DI in Scala: guide

Dependency Injection in Scala using MacWire

Home
MacWire on GitHub
Adam Warski's blog
Tweet

Table of contents

Introduction

↟ Back to top ↡ Comments

Dependency Injection (DI) is a popular pattern which encourages loose coupling between a services' clients and service implementations.

This guide describes how to do Dependency Injection using the Scala language constructs as much as possible, while remaining practical, with the help of the MacWire library where necessary.

Dependency Injection is a *simple* concept, and it can be implemented using relatively few *simple* constructs. We should avoid over-complicating and over-using containers or frameworks, without thoroughly analysing the costs.

> This guide is available on GitHub, so if you think something is missing, not clear enough, or can be done better, don't hesitate and send a pull request!

What is Dependency Injection?

DI is all about decoupling client and service code (the client may happen to be another service). Services need to expose information on what dependencies they need. Instead of creating dependent service implementations inside the service itself, *references* to dependent services are "injected". This makes the code easier to understand, more testable and more reusable.

The means of injecting the dependencies vary from approach to approach, but the one we will be using here is passing dependencies through constructor parameters. Other possibilities include setter/field injection, or using a service locator. Hence, the essence of DI can be summarised as *using constructor parameters*.

A very important aspect of DI is Inversion of Control. The service implementations have to be created "outside" the services, e.g. by a container or some external wiring code. Using `new` directly to create a dependency is not allowed inside a service.

If you are not yet sold on DI, I recommend reading the motivation behind Guice. It uses Java as the base language, but the ideas are the same and apply universally.

Other approaches

There are numerous frameworks and approaches implementing DI in various languages and on various platforms. Below are a couple of alternatives to the pure Scala+MacWire approach presented here.

Using frameworks:

- Subcut: mix of the service locator/dependency injection patterns
- Scaldi: similar to Subcut
- Spring: Spring is a very popular Java DI framework (and much more). It also works with Scala.
- Guice: another popular Java DI framework

Using pure Scala:

- Cake pattern
- Reader monad

Running example

To provide code examples throughout the guide, we will need a running example.

Suppose we are creating a system for a train station. Our goal is to call the method to prepare (load, compose cars) and dispatch the next train. In order to do that, we need to instantiate the following service classes:

```
class PointSwitcher()
class TrainCarCoupler()
class TrainShunter(
   pointSwitcher: PointSwitcher,
```

```
    trainCarCoupler: TrainCarCoupler)

class CraneController()
class TrainLoader(
    craneController: CraneController,
    pointSwitcher: PointSwitcher)

class TrainDispatch()

class TrainStation(
    trainShunter: TrainShunter,
    trainLoader: TrainLoader,
    trainDispatch: TrainDispatch) {

    def prepareAndDispatchNextTrain() { ... }
}
```

The dependencies of each class are expressed as constructor parameters. The dependencies form an object graph, which needs to be wired.

Manual Dependency Injection

↑ Back to top ↓ Comments

An approach that is often dismissed too easily is doing dependency injection by hand. While this requires a bit more typing, when doing things manually you are free from any constraints that a framework may impose, and hence a lot of flexibility is gained.

When using DI, we have to somehow create (wire) the object graph, that is create instances of the given classes with the appropriate dependencies. When using a framework, this task is delegated to a container. But nothing keeps us from simply writing the code!

The object graph should be created as late as possible. This may be, for example, the "end of the world", that is the main class of our application. If you have used DI containers before, when using manual DI with or without MacWire, you will soon re-discover the benefits of having a main method in your application.

To be more specific, here's a manual-DI version for our example:

```
object TrainStation extends App {
    val pointSwitcher = new PointSwitcher()
    val trainCarCoupler = new TrainCarCoupler()
    val trainShunter = new TrainShunter(
        pointSwitcher, trainCarCoupler)

    val craneController = new CraneController()
    val trainLoader = new TrainLoader(
        craneController, pointSwitcher)

    val trainDispatch = new TrainDispatch()

    val trainStation = new TrainStation(
      trainShunter, trainLoader, trainDispatch)

    trainStation.prepareAndDispatchNextTrain()
}
```

Advantages of Manual DI

The first advantage of the approach presented above is type-safety: the dependency resolution is done at compile-time, so you can be sure that all dependencies are met.

No run-time reflection is needed, which has a slight startup-time benefit (no need to scan the classpath), but also removes a lot of "magic" from the code. There are no annotations to scan for. We are only using plain Scala, and constructor parameters. It is possible to navigate to the place where the instance is created. The process of creating the object graph is clear. The application is also simple to use, and can be easily packaged e.g. as a fat-jar. No containers to start, no frameworks to fight against.

If creating an instance of some object is complex, or choosing an implementation depends on some configuration, thanks to the flexibility of manual DI, we can easily run arbitrary code which should compute the dependency to use.

`val` vs. `lazy val`

Defining dependencies using `val`s has one drawback: if a dependency is used, before being initialised, it will be `null` when referenced. That is because `val`s are calculated from top to bottom.

This can be solved by using `lazy val`s, which are calculated on-demand, and the right initialisation order will be calculated automatically.

