Code Review Gems

Richard Dallaway, @d6y

Agenda

The External Review

Three Code Examples

Problem → Solution → Key Points

Limits of Review

External Code Review

…can take many forms

…can be DIY

Focus on Two Questions

Are we producing good Scala?

…making the most of what Scala offers?

Is this code maintainable?

— Example 1 —

Option is not always the right option

```scala
class Settings() {
  val dbUser: Option[String] = read("db.user")
}

def doTheWork(conf: Settings): Unit = {
```

```scala
}

doTheWork(new Settings())
```

```
class Settings() {
  val dbUser: Option[String] = read("db.user")
}

def doTheWork(conf: Settings): Unit = {

  val user = try {
    conf.dbUser.get
  } catch {
    case nse: NoSuchElementException =>
     throw new IllegalArgumentException(“Missing setting")
  }

}

doTheWork(new Settings())
```

```
class Settings() {
  val dbUser: Option[String] = read("db.user")
}

def doTheWork(conf: Settings): Unit = {

  val user = try {
    conf.dbUser.get
  } catch {
    case nse: NoSuchElementException =>
     throw new IllegalArgumentException(“Missing setting")
  }

}

doTheWork(new Settings())
```

⇒ Unit

```
import com.typesafe.config.{Config,ConfigFactory}
```

```
def doTheWork(conf: Settings): Unit = {
  println(s"Using ${conf.dbUser}")
}
```

```
doWork(settings)
```

```scala
import com.typesafe.config.{Config,ConfigFactory}

class Settings(conf: Config) {
  val dbUser: String = conf.getString("db.user")
}

val settings = new Settings(ConfigFactory.load())

def doTheWork(conf: Settings): Unit = {
  println(s"Using ${conf.dbUser}")
}
```

```scala
doWork(settings)
```

