Circe

```
val circeVersion = "0.13.0"

libraryDependencies ++= Seq(
  "io.circe" %% "circe-core",
  "io.circe" %% "circe-generic",
  "io.circe" %% "circe-parser"
).map(_ % circeVersion)
```

- fork Argonaut'a
- core - зависит от cats
- parser - зависит от Jawn
- generic - зависит от Shapeless

sealed abstract class Json

- private[circe] final case object JNull
- private[circe] final case class JBoolean(value: Boolean)
- private[circe] final case class JNumber(value: JsonNumber)
- private[circe] final case class JString(value: String)
- private[circe] final case class JArray(value: Vector[Json])
- private[circe] final case class JObject(value: JsonObject)

Парсинг JSON (пример)

```
import io.circe._, io.circe.parser._
val json: String =
  """
  {
    "id": "c730433b-082c-4984-9d66-855c243266f0",
    "name": "Foo",
    "counts": [1, 2, 3],
    "values": {
      "bar": true,
      "baz": 100.001,
      "qux": ["a", "b"]
    }
  }"""

val doc: Json = parse(json).getOrElse(Json.Null)
```

Запись JSON

```
import io.circe._
val json = Json.obj(
  ("id", Json.fromString("c730433b-082c-4984-9d66-855c243266f0")),
  ("name", Json.fromString("Foo")),
  ("counts", Json.fromValues(Seq[Json](Json.fromInt(1), Json.fromInt(2), Json.fromInt(3)))),
  ("values",
    Json.obj(("bar", Json.fromBoolean(true)),
      ("baz", Json.fromDouble(100.01).getOrElse(0)),
      ("qux", Json.fromValues(Seq[Json](Json.fromString("a"), Json.fromString("b"))))))
)
json.noSpaces
json.spaces2SortKeys
```

Парсинг JSON (ошибки)

```
def parse(input: String): Either[ParsingFailure, Json]
```

```
final case class ParsingFailure(message: String, underlying: Throwable) extends Error {
  final override def getMessage: String = message
}
```

```
import  io.circe.parser._
val json: String =
  """
      {
        "id": "c730433b-082c-4984-9d66-855c243266f0",
        "name": "Foo",
        "counts": [1, 2, 3]
        "values": {
          "bar": true,
          "baz": 100.001,
          "qux": ["a", "b"]
        }
      }"""
val doc = parse(json)
```

```
Left(io.circe.ParsingFailure: expected } or , got '"value...' (line 6, column 13))
```

Cursor чтение

```
import io.circe._, io.circe.parser._
val json: String =
  """
      {
        "id": "c730433b-082c-4984-9d66-855c243266f0",
        "name": "Foo",
        "counts": [1, 2, 3],
        "values": {
          "bar": true,
          "baz": 100.001,
          "qux": ["a", "b"]
        }
      }"""

val doc: Json = parse(json).getOrElse(Json.Null)

val cursor: HCursor = doc.hcursor

val baz: Decoder.Result[Double] =
  cursor.downField("values").downField("baz").as[Double]

val secondQux: Decoder.Result[String] =
  cursor.downField("values").downField("qux").downN(1).as[String]

val taadaa: Decoder.Result[String] =
  cursor.downField("values").downField("bar").downField("oops").as[String]
```

```
Right(100.001)
Right(b)
Left(DecodingFailure(Attempt to decode value on failed cursor, List(DownField(oops),
DownField(bar), DownField(values))))
```

Cursor изменение

```
import io.circe._, io.circe.parser._
val json: String =
  """
      {
        "id": "c730433b-082c-4984-9d66-855c243266f0",
        "name": "Foo",
        "counts": [1, 2, 3],
        "values": {
          "bar": true,
          "baz": 100.001,
          "qux": ["a", "b"]
        }
      }"""

val doc: Json = parse(json).getOrElse(Json.Null)

val cursor: HCursor = doc.hcursor

val reversedNameCursor: ACursor =
  cursor.downField("name").withFocus(_.mapString(_.reverse))

val reversedName: Option[Json] = reversedNameCursor.top
```

Encoder

```scala
trait Encoder[A] extends Serializable { self =>

  /**
   * Convert a value to JSON.
   */
  def apply(a: A): Json

  /**
   * Create a new [[Encoder]] by applying a function to a value of type `B` before encoding as an
   * `A`.
   */
  final def contramap[B](f: B => A): Encoder[B] = new Encoder[B] {
    final def apply(a: B): Json = self(f(a))
  }

  /**
   * Create a new [[Encoder]] by applying a function to the output of this one.
   */
  final def mapJson(f: Json => Json): Encoder[A] = new Encoder[A] {
    final def apply(a: A): Json = f(self(a))
  }
}
```

Encoder instances

- Инстансы для всех стандартных типов Int, String, Long…
- Инстансы для Option[A], List[A], … при наличии инстанса для A

```
import io.circe._

Encoder[String](encodeString)
Encoder[List[String]](encodeList(encodeString))
```

```
/**
 * Return an instance for a given type `A`.
 *
 * @group Utilities
 */
final def apply[A](implicit instance: Encoder[A]): Encoder[A] = instance
```