Hence our manual-DI example becomes:

```
object TrainStation extends App {
   lazy val pointSwitcher = new PointSwitcher()
   lazy val trainCarCoupler = new TrainCarCoupler()
   lazy val trainShunter = new TrainShunter(
      pointSwitcher, trainCarCoupler)

   lazy val craneController = new CraneController()
   lazy val trainLoader = new TrainLoader(
      craneController, pointSwitcher)

   lazy val trainDispatch = new TrainDispatch()

   lazy val trainStation = new TrainStation(
      trainShunter, trainLoader, trainDispatch)

   trainStation.prepareAndDispatchNextTrain()
}
```

Using MacWire for wiring

↑ Back to top ↓ Comments

Manual DI isn't of course a silver bullet. Manually writing new instance creation code for each class, using the correct parameters may be tedious.

That is where MacWire and the `wire` method can help. `wire` is a Scala Macro, which generates the new instance creation code.

After using MacWire, the code becomes simpler:

```
object TrainStation extends App {
   lazy val pointSwitcher = wire[PointSwitcher]
   lazy val trainCarCoupler = wire[TrainCarCoupler]
   lazy val trainShunter = wire[TrainShunter]

   lazy val craneController = wire[CraneController]
   lazy val trainLoader = wire[TrainLoader]
   lazy val trainDispatch = wire[TrainDispatch]

   lazy val trainStation = wire[TrainStation]

   trainStation.prepareAndDispatchNextTrain()
}
```

If a new dependency is added to a service or if the order of parameters changes, the object-graph wiring code doesn't have to be altered; the macro will take care of that. Only when introducing a new service, it must be added to the list.

The new instance creation code is generated by `wire` at *compile time*, so if you compare the byte code generated by both examples, it will be identical. The generated code is type-checked in the usual way, so we keep the type-safety of the manual approach.

Usage of the `wire` macro can be mixed with creating new instances by hand; this may be needed if, as discussed earlier, creating a new instance isn't that straightforward.

To access `wire`, you should import `com.softwaremill.macwire._`. For details on how to integrate MacWire into your project, see the GitHub page.

How `wire` works

Given a class, the `wire` macro first tries to find a constructor annotated with `@Inject`, then the (non-private) primary constructor, and finally an `apply` method in the companion object, to determine the dependencies needed. For each dependency it then looks for a value which conforms to the parameter's type, in the enclosing method/class/object:

- first it tries to find a unique value declared as a value in the current block, argument of enclosing methods and anonymous functions
- then it tries to find a unique value declared or imported in the enclosing type
- finally it tries to find a unique value in parent types (traits/classes)
- if the parameter is marked as implicit, it is ignored by MacWire and handled by the normal implicit resolution mechanism

Here value can be either a `val`, `lazy val` or a no-parameter `def`, as long as the return type matches.

A compile-time error occurs if:

- there are multiple values of a given type declared in the enclosing block/method/function's arguments list, enclosing type or its parents
- parameter is marked as implicit and implicit lookup fails to find a value
- there is no value of a given type

Using implicit parameters

A similar effect to the one described above can be achieved by using implicit parameters and implicit values. If all constructor parameters are marked as `implicit`, and all instances are marked as `implicit` when the object graph is wired, the Scala compiler will create the proper constructor calls.

The class definitions then become:

```
class PointSwitcher()
class TrainCarCoupler()
class TrainShunter(
   implicit
   pointSwitcher: PointSwitcher,
   trainCarCoupler: TrainCarCoupler)

class CraneController()
class TrainLoader(
   implicit
   craneController: CraneController,
   pointSwitcher: PointSwitcher)

class TrainDispatch()

class TrainStation(
   implicit
   trainShunter: TrainShunter,
   trainLoader: TrainLoader,
   trainDispatch: TrainDispatch) {

   def prepareAndDispatchNextTrain() { ... }
}
```

And the wiring:

```
object TrainStation extends App {
   implicit lazy val pointSwitcher = new PointSwitcher
   implicit lazy val trainCarCoupler = new TrainCarCoupler
   implicit lazy val trainShunter = new TrainShunter

   implicit lazy val craneController = new CraneController
   implicit lazy val trainLoader = new TrainLoader

   implicit lazy val trainDispatch = new TrainDispatch

   implicit lazy val trainStation = new TrainStation

   trainStation.prepareAndDispatchNextTrain()
}
```

However, using implicits like that has two drawbacks. First of all, it is intrusive, as you have to mark the constructor parameter list of each class to be wired as `implicit`. That may not be desireable, and can cause the person reading the code to wonder why the parameters are implicit.

Secondly, implicits are used in many other places in Scala for other, rather different purposes. Adding a large number of implicits as described here may lead to confusion. Still, such a style may be a perfect fit in some use-cases, of course!

Simple scoping

↑ Back to top ↓ Comments

So far all of the dependencies have been declared as `lazy val` s, making them essentially singletons in the scope of a single app usage. Note that these aren't singletons in the global sense, as we can create multiple copies of the object graph.