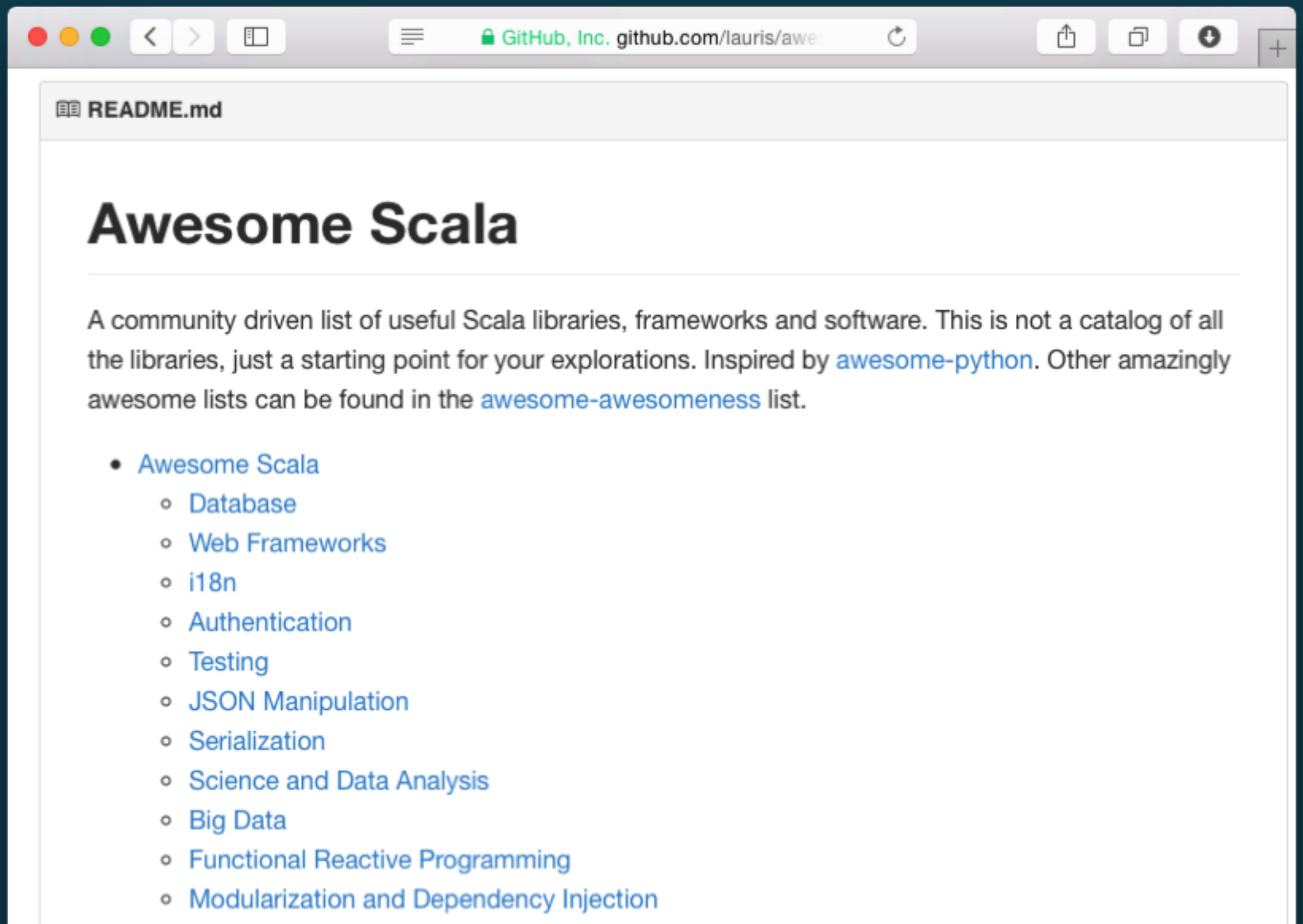
Observations

Watch out for ugly code

Make time to discover what's already out there

…and adapt if it doesn't quite fit.

Resources


GitHub, Inc. github.com/lauris/awe
README.md
Awesome Scala
A community driven list of useful Scala libraries, frameworks and software. This is not a catalog of all the libraries, just a starting point for your explorations. Inspired by awesome-python. Other amazingly awesome lists can be found in the awesome-awesomeness list.
Awesome Scala
Database
Web Frameworks
i18n
Authentication
Testing
JSON Manipulation
Serialization
Science and Data Analysis
Big Data
Functional Reactive Programming
Modularization and Dependency Injection

Observations

Watch out for ugly code

Make time to discover what's already out there

…and adapt if it doesn't quite fit.

Unit is the end of the line

Good Scala?

Good Scala?

Something the team arrives at

Based on the broader community

Changes over time

— Example 2 —

Stringly typed

```
object PaymentStatus {
  val NOT_STARTED = "n/a"
  val PENDING     = "pending"
  val COMPLETE    = "complete"
  val REJECTED    = "rejected"
}
```

```
object PaymentStatus {
  val NOT_STARTED = "n/a"
  val PENDING     = "pending"
  val COMPLETE    = "complete"
  val REJECTED    = "rejected"
}

def isFinished(status: String): Boolean =
  status == COMPLETE || status == REJECTED

// Run-time match error:
"uh-oh" match {
  case NOT_STARTED | PENDING  => false
  case COMPLETE    | REJECTED => true
}
```

“Enums fill a different niche: essentially as efficient as integer constants but safer and more convenient to define and to use.”