Encoders tuning

```
val reverseStringEncoder: Encoder[String] = Encoder[String].mapJson(_.mapString(_.reverse))

val urlEncoder: Encoder[URL] = Encoder[String].contramap[URL](d => d.toString)
```

Encoding examples

```
import io.circe.syntax._
val intsJson = List(1, 2, 3).asJson
// intsJson: io.circe.Json = JArray(
// Vector(JNumber(JsonLong(1L)),JNumber(JsonLong(2L)), JNumber(JsonLong(3L)))
// )
```

```
val intsJson = EncoderOps(List(1, 2, 3)).asJson(encodeList(encodeInt))
```

```
/**
 * This package provides syntax via enrichment classes.
 */
package object syntax {

  implicit final class EncoderOps[A](private val value: A) extends AnyVal {

    final def asJson(implicit encoder: Encoder[A]): Json = encoder(value)
  }

}
```

Custom Encoders

```
import io.circe.Encoder

class Thing(val foo: String, val bar: Int)

implicit val encodeFoo: Encoder[Thing] = new Encoder[Thing] {
  final def apply(a: Thing): Json  =  Json.obj(
    ("foo", Json.fromString(a.foo)),
    ("bar", Json.fromInt(a.bar))
  )
}
```

forProductN codecs

```
import io.circe.Encoder

class Thing(val foo: String, val bar: Int)

implicit val encodeFoo: Encoder[Thing] =
  Encoder.forProduct2("foo", "bar")(f => (f.foo, f.bar))
```

Encoder for sum types

```
import io.circe.Encoder,io.circe.syntax._

sealed trait Event

case class Foo(i: Int) extends Event
case class Bar(s: String) extends Event

implicit val encodeFoo: Encoder[Foo] = Encoder.forProduct1("i")(f => f.i)
implicit val encodeBar: Encoder[Bar] = Encoder.forProduct1("s")(f => f.s)

implicit val encodeEvent: Encoder[Event] = Encoder.instance {
  case foo @ Foo(_) => foo.asJson
  case bar @ Bar(_) => bar.asJson
}
println((Bar("Hello"):Event).asJson)
```

Decoder

```scala
/**
 * A type class that provides a way to produce a value of type `A` from a [[Json]] value.
 */
trait Decoder[A] extends Serializable { self =>

  /**
   * Decode the given [[HCursor]].
   */
  def apply(c: HCursor): Decoder.Result[A]

  def decodeAccumulating(c: HCursor): Decoder.AccumulatingResult[A]
  …
}
```

Decoder instances

- Инстансы для всех стандартных типов Int, String, Long…
- Инстансы для Option[A], List[A], … при наличии инстанса для A

Decoder tuning

```
import java.net.URL, io.circe.Decoder

implicit val decodeInstant: Decoder[URL] = Decoder.decodeString.emapTry { str =>
  Try(new URL(str))
}
```

Decoding examples

```
import io.circe.syntax._, io.circe.parser._

val intsJson = List(1, 2, 3).asJson

val decodedList = intsJson.as[List[Int]]

val decodeResult: Either[Error, List[Int]] = decode[List[Int]]("[1, 2, 3]")
```

Custom Decoder (monadic)

```scala
import io.circe.{ Decoder,  HCursor }

class Thing(val foo: String, val bar: Int)

implicit val decodeFoo: Decoder[Thing] = new Decoder[Thing] {
  final def apply(c: HCursor): Decoder.Result[Thing] =
    for {
      foo <- c.downField("foo").as[String]
      bar <- c.downField("bar").as[Int]
    } yield {
      new Thing(foo, bar)
    }
}
```

Custom Decoder (applicative)

```
import io.circe.{Decoder}
import cats.syntax.apply._

case class Thing(val foo: String, val bar: Int)
implicit val decodeFoo: Decoder[Thing] = (
  Decoder.instance(_.downField("foo").as[String]),
  Decoder.instance(_.downField("bar").as[Int])
  ).mapN(Thing.apply)

val res: ValidatedNel[Error, Thing] =
decodeAccumulating[Thing]("{\"foo\":55,\"bar\":\"dfwdfw\"}")
```

```
Invalid(NonEmptyList(DecodingFailure(String, List(DownField(foo))),
DecodingFailure(Int, List(DownField(bar)))))
```

Validation

```
implicit val decodeFoo: Decoder[Thing] = (
  Decoder.instance(_.downField("foo").as[String])
                   .validate(_.downField("foo")
                              .as[String].exists(_.nonEmpty),"foo must be nonempty"),
  Decoder.instance(_.downField("bar").as[Int])
  ).mapN(Thing.apply)
```

Decoder Sum types

```
import io.circe.Decoder
import cats.syntax.functor._

sealed trait Event
case class Foo(i: Int) extends Event
case class Bar(s: String) extends Event

implicit val decodeEvent: Decoder[Event] =
  List[Decoder[Event]](
    Decoder[Foo].widen,
    Decoder[Bar].widen,
  ).reduceLeft(_ or _)
```