However, sometimes we want to create new instances of a dependency for each usage (sometimes called the "dependent scope"). To achieve this, we can simply declare the instance as a `def` , instead of a `lazy val` . If, for example, we needed a new instance of the train dispatcher each time, the code would become:

```
object TrainStation extends App {
   lazy val pointSwitcher = wire[PointSwitcher]
   lazy val trainCarCoupler = wire[TrainCarCoupler]
   lazy val trainShunter = wire[TrainShunter]

   lazy val craneController = wire[CraneController]
   lazy val trainLoader = wire[TrainLoader]

   // note the def instead of lazy val
   def trainDispatch = wire[TrainDispatch]

   // the stations share all services except the train dispatch,
   // for which a new instance is create on each usage
   lazy val trainStationEast = wire[TrainStation]
   lazy val trainStationWest = wire[TrainStation]

   trainStationEast.prepareAndDispatchNextTrain()
   trainStationWest.prepareAndDispatchNextTrain()
}
```

Hence, using Scala constructs we can implement two scopes: singleton and dependent.

Modularising object graph creation

↑ Back to top ↓ Comments

Thin cake pattern

At some point, creating the whole object graph at "the end of the world" will become unpractical, and the code large and hard to read. We should then somehow divide it to smaller pieces. Luckily, Scala's `trait` s fit perfectly for that task; they can be used to split the object graph creation code.

In each trait, which for purpose of this task is also called a "module", part of the object graph is created. Everything is later re-combined by putting all the necessary traits together.

There may be various rules on how to divide code into modules. A good place to start is to consider creating a pre-wired module per-package. Each package should contain a group of classes sharing or implementing some specific functionality. Most probably these classes cooperate in some way, and hence can be wired.

The additional benefit of shipping a package not only with the code, but also with a wired object graph fragment, is that it is more clear how the code should be used. There are no requirements on actually using the wired module, so if needed, wiring can be done in a different way.

However, such modules can't usually exist stand-alone: very often they will depend on some classes from other modules. There are two ways of expressing dependencies.

Expressing dependencies via abstract members

As each module is a trait, it is possible to leave some dependencies undefined, as abstract members. Such abstract members can be used when wiring (either manually or using the `wire` macro), but the specific implementation doesn't have to be given.

When all the modules are combined in the end application, the compiler will verify that all such dependencies defined as abstract members are defined.

Note that we can declare all abstract members as `def` s, as they can be later implemented as `val` s, `lazy val` s, or left as `def` s. Using a `def` keeps all options possible.

The wiring for our example code can be divided as follows; the classes are now grouped into packages:

```
package shunting {
   class PointSwitcher()
   class TrainCarCoupler()
   class TrainShunter(
      pointSwitcher: PointSwitcher,
      trainCarCoupler: TrainCarCoupler)
}

package loading {
   class CraneController()
   class TrainLoader(
      craneController: CraneController,
      pointSwitcher: PointSwitcher)
}

package station {
   class TrainDispatch()

   class TrainStation(
      trainShunter: TrainShunter,
      trainLoader: TrainLoader,
      trainDispatch: TrainDispatch) {

      def prepareAndDispatchNextTrain() { ... }
   }
}
```

Each package has a corresponding trait-module. Note that the dependency between the `shunting` and `loading` packages is expressed using an abstract member:

```
package shunting {
   trait ShuntingModule {
      lazy val pointSwitcher = wire[PointSwitcher]
      lazy val trainCarCoupler = wire[TrainCarCoupler]
      lazy val trainShunter = wire[TrainShunter]
   }
}

package loading {
   trait LoadingModule {
```

```
      lazy val craneController = wire[CraneController]
      lazy val trainLoader = wire[TrainLoader]

      // dependency of the module
      def pointSwitcher: PointSwitcher
   }
}

package station {
   trait StationModule {
      lazy val trainDispatch = wire[TrainDispatch]

      lazy val trainStation = wire[TrainStation]

      // dependencies of the module
      def trainShunter: TrainShunter
      def trainLoader: TrainLoader
   }
}

object TrainStation extends App {
   val modules = new ShuntingModule
      with LoadingModule
      with StationModule

   modules.trainStation.prepareAndDispatchNextTrain()
}
```

To implement dependencies this way a consistent naming convention is needed, as the abstract member is reconciled with the implementation by-name. Naming the values same as the classes, but with the initial letter lowercase is a good example of such a convention.

This approach is in some parts similar to the Cake Pattern, hence the name: **Thin Cake Pattern**.

Expressing dependencies via self-types

Another way of expressing dependencies is by using self-types or extending other trait-modules. This way creates a much stronger connection between the two modules, instead of the looser coupled abstract member approach, however in some situations is desirable (e.g. when having a module-interface with multiple implementations, see below).

For example, we could express the dependency between the shunting and loading modules and the station module by extending trait-module, instead of using the abstract members:

```
package shunting {
   trait ShuntingModule {
      lazy val pointSwitcher = wire[PointSwitcher]
      lazy val trainCarCoupler = wire[TrainCarCoupler]
      lazy val trainShunter = wire[TrainShunter]
   }
}

package loading {
   trait LoadingModule {
      lazy val craneController = wire[CraneController]
      lazy val trainLoader = wire[TrainLoader]

      // dependency expressed using an abstract member
      def pointSwitcher: PointSwitcher
```

```
    }
}

package station {
    // dependencies expressed using extends
    trait StationModule extends ShuntingModule with LoadingModule {
        lazy val trainDispatch = wire[TrainDispatch]

        lazy val trainStation = wire[TrainStation]
    }
}

object TrainStation extends App {
    val modules = new ShuntingModule
        with LoadingModule
        with StationModule

    modules.trainStation.prepareAndDispatchNextTrain()
}
```

A very similar effect would be achieved by using a self-type.

