24 Days of Hackage: aeson

Many of today's programmers would likely consider the ability to parse JSON fairly essential when choosing a programming language. With tools like CouchDB, not to mention a large proportion of web services offering JSON (and sometimes *only* JSON), JSON is unavoidable. Fret not, Haskell is perfectly capable of this task, and in today's post we will look at Bryan O'Sullivan's aeson library.

Aeson is a small library, offering not much in way of an API - something I regard as a feature. Essentially, `aeson` consists of a parser, and a pair of To/From JSON classes to convert between JSON and richer Haskell structures.

These type classes are probably where most people will spend time programming with `aeson`, so let's have a look at these first. The `ToJSON` type class allows you to convert any Haskell value into JSON. `Aeson` comes with some batteries included, allowing you to convert text, numbers, lists, dates, and more. The `FromJSON` type class acts in the opposite direction, mapping a JSON AST into Haskell values. Again, there are many instances out of the box.

Careful readers will have noticed that there are two separate type classes here, rather than one - why is that? After all, wouldn't the API be simpler if we only had one type class with both `toJSON` *and* `parseJSON`? On the one hand, yes, but by having two classes, we are able to be a lot more flexible. Perhaps our Haskell values contain internal identifiers in their construction, something we don't want to expose in our JSON web service, so it would be impossible to implement `parseJSON`. By splitting the type classes, this is now no longer a problem.

I think it's time for an example!

```
data Cheese = Cheese { cheeseMaturity :: Maturity
```

```
                    , cheeseWeight :: Double
                    -- etc
                    }

data Maturity = Strong | Mild

instance ToJSON Maturity where
  toJSON Strong = String "strong"
  toJSON Mild   = String "mild"

instance FromJSON Maturity where
  parseJSON (String "strong") = pure Strong
  parseJSON (String "mild")   = pure Mild
  parseJSON _                 = fail "Unknown cheese strength!"

instance ToJSON Cheese where
  toJSON Cheese{..} = object [ "maturity" .= cheeseMaturity
                             , "weight" .= cheeseWeight
                             ]

instance FromJSON Cheese where
  parseJSON (Object o) = Cheese <$> o .: "maturity"
                                <*> o .: "weight"
  parseJSON _ = fail "Failed to parse cheese!"
```

As you can see, `aeson` provides some nice combinators for implementing `toJSON` - `object` and `.=` combine to provide an expressive readable EDSL for building up JSON structures. As I'm stressing throughout this series, combinators let us build up complexity from small building blocks. Combinators, combinators, combinators!

`FromJSON` also benefits from a rich parser, and this time you don't have to learn anything new! All parsing is done in the `Parser Monad`, which is also an `Applicative` functor, which we take advantage to construct `Cheese`!

`Parser` provides even more than that though - there is also an `Alternative` instance so you can try parsing multiple representations - which comes in really handy when dealing with multiple versions of a web service!

`Aeson` can also be used to work directly with the AST too , you don't have to write

`To/FromJSON` instances if you don't want. The AST is simple, in fact the Value data type is just the sum of string, array, object, number and null constructors. Chris Done has written some great documentation on why you might want to do this. Hopefully soon `aeson` will have a new release and this will be on Hackage for all the world to see.

So there we have it, a great API with a fast implementation and type safety - is there anything Haskell can't do?

You can contact me via email at ollie@ocharles.org.uk or tweet to me @acid2. *I share almost all of my work at GitHub.*

I accept Bitcoin donations: `14SsYeM3dmcUxj3cLz7JBQnhNdhg7dUiJn`. Alternatively, please consider leaving a tip on 0