Semi-automatic/automatic Derivation

Generic programming

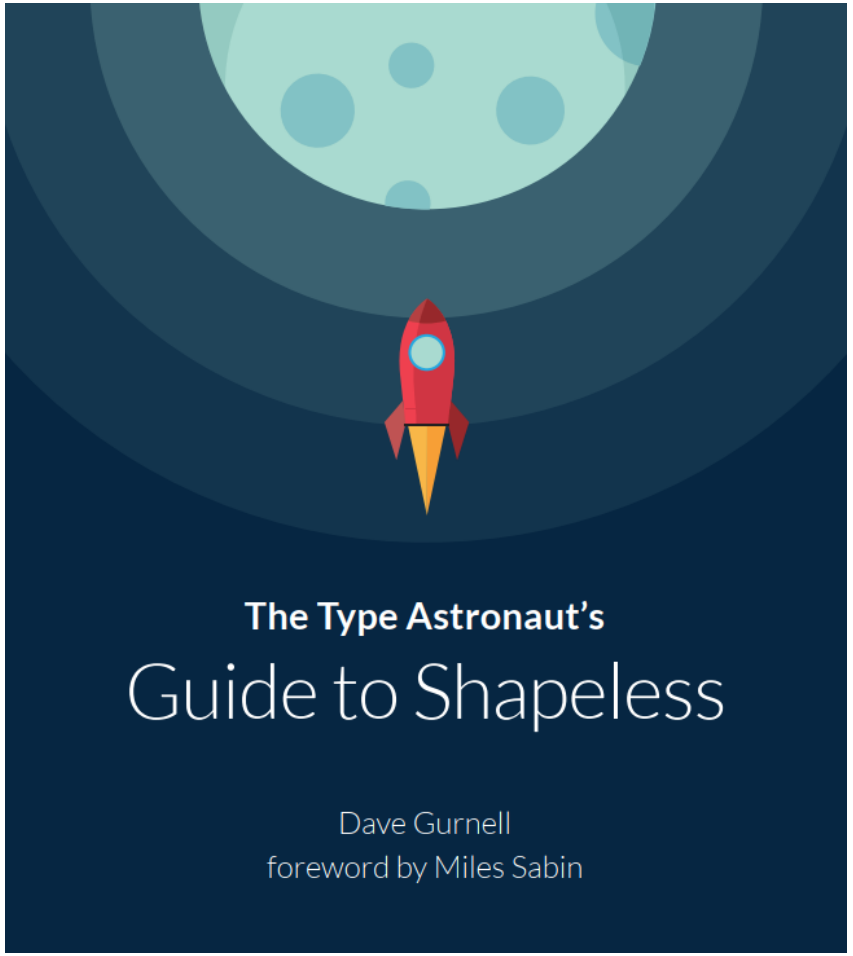
```
case class Employee(name: String, number: Int, manager: Boolean)

case class IceCream(name: String, numCherries: Int, inCone: Boolean)

def employeeCsv(e: Employee): String =
  s"${e.name}; ${e.number.toString}, ${e.manager.toString}“

def iceCreamCsv(c: IceCream): String =
    s"${c.name}, ${c.numCherries.toString}, ${c.inCone.toString}"
```

Shapeless


The Type Astronaut's
Guide to Shapeless
Dave Gurnell
foreword by Miles Sabin

HList

```
import shapeless._

val genericEmployee = Generic[Employee].to(Employee("Dave", 123, false))
// genericEmployee: String :: Int :: Boolean :: shapeless.HNil = Dave:: 123 :: false :: HNil

val genericIceCream = Generic[IceCream].to(IceCream("Sundae", 1, false))
// genericIceCream: String :: Int :: Boolean :: shapeless.HNil = Sundae :: 1 :: false :: HNil
```

Type Class Derivation (Scala 3)

Type class derivation is a way to automatically generate given instances for type classes which satisfy some simple conditions.

```
enum Tree[T] derives Eq, Ordering, Show {
  case Branch[T](left: Tree[T], right: Tree[T])
  case Leaf[T](elem: T)
}
```

```
given [T: Eq]       as Eq[Tree[T]]       = Eq.derived
given [T: Ordering] as Ordering[Tree] = Ordering.derived
given [T: Show]     as Show[Tree]     = Show.derived
```

Semi-automatic derivation

```
import io.circe._, io.circe.generic.semiauto._

case class Foo(a: Int, b: String, c: Boolean)

implicit val fooDecoder: Decoder[Foo] = deriveDecoder
implicit val fooEncoder: Encoder[Foo] = deriveEncoder
```

Automatic derivation

```
import io.circe.generic.auto._, io.circe.syntax._

case class Person(name: String)
case class Greeting(salutation: String, person: Person, exclamationMarks: Int)

Greeting("Hey", Person("Chris"), 3).asJson
```