– Martin

```
object StatusEnum extends Enumeration {
  type Status     = Value

  val NotStarted = Value("n/a")
  val Pending    = Value("pending")
  val Complete   = Value("complete")
  val Rejected   = Value("rejected")
}
```

```
object StatusEnum extends Enumeration {
  type Status     = Value

  val NotStarted = Value("n/a")
  val Pending    = Value("pending")
  val Complete   = Value("complete")
  val Rejected   = Value("rejected")
}

def isFinished(status: Status): Boolean =
  status == Complete || status == Rejected
```

```
def isInProgress(status: Status): Boolean =
  status match {
    case Pending                 => true
    case Complete | Rejected => false
  }
```

Enumerations

😀	Automatic integer identifier
😀	Automatic string name
😀	Values are ordered
😀	Automatic iterator
😀	Light (not one class per value)
😦	Same type after erasure
😦	No non-exhaustive match warning

```scala
object Status {
  sealed abstract class Value(val name: String)

  case object NotStarted extends Value("n/a")
  case object Pending    extends Value("pending")
  case object Complete   extends Value("complete")
  case object Rejected   extends Value("rejected")

  val values = Seq(NotStarted, Pending, Complete, Rejected)
}
```

```scala
object Status {
  sealed abstract class Value(val name: String)

  case object NotStarted extends Value("n/a")
  case object Pending    extends Value("pending")
  case object Complete   extends Value("complete")
  case object Rejected   extends Value("rejected")

  val values = Seq(NotStarted, Pending, Complete, Rejected)
}
```

```scala
def isFinished(status: Status.Value): Boolean =
  status == Complete || status == Rejected

// Compile error:  match may not be exhaustive
// It would fail on the following input: NotStarted
def isInProgress(status: Status.Value): Boolean =
  status match {
    case Pending                => true
    case Complete | Rejected => false
  }
```

But I have `VARCHAR`
not types...

```scala
case class Payment(status: Value, id: Long=0L)

lazy val payments = TableQuery[Payments]

def pending(implicit s: Session) =
  payments.filter(_.status === (Pending : Value)).list
```

```
implicit val statusColumnType =
  MappedColumnType.base[Value, String](
    _.name,
    s => Status.values.find(_.name == s) getOrElse NotStarted
  )

case class Payment(status: Value, id: Long=0L)

lazy val payments = TableQuery[Payments]

def pending(implicit s: Session) =
  payments.filter(_.status === (Pending : Value)).list
```

Observations

Let the type system help you

Get data into types as soon as you can

"Scala Enumerations"
https://underscoreconsulting.com/blog/posts/2014/09/03/enumerations.html

— Example 3 —

Tests as docs

```
class ProblemSpec extends Specification {
  "The processor" should {
    "detect a name change" in {
```

```
    }
}
```

```
class ProblemSpec extends Specification {
  "The processor" should {
    "detect a name change" in {

      // If this was in the database...
      val ap = Applicant(number="0000000002",
                 surname="Smith", forenames=None)

       // ...and we received this change...
       val input = CsvRow(
         year       = "2014",
         code       = Code.Important,
         personId   = "000000001",
         number     = "0000000002",
         choice     = Choice.Third,
         surname    = Some("Smith"),
         forenames = Some("Alice")
       )

       Processor.diff(ap, input) must beSome(NameChange())
    }
  }
}
```

```
class SolutionSpec extends Specification {
  "The processor" should {
    "detect a name change" in {

      // If this was in the database...
      val app = TestApplicant() withNoForenames

      // ...and we received this change...
      val input =
        TestCsvRow() matching(app) withForenames "Alice"

      // ...the action must be a name change:
      Processor.diff(app.create, input.create)
        must beSome(NameChange())

    }
  }
}
```

Builder with fixed data...

```
case class TestApplicant(
  number:    String="0000000002",
  surname:   String="Smith",
  forenames: Option[String]=Some("Colin")) {

  def withNoForenames: TestApplicant =
    copy(forenames=None)

  def create: Applicant =
    Applicant(number, surname, forenames)
}
```

