ZIO Schema

A toolkit for distributed functional programming

TL;DR: What is ZIO Schema

```
trait Schema[A]
```

- Describes a Scala data type.
- Turns a compile-time construct (Type) into a runtime value.

TL; DR: What problems does ZIO Schema solve?

- Metaprogramming **without** macros, reflection or complicated implicit derivations
  - Serialization codecs
  - Ordering
  - Default values
- Automate ETL tasks
  - Diffing/patching
  - Migration
- Computations as data
  - Optics

Anatomy of a Schema: Primitives

Predefined schemas for "primitive types" (JVM primitives, String, UUID, java.time.*, ...)

```
case class Primitive[A](standardType: StandardType[A]) extends Schema[A]

sealed trait StandardType[A]
object StandardType {
    implicit object UnitType extends StandardType[Unit]
    implicit object IntType extends StandardType[Int]
    // and so on ...
}
```

Anatomy of a Schema: Transforms

Define new schemas as an isomorphism of an existing Schema

```
case class Transform[A, B](
    codec: Schema[A],
    f: A => Either[String, B],
    g: B => Either[String, A]
) extends Schema[B]

case class Age(i: Int) extends AnyVal

val naturalNumberSchema: Schema[Age] =
    Schema.primitive(StandardType.IntType).tranform(
        (i: Int) => if (i >= 0 && i <= 120) Right(Age(i)) else Left("Age must be between 1 and 120"),
        (age: Age) => Right(age.i)
    )
```

Anatomy of a Schema: Collections

```
sealed trait Collection[Col, Elem] extends Schema[Col]
```

Sequence: Anything isomorphic to a list

```
case class Sequence[Col, Elem](
    schemaA: Schema[Elem],
    fromChunk: Chunk[Elem] => Col,
    toChunk: Col => Chunk[Elem]
) extends Collection[Col, Elem]
```

MapSchema: Specialized encoding for Maps

```
final case class MapSchema[K, V](
    ks: Schema[K],
    vs: Schema[V]
) extends Collection[Map[K, V], (K, V)]
```

Anatomy of a Schema: Scala standard library types

All the standard types you know and love

```
case class Optional[A](codec: Schema[A]) extends Schema[Option[A]]

case class Tuple[A, B](left: Schema[A], right: Schema[B]) extends Schema[(A, B)]

case class EitherSchema[A, B](left: Schema[A], right: Schema[B]) extends Schema[Either[A, B]]
```

Anatomy of a Schema: Product types

```
case class Field[A](label: String, schema: Schema[A])

sealed trait Record[R] extends Schema[R] {
    def structure: Chunk[Field[_]]
}

case class CaseClass1[A, Z](
    field: Field[A],
    construct: A => Z,
    extractField: Z => A) extends Record[Z]

case class CaseClass2[A1, A2, Z](
    field1: Field[A1],
    field2: Field[A2],
    construct: (A1, A2) => Z,
    extractField1: Z => A1,
    extractField2: Z => A2) extends Record[Z]
// up to CaseClass22

case class GenericRecord(fieldSet: FieldSet) extends Record[ListMap[String, _]]
```

Anatomy of a Schema: Sum types

```
sealed trait Enum[A] extends Schema[A] {
    def structure: ListMap[String, Schema[_]]
}

case class Case[A, Z](
    id: String,
    codec: Schema[A],
    unsafeDeconstruct: Z => A) {
    def deconstruct(z: Z): Option[A]
}

case class Enum1[A <: Z, Z](case1: Case[A, Z]) extends Enum[Z]

case class Enum2[A1 <: Z, A2 <: Z, Z](case1: Case[A1, Z], case2: Case[A2, Z]) extends Enum[Z]

// Up to Enum22

// Generic encoding
case class EnumN[Z, C <: CaseSet.Aux[Z]](caseSet: C) extends Enum[Z]
```

Example: case class schema

```
case class User(id: Int, name: String, age: Int)

// The hard way :(
val userSchema: Schema[User] = Schema.CaseClass3[Int,String,Int,User](
    Schema.Field[Int]("id", Schema.primitive(StandardType.IntType)),
    Schema.Field[String]("name", Schema.primitive(StandardType.StringType)),
    Schema.Field[Int]("age", Schema.primitive(StandardType.IntType)),
    (id: Int, name: String, age: Int) => User(id,name,age),
    (user: User) => user.id,
    (user: User) => user.name,
    (user: User) => user.age
)

// The easy way :)
val userSchema: Schema[User] = DeriveSchema.gen[User]
```