This approach can also be useful to create bigger modules out of multiple smaller ones, without the need to re-express the dependencies of the smaller modules. Simply define a bigger-module-trait extending a number of smaller-module-traits.

Composing modules

Modules can be also combined using composition, that is you can nest modules as members and use dependencies defined in the nested modules to wire objects.

For example, we can add a plugin to our train management application which will allow gathering statistics:

```
package stats {
    class LoadingStats(trainLoader: TrainLoader)
    class ShuntingStats(trainShunter: TrainShunter)

    class StatsModule(
        shuntingModule: ShuntingModule,
        loadingModule: LoadingModule) {

        import shuntingModule._
        import loadingModule._

        lazy val loadingStats = wire[LoadingStats]
        lazy val shuntingStats = wire[ShuntingStats]
    }
}
```

Note the `import` statements, which bring any dependencies defined in the nested modules into scope.

This can be further shortened by using an experimental `@Module` annotation for module `trait` s/ `class` es; members of nested modules with that annotaiton will be taken into account automatically during wiring:

```
package loading {
    @Module
    trait LoadingModule { ... }
}
```

```
package shunting {
   @Module
   trait ShuntingModule { ... }
}

package stats {
   class LoadingStats(trainLoader: TrainLoader)
   class ShuntingStats(trainShunter: TrainShunter)

   class StatsModule(
      shuntingModule: ShuntingModule,
      loadingModule: LoadingModule) {

      lazy val loadingStats = wire[LoadingStats]
      lazy val shuntingStats = wire[ShuntingStats]
   }
}
```

In this scenario no imports are necessary.

Multiple implementations

↑ Back to top ↓ Comments

Often there are cases when we have multiple implementations for some functionality, and we need to choose one depending on configuration. This can be modelled in at least two ways.

Firstly, we can have a single module, which contains conditional logic choosing the right implementation. Suppose we have two options for train shunting, either the traditional one, or using teleportation, and a config flag:

```
package shunting {
   trait TrainShunter

   class PointSwitcher()
   class TrainCarCoupler()
   class TraditionalTrainShunter(
      pointSwitcher: PointSwitcher,
      trainCarCoupler: TrainCarCoupler)
      extends TrainShunter

   class TeleportingTrainShunter() extends TrainShunter

   trait ShuntingModule {
      lazy val pointSwitcher = wire[PointSwitcher]
      lazy val trainCarCoupler = wire[TrainCarCoupler]

      lazy val trainShunter = if (config.modern) {
         wire[TeleportingTrainShunter]
      } else {
         wire[TraditionalTrainShunter]
      }

      def config: Config
   }
}
```

Secondly, a module can have multiple implementations. In such a case, we can create an interface-module containing only abstract members, which are implemented in the proper modules. Such an interface-module can also be very

useful for expressing dependencies (without relying on a naming convention), and creating very strong links:

```
package shunting {
   trait TrainShunter

   class PointSwitcher()
   class TrainCarCoupler()
   class TraditionalTrainShunter(
      pointSwitcher: PointSwitcher,
      trainCarCoupler: TrainCarCoupler)
      extends TrainShunter

   class TeleportingTrainShunter() extends TrainShunter

   trait ShuntingModule {
      lazy val pointSwitcher = wire[PointSwitcher]

      def trainShunter: TrainShunter
   }

   trait TraditionalShuntingModule extends ShuntingModule {
      lazy val trainCarCoupler = wire[TrainCarCoupler]
      lazy val trainShunter = wire[TraditionalTrainShunter]
   }

   trait ModernShuntingModule extends ShuntingModule {
      lazy val trainShunter = wire[TeleportingTrainShunter]
   }
}

// ...

object TrainStation extends App {
   val traditionalModules = new TraditionalShuntingModule
      with LoadingModule
      with StationModule

   val modernModules = new ModernShuntingModule
      with LoadingModule
      with StationModule

   traditionalModules.trainStation.prepareAndDispatchNextTrain()
   modernModules.trainStation.prepareAndDispatchNextTrain()
}
```

The downside of this approach is that the module stack must be known at compile time (cannot be chosen dynamically). While it is possible to create 2 or 4 different stacks for a couple of config options, with increasing config options the number of stacks grows exponentially.

The interface-trait-module and implementation-trait-module is in fact part of the approach taken by the Cake Pattern for expressing dependencies. However this results in quite a lot of boilerplate code, so it's good to use only when needed.

Testing

↟ Back to top ↓ Comments

Individual components can be tested by providing mock/stub implementations of some of the dependencies.

Moreover, when using the thin cake pattern, modules can be integration-tested, using the wiring defined in the module.

Of course, we will need to provide some implementation (again, can be a mock/stub), for any dependencies expressed as abstract members. However, it is also possible to override some of the dependencies, to provide alternative implementations for testing. These implementations will be used to wire the graph fragment defined in the module.