Alternative with random test data...

```scala
import org.scalacheck._, Arbitrary._, Gen._

object Builders {

  lazy val applicantGenerator: Gen[Applicant] =
    for {
      number    <- tenRandomDigits
      surname   <- arbitrary[String]
      forenames <- arbitrary[Option[String]]
    } yield Applicant(number, surname, forenames)

  val tenRandomDigits =
    listOfN(10, Gen.numChar).map(_.mkString)



}
```

```
import org.scalacheck._, Arbitrary._, Gen._

object Builders {

  lazy val applicantGenerator: Gen[Applicant] =
    for {
      number    <- tenRandomDigits
      surname   <- arbitrary[String]
      forenames <- arbitrary[Option[String]]
    } yield Applicant(number, surname, forenames)
```

```
val tenRandomDigits =
Applicant(8698360245,,None)
```

```
}
```

```scala
import org.scalacheck._, Arbitrary._, Gen._

object Builders {

  lazy val applicantGenerator: Gen[Applicant] =
    for {
      number    <- tenRandomDigits
      surname   <- arbitrary[String]
      forenames <- arbitrary[Option[String]]
    } yield Applicant(number, surname, forenames)

  val tenRandomDigits =
    listOfN(10, Gen.numChar).map(_.mkString)

}
```

```
Applicant(8698360245,,None)

Applicant(5652792232,㊉襟텝稙勺涿侹릅¡僟悆♻蝓▩|||躪憴辭
珺苁✋鄯륃椙巄휊⦶↓맨¦¢璒鐦ʋ偄ᔕ藜릴铜,
Some(퓸杓c艂顧))
```

```scala
import org.scalacheck._, Arbitrary._, Gen._

object Builders {

  lazy val applicantGenerator: Gen[Applicant] =
    for {
      number    <- tenRandomDigits
      surname   <- arbitrary[String]
      forenames <- arbitrary[Option[String]]
    } yield Applicant(number, surname, forenames)

  val tenRandomDigits =
    listOfN(10, Gen.numChar).map(_.mkString)



}
```

```scala
import org.scalacheck._, Arbitrary._, Gen._

object Builders {

  lazy val applicantGenerator: Gen[Applicant] =
    for {
      number    <- tenRandomDigits
      surname   <- arbitrary[String]
      forenames <- arbitrary[Option[String]]
    } yield Applicant(number, surname, forenames)

  val tenRandomDigits =
    listOfN(10, Gen.numChar).map(_.mkString)

  implicit class ApplicantOps(app: Applicant) {
    def withNoForenames: Applicant =
      app.copy(forenames=None)
  }



}
```

```scala
import org.scalacheck._, Arbitrary._, Gen._

object Builders {

  lazy val applicantGenerator: Gen[Applicant] =
    for {
      number    <- tenRandomDigits
      surname   <- arbitrary[String]
      forenames <- arbitrary[Option[String]]
    } yield Applicant(number, surname, forenames)

  val tenRandomDigits =
    listOfN(10, Gen.numChar).map(_.mkString)

  implicit class ApplicantOps(app: Applicant) {
    def withNoForenames: Applicant =
      app.copy(forenames=None)
  }

  implicit val applicant: Arbitrary[Applicant] =
    Arbitrary(applicantGenerator)
}
```

```
class RandSpec extends Specification with ScalaCheck {

  import Builders._

  "The processor" should {
    "detect a name change" in prop {
      (applicant: Applicant, input: CsvRow) =>

      Processor.diff(
        applicant withNoForenames,
        input matching(applicant) withForenames “Alice")
            must beSome(NameChange())
    }
  }
}
```

Observations

You can still use what you know

Make use of an expressive syntax

…and neat libraries

Don’t settle for crappy tests

Challenges

Code	Maintenance
Learning the language	Explaining the problem
Framework experience	Explaining the thinking
Exploring the ecosystem	Readable tests

“You cannot inspect quality into a product.”

–Harold S. Dodge

Summary

Learn about the wider Scala ecosystem
—get out there!

Don’t leave it until “the review”
—think about code as if you were external

Thanks!

Richard Dallaway, @d6y

https://github.com/d6y/gems