What can we do with a Schema: Serialization codecs

```
trait Codec {
  // Streaming variants
  def encoder[A](schema: Schema[A]): ZTransducer[Any, Nothing, A, Byte]
  def decoder[A](schema: Schema[A]): ZTransducer[Any, String, Byte, A]

  // Non-streaming variants
  def encode[A](schema: Schema[A]): A => Chunk[Byte]
  def decode[A](schema: Schema[A]): Chunk[Byte] => Either[String, A]
}
```

Codecs for JSON and Protocol Buffers provided out of the box

What can we do with a Schema: Ordering and default values

```
sealed trait Schema[A] {

    def defaultValue: Either[String, A]

    def ordering: Ordering[A]

}
```

What can we do with a Schema: Diffing/ Patching

Out of the box diffing. Diffs are fully serializable and can be applied as a patch

```
sealed trait Schema[A] {

  def diff(thisValue: A, thatValue: A, differ: Option[Differ[A]] = None): Diff = differ match {
        case Some(differ) => differ(thisValue, thatValue)
        case None         => Differ.fromSchema(self)(thisValue, thatValue)
    }

  def patch(oldValue: A, diff: Diff): Either[String, A] = diff.patch(oldValue)(self)

}
```

What can we do with a Schema: Automatic migrations

Derive a migration between comparable data types

```
sealed trait Schema[A] {

    def migrate[B](newSchema: Schema[B]): Either[String, A => Either[String, B]]

    def coerce[B](newSchema: Schema[B]): Either[String, Schema[B]] =
        for {
          f <- self.migrate(newSchema)
          g <- newSchema.migrate(self)
        } yield self.transformOrFail(f, g)

}
```

What does any of this have to do with distributed systems?

Move computations to data instead of moving data to computations. All the rest is commentary.

– Not Maimonides

Schema Serialization

```
sealed trait Schema[A] {
    def serializable: Schema[Schema[A]]
}
```

The Schema itself has a Schema, so we can treat the structure (aka the type information) as pure data!

Schema Serialization: Generic encoding of Schema as an Abstract Syntax Tree

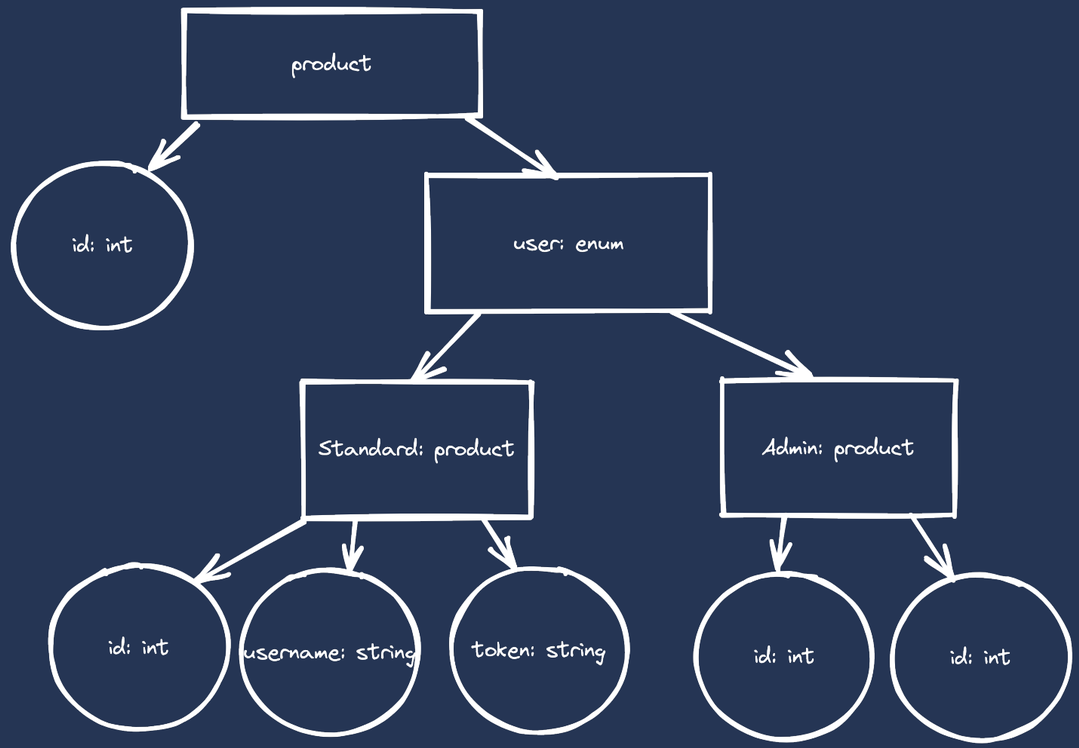
```
sealed trait User

object User {

  case class Standard(
    id: Int,
    username: String,
    token: String) extends User

  case class Admin(
    id: Int,
    level: Int) extends User
}

case class Session(id: String, user: User)
```



