depop

MAKING YOUR LIFE EASIER WITH MACROS

https://github.com/depop/json-macros

Andrew (Gus) Gustafson gus@depop.com

Problem - writing out a class using snake case JSON field names

I want to write this:

```
SprayJsonMacros.jsonWriter[Car](toSnakeCase).write(Car(2, 2.4))
SprayJsonMacros.jsonWriter[Airplane](toSnakeCase).write(Airplane(4))
```

So that I can get this output:

```
JsObject("number_of_doors" -> JsNumber(2), "engine_capacity" -> JsNumber(2.4))
JsObject("number_of_engines" -> JsNumber(4))
```

Using some function:

```
def toSnakeCase: String => String
```

But I end up having to write:

```
new JsonWriter[Car] {
  override def write(car: Car): JsValue = {
    JsObject(
      "number_of_doors" -> car.numberOfDoors.toJson,
      "engine_capacity" -> car.engineCapacity.toJson
    )
  }
}
```

Problem - writing JSON for a parent type

Given I have this:

```
sealed trait Vehicle
case class Car(numberOfDoors: Int, engineCapacity: Double) extends Vehicle
case class Airplane(numberOfEngines: Int) extends Vehicle
```

And I have an implicit `JsonWriter[Car]` and `JsonWriter[Airplane]` in scope

When I want to create a `JsonWriter[Vehicle]`

Then I have to write this sort of crap:

```
new JsonWriter[Vehicle] {
  override def write(vehicle: Vehicle): JsValue = {
    vehicle match {
      case car: Car => car.toJson
      case airplane: Airplane => airplane.toJson
    }
  }
}
```

But I really want to just write:

```
val vehicleJsonWriter = SprayJsonMacros.jsonWriterFromSubTypes[Vehicle]
```

Benefits of Macros

- Allows a simple way of generating code
- If we can generate code, then it's simple for us to convert a domain model to a representation in a persistence layer or in json
- DRY principle - duplicating logic is annoying and error-prone
- Duplicated logic blocks across different types can be extracted into a macro, just as duplicated code blocks can be refactored to extract new methods

Tips and Tricks

- Auto-complete and `println` are your friends. Just keep typing until something good happens
- Your macro code needs to be in a separate code base from your domain model

