Table of Contents

Metaconfig

Metaconfig is a library to read HOCON configuration into Scala case classes. Key features of Metaconfig include

- helpful error messages on common mistakes like typos or type mismatch (expected string, obtained int)
- configurable, semi-automatic derivation of decoders, with support for deprecating setting options
- cross-platform, supports JS/JVM. Native support is on the roadmap

The target use-case for metaconfig is tool maintainers who support HOCON configuration in their tool. Metaconfig is used by scalafmt to read `.scalafmt.conf` and scalafix to read `.scalafix.conf`. With metaconfig, tool maintainers should be able to safely evolve their configuration (deprecate old fields, add new fields) without breaking existing configuration files. Users should get helpful error messages when they mistype a setting name.

There are alternatives to metaconfig that you might want to give a try first

- https://github.com/circe/circe-config
- https://github.com/pureconfig/pureconfig

Getting started

```
libraryDependencies += "com.geirsson" %% "metaconfig-core" % "0.5.4+4-08b33047"

// Use https://github.com/lightbend/config to parse HOCON
libraryDependencies += "com.geirsson" %% "metaconfig-typesafe-config" % "0.5.4+4-08b33047"
```

Use this import to access the metaconfig API

```
import metaconfig._
```

All of the following code examples assume that you have `import metaconfig._` in scope.

Conf

`Conf` is a JSON-like data structure that is the foundation of metaconfig.

```
scala> val string = Conf.fromString("string")
string: metaconfig.Conf = "string"

scala> val int = Conf.fromInt(42)
int: metaconfig.Conf = 42

scala> Conf.fromList(int :: string :: Nil)
res0: metaconfig.Conf = [42, "string"]

scala> Conf.fromMap(Map("a" -> string, "b" -> int))
res1: metaconfig.Conf = {"a": "string", "b": 42}
```

Conf.parse

You need an implicit `MetaconfigParser` to convert HOCON into `Conf`. Assuming you depend on the `metaconfig-typesafe-config` module,

```
scala> import metaconfig.typesafeconfig._
import metaconfig.typesafeconfig._

scala> Conf.parseString("""
     | a.b.c = 2
     | a.d = [ 1, 2, 3 ]
     | reference = ${a}
     | """)
res2: metaconfig.Configured[metaconfig.Conf] = Ok({"a": {"d": [1, 2
, 3], "b": {"c": 2}}, "reference": {"d": [1, 2, 3], "b": {"c": 2}}}
)

scala> Conf.parseFile(new java.io.File(".scalafmt.conf"))
res3: metaconfig.Configured[metaconfig.Conf] = Ok({"align": "none",
 "project": {"git": true}, "assumeStandardLibraryStripMargin": true
})
```

Note. The example above is JVM-only. For a Scala.js alternative, depend on the `metaconfig-hocon` module and replace `metaconfig.typesafeconfig` with

```
import metaconfig.hocon._
```

ConfDecoder.instance

To convert `Conf` into higher-level data structures you need a `ConfDecoder[T]` instance. Convert a partial function from `Conf` to your target type using `ConfDecoder.instance[T]`.

```
val number2 = ConfDecoder.instance[Int] {
    case Conf.Str("2") => Configured.Ok(2)
}
```

```
scala> number2.read(Conf.fromString("2"))
res4: metaconfig.Configured[Int] = Ok(2)

scala> number2.read(Conf.fromInt(2))
res5: metaconfig.Configured[Int] =
NotOk(Type mismatch;
  found    : Number (value: 2)
  expected : int)
```

Convert a regular function from `Conf` to your target type using `ConfDecoder.instanceF[T]`.

```
case class User(name: String, age: Int)
val decoder = ConfDecoder.instanceF[User] { conf =>
  conf.get[String]("name").product(conf.get[Int]("age")).map {
      case (name, age) => User(name, age)
  }
}
```

```
scala> decoder.read(Conf.parseString("""
     | name = "Susan"
     | age = 29
     | """))
res6: metaconfig.Configured[User] = Ok(User(Susan,29))

scala> decoder.read(Conf.parseString("""
     | name = 42
     | age = "Susan"
     | """))
res7: metaconfig.Configured[User] =
NotOk(2 errors
[E0] Type mismatch;
  found    : Number (value: 42)
  expected : String
[E1] Type mismatch;
  found    : String (value: "Susan")
  expected : Number
)
```

ConfError

`ConfError` is a helper to produce readable and potentially aggregated error messages.

```
scala> ConfError.message("Not good!")
res8: metaconfig.ConfError = Not good!

scala> ConfError.exception(new IllegalArgumentException("Expected S
tring!"), stackSize = 2)
res9: metaconfig.ConfError =
java.lang.IllegalArgumentException: Expected String!
    at .<init>(<console>:19)
    at .<clinit>(<console>)

scala> ConfError.typeMismatch("Int", "String", "field")
res10: metaconfig.ConfError =
Type mismatch at 'field';
  found    : String
  expected : Int

scala> ConfError.message("Failure 1").combine(ConfError.message("Fa
ilure 2"))
res11: metaconfig.ConfError =
2 errors
[E0] Failure 1
[E1] Failure 2
```