Schema Serialization: Automatic migration as AST diff


```
case class UserV1(id: Int, name: String, age: Int)
```

```
case class UserV2(id: Int, name: String)
```

DynamicValue: A dynamic representation of Scala data types

```
trait DynamicValue {
    def toTypedValue[A](schema: Schema[A]): Either[String, A]
}

object DynamicValue {
    def fromSchemaAndValue[A](schema: Schema[A], value: A): DynamicValue

      final case class Record(values: ListMap[String, DynamicValue]) extends DynamicValue
      final case class Enumeration(value: (String, DynamicValue))    extends DynamicValue

      final case class Sequence(values: Chunk[DynamicValue]) extends DynamicValue

      final case class Dictionary[K, V](entries: Chunk[(DynamicValue, DynamicValue)]) extends DynamicValue

      sealed case class Primitive[A](value: A, standardType: StandardType[A]) extends DynamicValue
    // Some other specializations....
}
```

Reified Optics: Encoding computation as data

```
trait AccessorBuilder {
  type Lens[S, A]
  type Prism[S, A]
  type Traversal[S, A]

  def makeLens[S, A](
    product: Schema.Record[S],
    term: Schema.Field[A]): Lens[S, A]

  def makePrism[S, A](
    sum: Schema.Enum[S],
    term: Schema.Case[A, S]): Prism[S, A]

  def makeTraversal[S, A](
    collection: Schema.Collection[S, A],
    element: Schema[A]): Traversal[S, A]
}
```

```
sealed trait Schema[A] {
  self =>

  type Accessors[Lens[_, _], Prism[_, _], Traversal[_, _]]

  def makeAccessors(
        b: AccessorBuilder
    ): Accessors[b.Lens, b.Prism, b.Traversal]

}
```

Reified Optics: In practice

```
case class CaseClass2[A1, A2, Z](
  field1: Field[A1],
  field2: Field[A2],
  construct: (A1, A2) => Z,
  extractField1: Z => A1,
  extractField2: Z => A2) extends Record[Z] { self =>

  type Accessors[Lens[_, _], Prism[_, _], Traversal[_, _]] =
      (Lens[Z, A1], Lens[Z, A2])

  override def makeAccessors(
  b: AccessorBuilder): (b.Lens[Z, A1], b.Lens[Z, A2]) =
      (b.makeLens(self, field1), b.makeLens(self, field2))
}

case class Enum2[A1 <: Z, A2 <: Z, Z](
  case1: Case[A1, Z],
  case2: Case[A2, Z]) extends Enum[Z] { self =>
  override type Accessors[Lens[_, _], Prism[_, _], Traversal[_, _]] =
      (Prism[Z, A1], Prism[Z, A2])

  override def makeAccessors(
      b: AccessorBuilder): (b.Prism[Z, A1], b.Prism[Z, A2]) =
    (b.makePrism(self, case1), b.makePrism(self, case2))
}
```

Reified Optics: In Practice

```
object MyAccessorBuilder extends AccessorBuilder {
    // Some optics encoding here :)
}

case class User(id: Int, name: String, age: Int)

object User {

    implicit val schema = DeriveSchema.gen[User]

    val (id, name, age) = schema.makeAccessors(MyAccessorBuilder)
}
```

Reified Optics: Encoding something useful!