For example, to test our shunting module, we could mock the point switcher, which interacts with some external systems, and write an integration test:

```
// main code
package shunting {
   trait ShuntingModule {
      lazy val pointSwitcher = wire[PointSwitcher]
      lazy val trainCarCoupler = wire[TrainCarCoupler]
      lazy val trainShunter = wire[TrainShunter]
   }
}

// test
class ShuntingModuleItTest extends FlatSpec {
   it should "work" in {
      // given
      val mockPointSwitcher = mock[PointSwitcher]

      // when
      val moduleToTest = new ShuntingModule {
         // the mock implementation will be used to wire the graph
         override lazy val pointSwitcher = mockPointSwitcher
      }
      moduleToTest.trainShunter.shunt()

      // then
      verify(mockPointSwitcher).switch(...)
   }
}
```

Interceptors

↑ Back to top ↓ Comments

Interceptors are very useful for implementing cross-cutting concerns, and are a part of almost every DI framework/container. While there's no direct support for interceptors in Scala, with a thin library layer (provided by MacWire), it is easy to write and use interceptors.

Using interceptors is a two-step process. First, we have to declare *what* should be intercepted. Ideally, this shouldn't involve the implementation of the interceptor in any way. Secondly, we have to define what the interceptor does - the behaviour.

To implement the first part, we will define an abstract interceptor, and apply it to selected values. Let's say that we want to audit all point switches and car couplings events to some external system. To do that, we need to intercept all method calls on the `PointSwitcher` and `TrainCarCoupler` services:

```
package shunting {
   trait ShuntingModule {
      lazy val pointSwitcher: PointSwitcher =
            logEvents(wire[PointSwitcher])
      lazy val trainCarCoupler: TrainCarCoupler =
            logEvents(wire[TrainCarCoupler])
      lazy val trainShunter = wire[TrainShunter]

      def logEvents: Interceptor
   }
}
```

We have *declared* that we want to apply the `logEvents` interceptor to the `pointSwitcher` and `trainCarCoupler` services. Note that so far the implementation hasn't been mentioned in any way. We are only using the abstract `Interceptor` trait, which has an `apply` method, returning an instance of the same type, as passed to it through the parameter.

At some point we of course have to specify the implementation. We can do this as late as possible, the last point being the main entry point to the application:

```
object TrainStation extends App {
   val modules = new ShuntingModule
      with LoadingModule
      with StationModule {

      lazy val logEvents = ProxyingInterceptor { ctx =>
         println("Calling method: " + ctx.method.getName())
         ctx.proceed()
      }
   }

   modules.trainStation.prepareAndDispatchNextTrain()
}
```

Here we have specified that we want to create a proxying interceptor (which will create a Java proxy), with the given behaviour on method invocation. Note that we could use any of the services defined in the modules, when handling the proxied call.

For testing, it may be useful to skip interceptors. This can also easily be done by providing a no-op interceptor implementation:

```
class ShuntingModuleItTest extends FlatSpec {
   it should "work" in {
      // given
      val moduleToTest = new ShuntingModule {
         lazy val logEvents = NoOpInterceptor
      }

      // ...
   }
}
```

Advanced scoping

↑ Back to top ↓ Comments

Especially in web applications, it is useful to have scopes other than singleton and dependent - e.g. scopes tied to the duration of a request, or scopes tied to a user sessions.

Like interceptors, a lot of the DI containers/frameworks contain support for different scopes. They may seem "magical", however they are in fact pretty simple.

MacWire contains a general skeleton for defining scopes, similar to interceptors. The `Scope` trait defines two methods: `apply`, which when applied to an instance should create a scoped value, and `get`, to get the current underlying value of the scope. The scope's life cycle is entirely managed by the implementor.

Similarly to interceptors, usage of scopes is declarative, by using an abstract `Scope` value, and the definition can be provided as late as in the main application entry point.

For example, in a Java servlet-based web project, to make the train dispatch session-scoped (new dispatch for each session), we would first need to declare the usage of the session scope:

```
package station {
   trait StationModule extends ShuntingModule with LoadingModule {
      lazy val trainDispatch: TrainDispatch =
            session(wire[TrainDispatch])
      lazy val trainStation: TrainStation =
            wire[TrainStation]

      def session: Scope
   }
}
```

When starting the application, we need to provide the implementation of the scope. Two implementations are shipped by default, a `NoOpScope` (useful for testing), and a `ThreadLocalScope`, which holds the "current" scoped value in a thread-local variable (hence this implementation is only useful for synchronous web frameworks); the thread-local scope needs to be associated with a storage before each request:

```
object TrainStation extends App {
   val modules = new ShuntingModule
      with LoadingModule
      with StationModule {

      lazy val session = new ThreadLocalScope
   }

   // implement a filter which attaches the session to the scope
   // use the filter in the server

   modules.trainStation.prepareAndDispatchNextTrain()
}
```

For an example session scope implementation, see the MacWire site.

Factories

↑ Back to top ↓ Comments

Factories as functions

In the simplest form, a factory is a function: a parametrized way to create an object. You can also think of it as a partially wired dependency.

Let's say the `TrainLoader` additionally requires a `CarLoader` dependency parametrized by `CarType` (the type of the train car to load: coal, refrigerated, chemical etc.). As a convenience, we can create a type alias for the function, but that is entirerly optional. The definition of the `TrainLoader` class now becomes:

```
type CarLoaderFactory = CarType => CarLoader

class TrainLoader(
   craneController: CraneController,
   pointSwitcher: PointSwitcher,
   carLoaderFactory: CarLoaderFactory)
```

We can now wire `TrainLoader` as usual (either manually or using MacWire), however of course we need a dependency of type `CarLoaderFactory` defined somewhere; extending the `LoadingModule` we get:

```
trait LoadingModule {
   lazy val craneController = wire[CraneController]
   lazy val trainLoader = wire[TrainLoader]

   lazy val carLoaderFactory = (ct: CarType) => wire[CarLoader]
   // the above wire will expand to: new CarLoader(ct). Can also
   // be any other logic to instantiate a CarLoader

   // dependency of the module
   def pointSwitcher: PointSwitcher
}
```

Factories in trait-modules

You can also create factories by defining methods directly in the trait-modules. The method parameters will be also used for wiring when using `wire[]` .