Metaconfig uses Scalameta `Input` to represent an input source and `Position` to represent range positions in a given `Input`

```
import scala.meta.inputs._
val input = Input.VirtualFile(
  "foo.scala",
  """
    |object A {
    |  var x
    |}
  """.stripMargin
)
val i = input.value.indexOf('v')
val pos = Position.Range(input, i, i)
```

```
scala> ConfError.parseError(pos, "No var")
res12: metaconfig.ConfError =
foo.scala:3: error: No var
var x
  ^
```

Configured

`Configured[T]` is like an `Either[metaconfig.ConfError, T]` which is used througout the metaconfig API to either represent a successfully parsed/decoded value or a failure.

```
scala> Configured.ok("Hello world!")
res13: metaconfig.Configured[String] = Ok(Hello world!)

scala> Configured.ok(List(1, 2))
res14: metaconfig.Configured[List[Int]] = Ok(List(1, 2))

scala> val error = ConfError.message("Boom!")
error: metaconfig.ConfError = Boom!

scala> val configured = error.notOk
configured: metaconfig.Configured[Nothing] = NotOk(Boom!)

scala> configured.toEither
res15: Either[metaconfig.ConfError,Nothing] = Left(Boom!)
```

To skip error handling, use the nuclear `.get`

```
scala> configured.get
java.util.NoSuchElementException: Boom!
  at metaconfig.Configured.get(Configured.scala:11)
  ... 45 elided
```

```
scala> Configured.ok(42).get
res17: Int = 42
```

generic.deriveSurface

To use automatic derivation, you first need a `Surface[T]` typeclass instance

```
scala> implicit val userSurface: Surface[User] =
     |   generic.deriveSurface[User]
userSurface: metaconfig.Surface[User] = Surface(List(Field(name,None
,java.lang.String,List()), Field(age,None,Int,List())))
```

The surface is used by metaconfig to support configurable decoding such as alternative fields names. In the future, the plan is to use `Surface[T]` to automatically generate html/markdown documentation for configuration settings. For now, you can ignore `Surface[T]` and just consider it as an annoying requirement from metaconfig.

generic.deriveDecoder

Writing manual decoder by hand grows tiring quickly. This becomes especially true when you have documentation to keep up-to-date as well.

```
implicit val decoder: ConfDecoder[User] =
  generic.deriveDecoder[User](User("John", 42)).noTypos
```

```
scala> ConfDecoder[User].read(Conf.parseString("""
     | name = Susan
     | age = 34
     | """))
res18: metaconfig.Configured[User] = Ok(User(Susan,34))

scala> ConfDecoder[User].read(Conf.parseString("""
     | nam = John
     | age = 23
     | """))
res19: metaconfig.Configured[User] = NotOk(Invalid field: nam. Expe
cted one of name, age)

scala> ConfDecoder[User].read(Conf.parseString("""
     | name = John
     | age = Old
     | """))
res20: metaconfig.Configured[User] =
NotOk(Type mismatch;
  found    : String (value: "Old")
  expected : Number)
```

Sometimes automatic derivation fails, for example if your class contains fields that have no `ConfDecoder` instance

```
scala> import java.io.File
import java.io.File

scala> case class Funky(file: File)
defined class Funky

scala> implicit val surface = generic.deriveSurface[Funky]
surface: metaconfig.Surface[Funky] = Surface(List(Field(file,None,j
ava.io.File,List())))
```

This will fail wiith a fail cryptic compile error

```
scala> implicit val decoder = generic.deriveDecoder[Funky](Funky(new
 File("")))
<console>:27: error: could not find implicit value for parameter ev
: metaconfig.ConfDecoder[java.io.File]
       implicit val decoder = generic.deriveDecoder[Funky](Funky(new
 File("")))
                                                           ^
```

Observer that the error message is complaining about a missing `metaconfig.ConfDecoder[java.io.File]` implicit.

DeprecatedName

As your configuration evolves, you may want to rename some settings but you have existing users who are using the old name. Use the `@DeprecatedName` annotation to continue supporting the old name even if you go ahead with the rename.

```
case class EvolvingConfig(
    @DeprecatedName("goodName", "Use isGoodName instead", "1.0")
    isGoodName: Boolean
)
implicit val surface = generic.deriveSurface[EvolvingConfig]
implicit val decoder = generic.deriveDecoder[EvolvingConfig](Evolvi
ngConfig(true)).noTypos
```

```
scala> decoder.read(Conf.Obj("goodName" -> Conf.fromBoolean(false))
)
res21: metaconfig.Configured[EvolvingConfig] = Ok(EvolvingConfig(fa
lse))

scala> decoder.read(Conf.Obj("isGoodName" -> Conf.fromBoolean(false
)))
res22: metaconfig.Configured[EvolvingConfig] = Ok(EvolvingConfig(fa
lse))

scala> decoder.read(Conf.Obj("gooodName" -> Conf.fromBoolean(false)
))
res23: metaconfig.Configured[EvolvingConfig] = NotOk(Invalid field:
 gooodName. Expected one of isGoodName)
```