Code - writing out a class using snake case JSON field names

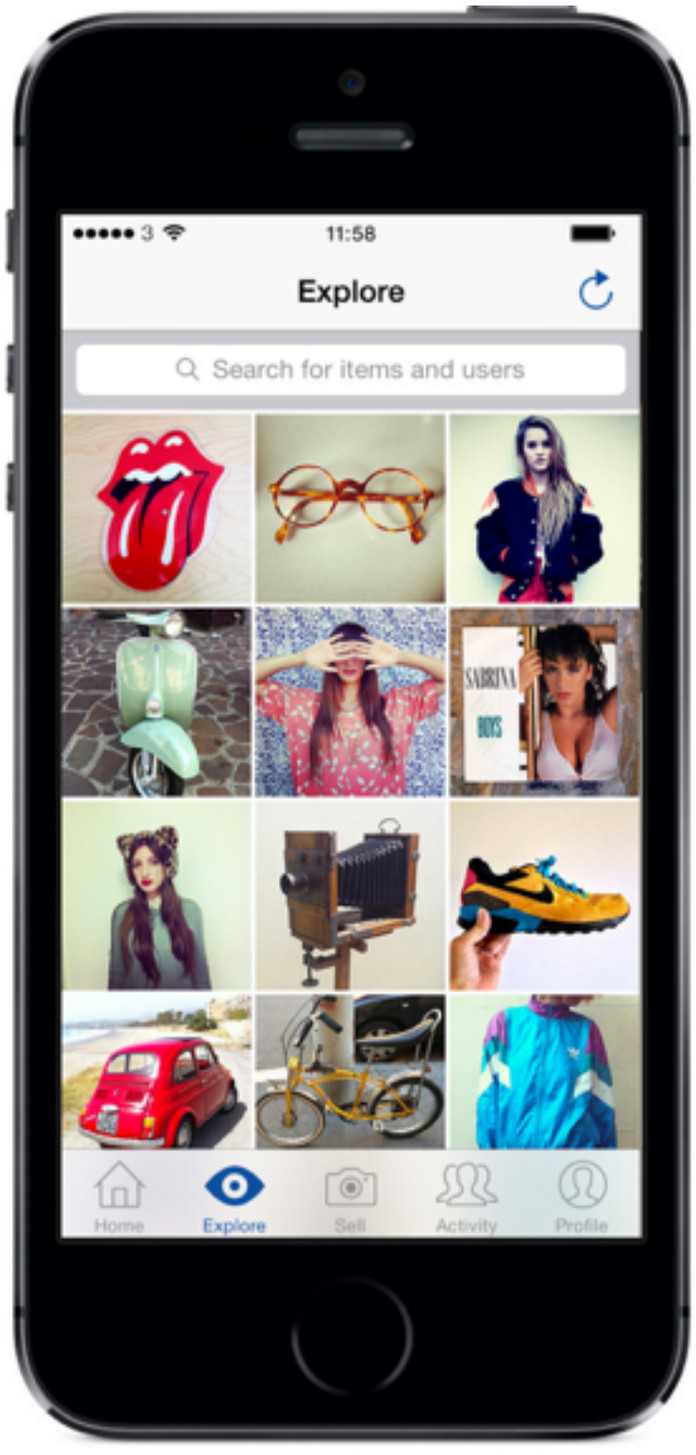
```scala
import spray.json.JsonWriter
import scala.reflect.macros.blackbox.Context

trait SprayJsonFieldNamingMacros {
  def jsonWriterMacro[T : c.WeakTypeTag](c: Context)(nameWriter: c.Expr[String => String]): c.Expr[JsonWriter[T]] = {
    import c.universe._

    // the type representation of T
    val baseType: c.universe.Type = weakTypeOf[T]

    // fields within the type T
    val fields: Iterable[c.universe.MethodSymbol] = baseType.decls.collect {
      case methodSymbol: MethodSymbol if methodSymbol.isCaseAccessor =>
        methodSymbol
    }

    // ensure all JsonWriters exist for field return types
    fields.map(_.returnType).toSet.foreach { fieldType: Type =>
      q"""implicitly[spray.json.JsonWriter[$fieldType]]"""
    }

    // pairs of String -> JsValue which represent json field name -> json field value
    val pairs: List[c.universe.Tree] = fields.toList.map { field =>
      val classFieldName = field.name.decodedName.toTermName
      val jsonFieldName = q"$nameWriter(${classFieldName.toString})"
      q"($jsonFieldName, item.$classFieldName.toJson)"
    }

    // create the JsonWriter
    val jsonWriterType: c.universe.Type = weakTypeOf[JsonWriter[T]]
    val jsonWriter: c.universe.Tree =
      q"""
        new $jsonWriterType {
          def write(item: $baseType): spray.json.JsValue = {
            JsObject($pairs)
          }
        }
      """

    c.Expr[JsonWriter[T]](jsonWriter)
  }
}
```

Code - writing JSON for a parent type

```
import spray.json.JsonWriter
import scala.language.experimental.macros
import scala.reflect.macros.blackbox._

trait ADTJsonMacros {
  def jsonWriterFromSubTypesMacro[T: c.WeakTypeTag](c: Context): c.Expr[JsonWriter[T]] = {
    import c.universe._

    // the type representation of T
    val baseType: c.universe.Type = weakTypeOf[T]

    // all subclasses for the type T which are "known", ie: extending a sealed trait
    val subclasses: Set[c.universe.Symbol] = baseType.typeSymbol.asClass.knownDirectSubclasses

    // the writers for all of the subclasses
    val writers: Set[Any] = subclasses.map { subclass =>
      // function taking an instance of the subclass and returning that instance as json
      val q"{ case $writer }" =
        q""" {
          case item: $subclass =>
            val jsonWriter = implicitly[spray.json.JsonWriter[$subclass]]
            jsonWriter.write(item)
        }
      """
      writer
    }

    val jsonWriterType: c.universe.Type = weakTypeOf[JsonWriter[T]]

    // the JsonWriter for the parent type which matches against each of the subclass writers
    val jsonWriterFound: c.Tree =
      q"""
      new $jsonWriterType {
        def write(item: $baseType): spray.json.JsValue = {
          item match { case ..$writers }
        }
      }
      """

    c.Expr[JsonWriter[T]](jsonWriterFound)
  }
}
```

Links/Resources

- http://docs.scala-lang.org/overviews/quasiquotes/syntax-summary.html is a great summary of quasiquotes syntax and has links on how to use, but doesn't necessarily have a high level overview of how everything might fit together



depop

Depop is your little shop in your pocket.

A global social marketplace

The easiest and fun way to buy and sell

Discover the best things from the community

Privately chat to negotiate an item

Growing fast in the UK and US

3 million users

12 person engineering team (4 Scala)

Looking for 4+ Scala Engineers early next year

gus@depop.com