Let's say the `TrainStation` requires a name:

```
class TrainStation(
   name: Name,
   trainShunter: TrainShunter,
   trainLoader: TrainLoader,
   trainDispatch: TrainDispatch) {

   def prepareAndDispatchNextTrain() { ... }
}
```

We can expose a parametrized train station in the module by defining a method with a name parameter:

```
trait StationModule {
      def trainStation(name: Name) = wire[TrainStation]

      lazy val trainDispatch = wire[TrainDispatch]

      // dependencies of the module
```

```
        def trainShunter: TrainShunter
        def trainLoader: TrainLoader
    }
```

When retrieving an instance of a train station from the combined modules, we now have to provide a name. The method parameters, together with other dependencies looked up in the usual way, will be used to create a `TrainStation` instance.

Wiring using factory methods

Using MacWire, it is also possible to wire objects with a factory method instead of a constructor. For example, suppose that instances of `TrainLoader` needs to be created using a special method which passes some numeric parameters to the crane; in such situations, we can use `wireWith`:

```
package loading {
   class TrainLoader(
      craneController: CraneController,
      pointSwitcher: PointSwitcher,
      xAxisCoefficient: Double,
      yAxisCoefficient: Double)

   object TrainLoader {
      def createDefault(
         craneController: CraneController,
         pointSwitcher: PointSwitcher) =
         new TrainLoader(craneController, pointSwitcher,
            10.0, 12.5)
   }

   trait LoadingModule {
      lazy val craneController = wire[CraneController]

      lazy val trainLoader = wireWith(TrainLoader.createDefault)

      // dependency of the module
      def pointSwitcher: PointSwitcher
   }
}
}
```

Accessing the object graph dynamically

↑ Back to top ↓ Comments

While it would be great to be able to define in a type-safe way the whole object graph for an application upfront, there are cases when it is necessary to access and extend it dynamically.

First use-case is when integrating with web frameworks. There it is often needed to access a wired instance by-class. Second use-case is dynamically creating instances of classes, which names are only known at run-time, such as plugins.

Both of these use-cases are realised by the `Wired` class, which can be created given an instance of a module, containing the object graph, using the `wiredInModule` macro. Any `val`s, `lazy val`s and parameter-less `def`s will be available.

If our train station management application had a plugin system, which could use any of the dependencies in the object graph, we could instantiate the plugins as follows:

```
trait TrainStationPlugin {
   def init(): Unit
}

object TrainStation extends App {
   val modules = new ShuntingModule
      with LoadingModule
      with StationModule

   val wired = wiredInModule(modules)

   val plugins = config.pluginClasses.map { pluginClass =>
      wired
         .wireClassInstanceByName(pluginClass)
         .asInstanceOf[TrainStationPlugin]
   }

   plugins.foreach(_.init())

   modules.trainStation.prepareAndDispatchNextTrain()
}
```

An instance of `Wired` can be also extended with new instances and instance factories, using the `withInstances` and `withInstanceFactory` methods.

For an example of integrating Dependency Injection and MacWire with Play! Framework, see the Play+MacWire activator.

Multiple instances

↟ Back to top ↓ Comments

In some cases we have a couple of objects of the same type that we want to use as dependencies. For example, assume that our train station now needs two train loaders, one for regular freight, one for liquid freight:

```
class TrainStation(
   trainShunter: TrainShunter,
   regularTrainLoader: TrainLoader,
   liquidTrainLoader: TrainLoader,
   trainDispatch: TrainDispatch) { ... }
```

In our wiring code we then need to create the two instances:

```
lazy val regularTrainLoader = new TrainLoader(...)
lazy val liquidTrainLoader = new TrainLoader(...)
lazy val trainStation = new TrainStation(
   trainShunter,
   regularTrainLoader,
   liquidTrainLoader)
```

This is a perfectly good solution if we are using manual dependency injection - everything works as expected. One downside is that it's not entirely type-safe: if we

mix-up the regular and liquid train loaders when passing them as arguments to the `TrainStation` constructor, everything will still compile just fine.

However, if we try to use MacWire (or implicits), compilation will fail: MacWire has no chances of knowing, which instance should be used where. We need to somehow differentiate the instances so that the compiler will be able to tell which goes where.

Using different types

The first solution is to give the two dependencies distinct types, however of course that is not always possible. In our case, these might be for example traits, or subclasses:

```
class TrainStation(
   trainShunter: TrainShunter,
   regularTrainLoader: TrainLoader with Regular,
   liquidTrainLoader: TrainLoader with Liquid,
   trainDispatch: TrainDispatch) { ... }

lazy val regularTrainLoader = new TrainLoader(...) with Regular
lazy val liquidTrainLoader = new TrainLoader(...) with Liquid
lazy val trainStation = wire[TrainStation]
```

This is also a type-safe solution, we are now **not** able to mix up the two dependencies when passing them as arguments to `TrainStation`.