No!

```
trait Cache[K,V] {
  def get(key: K): Option[V]
  def find(f: V => Boolean): List[V]
}
```

Yes!

```
trait Query[V,A] {
  def apply(value: V): Boolean
}

trait Cache[K,V] {
  def get(key: K): Option[V]
  def find[S](query: Query[V,S]): List[V]
}
```

Reified Optics: Lens encoding

```
case class Selector[V, A](whole: Schema[V], path: NonEmptyList[String], part: Schema[A]) {

  def apply(value: V): Either[String,A] = {
      @tailrec
      def go(rec: DynamicValue.Record, path: NonEmptyList[String]): Either[String, A] =
        path.peelNonEmpty match {
          case (next, _) if !rec.values.contains(next) => Left(s"Field $next does not exist")
          case (next, None)                            => fieldSchema.fromDynamic(rec.values(next))
          case (next, Some(rest)) =>
            rec.values(next) match {
              case rec0: DynamicValue.Record => go(rec0, rest)
              case _                         => Left(s"Field $next is not a record type")
            }
        }

      whole.toDynamic(value) match {
        case rec @ DynamicValue.Record(_) => go(rec, fieldPath)
        case _                            => Left(s"Cannot select field from non-record type")
      }
  }
}
```

Reified Optics: Building an AccessorBuilder

```
object QueryAccessorBuilder extends AccessorBuilder {
  override type Lens[S, A]      = Selector[S, A]
  override type Prism[S, A]     = ???
  override type Traversal[S, A] = ???

  override def makeLens[S, A](product: Schema.Record[S], term: Schema.Field[A]): Selector[S, A] =
    Selector(product, NonEmptyList(term.label), term.schema)

  override def makePrism[S, A](sum: Schema.Enum[S], term: Schema.Case[A, S]): ??? =
    ???

  override def makeTraversal[S, A](collection: Schema.Collection[S, A], element: Schema[A]): ??? = ???
}
```

Reified Optics: Composition

```
case class Selector[V, A](whole: Schema[V], path: NonEmptyList[String], part: Schema[A]) { self =>

    def compose[A1](that: Selector[A,A1]): Selector[V,A1] =
        self.copy(path = self.path ++ that.path, part = that.part)

    def /[A1](that: Selector[V,A1]): Selector[V,A1] = compose(that)

}
```

Reified Optics: Encoding queries

```scala
sealed trait Query[S, A] { self =>
  def apply(value: S): Boolean

  def and[A2](that: Query[S, A2]): Query[S, (A, A2)] = Query.And(self, that)

  def &[A2](that: Query[S, A2]): Query[S, (A, A2)] = Query.And(self, that)

  def or[A2](that: Query[S, A2]): Query[S, (A, A2)] = Query.Or(self, that)

  def |[A2](that: Query[S, A2]): Query[S, (A, A2)] = Query.Or(self, that)
}

object Query {
  case class GreaterThan[S, A](selector: Selector[S, A], that: A) extends Query[S, A] {
    override def apply(value: S): Boolean =
      selector(value).map { v =>
        selector.fieldSchema.ordering.compare(v, that) > 0
      }.getOrElse(false)
  }

  case class LessThan[S, A](selector: Selector[S, A], that: A) extends Query[S, A] {
    override def apply(value: S): Boolean =
      selector(value).map { v =>
        selector.fieldSchema.ordering.compare(v, that) < 0
      }.getOrElse(false)
  }

  case class EqualTo[S, A](selector: Selector[S, A], that: A) extends Query[S, A] {
    override def apply(value: S): Boolean =
      selector(value).map(_ == that).getOrElse(false)
  }
}
```

Reified Optics: A serializable query DSL

```
case class User(id: Long, name: String, age: Int)
object User {
  val schema = DeriveSchema.gen[User]

  val (id, name, age) = schema.makeAccessors(QueryAccessorBuilder)
}

case class Session(id: String, user: User)
object Session {
  val schema = DeriveSchema.gen[Session]

  val (id, user) = schema.makeAccessors(QueryAccessorBuilder)
}

val usersNamedDan = User.name =:= "Dan"

val danSessions = Session.user / User.name =:= "Dan"

val compundQuery = (Session.id > 0) & (Session.user / User.age > 30)
```

Learn more!

- https://github.com/zio/zio-schema
- https://github.com/zio/zio-schema/zio-schema-examples (coming soon!)