple closed-generic abstractions are expected, **GetClosedTypesOf** can be used. It returns a *Type* array.

CHAPTER 16

Indices and tables

- genindex
- modindex
- search