Using qualifiers/tags

Another solution is to use tagging from MacWire, which is inspired by the work of Miles Sabin and what is present in Scalaz.

A tag is a Scala trait, usually an empty one. An instance of type `X` tagged with tag `T` has type `X @@ T`, or `Tagged[X, T]`, depending which syntax you prefer. You can add tags to instances using the `x.taggedWith[T]` method available on any type. All these declarations can be brough into scope by importing `com.softwaremill.macwire._`.

The tags follow proper subtyping rules: `X <: X @@ T`, so you can use a tagged instance if an untagged one is required. On the other hand, `X @@ T1` is not a subtype of `X @@ T2` if `T1` and `T2` are distinct, so you are safe from using the wrong instance with the wrong tag.

Tags incur zero runtime overhead, they are a purely compile-time construct.

Our train station dependencies and wiring can then be expressed in the following way:

```
trait Regular
trait Liquid

class TrainStation(
   trainShunter: TrainShunter,
   regularTrainLoader: TrainLoader @@ Regular,
   liquidTrainLoader: TrainLoader @@ Liquid,
   trainDispatch: TrainDispatch) { ... }
```

```
lazy val regularTrainLoader = wire[TrainLoader].taggedWith[Regular]
lazy val liquidTrainLoader = wire[TrainLoader].taggedWith[Liquid]
lazy val trainStation = wire[TrainStation]
```

Note that you can use any form of "tagging" instances (e.g. the one from Scalaz), MacWire does not require to use the tags from its distribution. The only thing MacWire does during wiring is regular Scala subtype checks.

DI in Akka

↑ Back to top ↓ Comments

Akka is a very popular toolkit for bulding concurrent, "reactive" applications. The main building block used in Akka-based systems are actors. A common question when using Akka is "how do I do dependency injection with actors?"; typical use-case is passing a datasource to an actor.

There's a bit dated article on Akka's blog focusing primarily on Spring. What if we are using Manual DI or MacWire?

Injecting services

The current best-practice for defining actor instantion code, when the actor instance needs some external dependencies, is to provide a method in the companion object returning the `Props` describing how to create a new actor instance (remember, actor instances can be created multiple times, even if it's a "singleton" actor!):

```
import akka.actor.{Actor, Props}

class ReactiveTrainDispatch(
   trainLoader: TrainLoader,
   trainDispatch: TrainDispatch) extends Actor {

   def receive = ...
}

object ReactiveTrainDispatch {
   def props(trainLoader: TrainLoader, trainDispatch: TrainDispatch)
 =
      Props(new ReactiveTrainDispatch(trainLoader, trainDispatch))
}
```

When using MacWire, we could already simplify this code a bit by using `wire` inside of `Props` : `Props(wire[ReactiveTrainDispatch])` . However, we still need to repeat all the dependencies in the `props` method signature.

An alternative is to move the props-creating code into a trait-module (assuming that `ReactiveTrainDispatch` lives in the `station` package):

```
class ReactiveTrainDispatch(
   trainLoader: TrainLoader,
   trainDispatch: TrainDispatch) extends Actor {

   def receive = ...
}

trait StationModule {
   lazy val reactiveTrainDispatchProps =
```

```
      Props(wire[ReactiveTrainDispatch])

   // as in the previous examples
   lazy val trainDispatch = wire[TrainDispatch]
   lazy val trainStation = wire[TrainStation]

   // dependencies of the module
   def trainShunter: TrainShunter
   def trainLoader: TrainLoader
}
```

Going further, if the `ActorSystem` is also a value defined in the modules, we can create an actor factory without the need for an intermediate props value (why a factory? As we may want to do an explicit call to create a single or multiple actors):

```
class ReactiveTrainDispatch(
   trainLoader: TrainLoader,
   trainDispatch: TrainDispatch) extends Actor {

   def receive = ...
}

trait StationModule {
   def createReactiveTrainDispatch =
      actorSystem.actorOf(Props(wire[ReactiveTrainDispatch]))
   // actor system module dependency

   def actorSystem: ActorSystem

   // as previously
   // ...
}
```

Injecting other actors

Apart from services, actors often depend on other actors, holding references to them via `ActorRef` s. The most common way to obtain references to other actors is by sending them via messages. If, however, you want to "inject" an `ActorRef` into your actor at the time of creation, you can do it via a constructor as well.

There's a problem of course with multiple actor references as they all have the same type. With manual DI things are easy - we can just pass the `ActorRef` we need as the proper constructor argument. If, however, we are using MacWire, or if we want the manual approach to be more typesafe, we can use tagging.

Again, here we are modelling actor creation as a factory, as actors are typically created explicitly when needed, unlike the "service" object graph, but we could model them as all other dependencies as well:

```
class ReactiveTrainDispatch(
   trainLoader: TrainLoader,
   trainDispatch: TrainDispatch,
   loadListener: ActorRef @@ LoadListener) extends Actor {

   def receive = ...
}

trait StationModule {
   def createReactiveTrainDispatch(
      loadListener: ActorRef @@ LoadListener) =
      actorSystem.actorOf(Props(wire[ReactiveTrainDispatch]))
```

```
    // actor system module dependency
    def actorSystem: ActorSystem

    // as previously
    // ...
}

// usage; statically checked ActorRef types!
val loadListener = actorSystem
      .actorOf(Props[LoadListenerActor])
      .taggedWith[LoadListener]

val reactiveTrainDispatch = modules
      .createReactiveTrainDispatch(loadListener)
```

Comments

↑ Back to top ↓ Comments

• Reply • Share ›

Michael Sandrof ➚ Adam Warski • 2 years ago

That's what I ended up doing. Not a big deal as this is just a compatibility layer. Thanks!

• Reply • Share ›

Michael Sandrof • 2 years ago

Just started using MacWire and I really like it! The compile-time wiring is a godsend compared to frameworks like Guice, which I've used extensively. I am experiencing, however, a problem when trying to use wiredInModule to access the graph. If I have exactly one module, everything works as expected. But if I have more than one, which is always the case, the compiler whines:

Error:(19, 30) the result type of an implicit conversion must be more specific than AnyRef
val wired = wiredInModule(modules)
^

Error:(19, 30) type mismatch;
found : Int
required: AnyRef
val wired = wiredInModule(modules)

Any idea what's going on? Thanks!

• Reply • Share ›

Adam Warski Mod ➚ Michael Sandrof • 2 years ago

Looks quite weird :) I'm using wiredInModule with multiple modules and it works ... any chances for a test case? (e.g. as an issue in GitHub https://github.com/adamw/macwi... Would be great!

• Reply • Share ›

Michael Sandrof ➚ Adam Warski • 2 years ago

While decomposing my code to create a simple test case, I discovered the cause of the problem. I had a module that defined a bare Int without a tag, for example:

lazy val thing = 1

This wasn't being injected, just used internally by the module to manually wire another injectable object while allowing overrides for testing/etc. There was no problem with this until I introduced wiredInModule. Simply adding a tag to it made the problem go away. I don't know if you want to consider this a bug or just a caveat, but for me, problem solved.

• Reply • Share ›

Adam Warski Mod ➚ Michael Sandrof • 2 years ago

Definitely a bug :) Should be fixed in 1.0.5 which will be in maven central in ~30 minutes.

• Reply • Share ›

Kamran Massoudi • 2 years ago

Hi Adam,

It was great meeting you @Scala.io and thanks for this great guide! I just wanted to mention a couple of minor points I figured while going through the

document:

- In the "Multiple implementations" section, the ModernShuntingModule would not bring in the PointSwitcher instance which is a dependency of LoadingModule and so the modernModules will not compile. Which is actually interesting as it shows it might not be a bad idea to define the instances that are only used in the scope of the module itself and are implementation-specific like trainCarCoupler as private lazy vals, just to make it clear what's exposed to the other modules and what's not.

- I couldn't express the dependencies via self-types (which in many cases I think of it as a better design than extending the other modules) as then MacWire couldn't find the dependencies anymore.

- As for Interceptors and Scopes I realised that the type should be explicit or otherwise it won't always compile, so as in:
lazy val pointSwitcher = logEvents(wire[PointSwitcher])
becoming:
lazy val pointSwitcher: PointSwitcher = logEvents(wire[PointSwitcher])

Cheers!

▲ ▼ • Reply • Share ›

Adam Warski Mod ➚ Kamran Massoudi • 2 years ago

Thanks for the detailed remarks!

1. is fixed - I moved the pointSwitcher wire to the super-trait. As for restricting visibility, using private members can be a good way, though it can get into the way when testing. You can do package-private when your test lives in the same package, but if you have integration tests this wouldn't work.

2. this is certainly a bug - can you report it here?
https://github.com/adamw/macwi...

3. ah yes, it's even a documented limitation (https://github.com/adamw/macwi..., but I added the explicit type ascription to the examples

▲ ▼ • Reply • Share ›

Kamran Massoudi ➚ Adam Warski • 2 years ago

Thanks for the detailed answer!
1. Indeed!
2. Done!
3. Right, I totally missed that section in the documentation.

▲ ▼ • Reply • Share ›

Tomer Ben David • 2 years ago

it's great! I can't tell you how much I've been waiting for such an overview/summary/overall for so long! thanks!

▲ ▼ • Reply • Share ›

Adam Warski Mod ➚ Tomer Ben David • 2 years ago

Thanks!

▲ ▼ • Reply • Share ›

Maatary Okouya • 2 years ago

This is a great work !!! I like what you did. I was just wondering, given that i use

quite a lost partial injection also known as assisted injection or injected factory, what is you prefered way to handle that case. I would like to introduce it in my project. Do you think macwire is stable enough at this stage ?

▲ ▼ • Reply • Share ›

Adam Warski Mod ➚ Maatary Okouya • 2 years ago

We are using MacWire internally in a couple of projects, and I would stay it is stable enough to use it safely. There shouldn't be any changes in the main API.

As for partial injection/assisted inject, what is your use case exactly? You can always define a "factory" dependency, e.g.:

```
val myServiceFactory = (p1: String, p2: Int) => wire[MyService]
```

or similar, and later declare the dependency as a function type:

```
class AnotherService(myServiceFactory: (String, Int) => MyService)
```

▲ ▼ • Reply • Share ›

Created using poole and jekyll.