aeson: Working with JSON data

aeson is an efficient and easy to use library for decoding (parsing) and encoding JSON. The ways to work with aeson ranges from simple (but inflexible) to complex (but flexible).

API docs

The API documentation can be found here. There's plenty of good information in the API docs.

Decoding and encoding standard data types

JSON strings can trivially be decoded to Haskell values, provided that the types of those values are instances of `FromJSON`. Analogously types that are instances of `ToJSON` can easily be encoded back to JSON strings. aeson comes with convenient `FromJSON` and `ToJSON` instances for the standard data types (numbers, booleans, lists, tuples, dates, and so on).

Let's try these instances out in GHCi. You'll need to import `Data.Aeson`, `Data.Map`, `Data.Text` and `Data.Time.Clock` in order to tr out the examples above. You'll also need to launch GHCi with `ghci -XOverloadedStrings`.

```
λ> decode "true" :: Maybe Bool
Just True

λ> encode True
"true"

λ> decode "[1, 2, 3]" :: Maybe [Int]
Just [1,2,3]

λ> encode [1, 2, 3]
"[1,2,3]"

λ> decode "\"1984-10-15T00:00:00Z\"" :: Maybe UTCTime
Just 1984-10-15 00:00:00 UTC

λ> encode (read "1984-10-15 00:00:00 UTC" :: UTCTime)
"\"1984-10-15T00:00:00Z\""

λ> decode "{ \"foo\": 0, \"bar\":1, \"baz\": 2 }" :: Maybe (Map Text Int)
Just (fromList [("bar",1),("baz",2),("foo",0)])

λ> encode (fromList [ ("bar" :: Text, 1 :: Int)
                    , ("baz" :: Text, 2 :: Int)
                  , ("foo" :: Text, 0 :: Int) ])
ar\":1,\"baz\":2,\"foo\":0}"
```

Automatic decoding and encoding

So how do you define `FromJSON` and `ToJSON` instances for types that don't have them?

One simple way to do this is to use the `DeriveGeneric` extension and have the compiler generate the implementation for you.

```
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE OverloadedStrings #-}
import Data.Aeson
import Data.Text
import GHC.Generics

data Person = Person { name :: Text, age :: Int } deriving (Generic, Show)

instance FromJSON Person
instance ToJSON Person

main :: IO ()
main = do
    putStrLn $ "Encode: " ++ (show (encode (Person { name = "Joe", age = 12 })))
    putStrLn $ "Decode: " ++
        (show (decode "{ \"name\": \"Joe\", \"age\": 12 }" :: Maybe Person))
```

For compatibility reasons, the default `DeriveGeneric` `ToJSON` implementation is relatively slow. So for better efficiency, you might war to override the `toEncoding` function `ToJSON` instance with:

```
instance ToJSON Person where
    toEncoding = genericToEncoding defaultOptions
```

You can also derive instances at compile-time, using `Data.Aeson.TH` and the `TemplateHaskell` extension.

Decoding and encoding in your own way

`FromJSON` and `ToJSON` instances can also be written by hand.

For `FromJSON` (decoding), you need to define the `parseJSON` function, which takes a JSON value (of the aeson type `Value`) and return a Haskell value of your type. In order to define your own `ToJSON` instance, you need to do the reverse - create a `Value` from a Haskell value of your type.

```
#!/usr/bin/env stack
-- stack script --resolver lts-12.21
{-# LANGUAGE OverloadedStrings #-}
import Control.Applicative
import Data.Aeson
import Data.Text (Text)

-- Same type as before, but without the Generic instance.
data Person = Person { name :: Text, age :: Int } deriving Show

-- We expect a JSON object, so we fail at any non-Object value.
instance FromJSON Person where
    parseJSON (Object v) = Person <$> v .: "name" <*> v .: "age"
    parseJSON _ = empty

instance ToJSON Person where
    toJSON (Person name age) = object ["name" .= name, "age" .= age]

-- The main function is unchanged from before.
main :: IO ()
main = do
    putStrLn $ "Encode: " ++ (show (encode (Person { name = "Joe", age = 12 })))
    putStrLn $ "Decode: " ++
        (show (decode "{ \"name\": \"Joe\", \"age\": 12 }" :: Maybe Person))
```

Above, `parseJSON` constructs the Haskell value using a JSON object. If some other kind of JSON value is provided, decoding will fail. Note that `parseJSON` above operates in an `Applicative Parser` context. This is what `parseJSON` could look like using a monadic computation:

```
instance FromJSON Person where
    parseJSON (Object v) = do
        name <- v .: "name"
        age <- v .: "age"
        return (Person { name = name, age = age })
    parseJSON _ = empty
```

The `toJSON` function above uses the `object` function to define a JSON object, which takes a list of key-value `Pair` values. The `Pair` type is simply defined as `(Text, Value)`, and `.=` is a helper to encode the right-hand side to its corresponding JSON value.

Working with a arbitrary JSON data

Sometimes you want to work with the JSON data without actually converting it to some specific type first. One way to do this is to work with the JSON abstract syntax tree (AST). This can be done by simply decoding it to a `Value`, like so:

```
λ> decode "{ \"foo\": false, \"bar\": [1, 2, 3] }" :: Maybe Value
Just (Object (fromList [("foo",Bool False),("bar",Array [Number 1.0,Number
2.0,Number 3.0])]))
```

You can also decode your JSON string to any of the JSON types that `Value` encapsulates:

JSON	aeson Type	Standard Type

Array	`Array`	`Vector Value`
Boolean	`Bool`	`Bool`
Number	`Number`	`Scientific`
Object	`Object`	`HashMap Text Value`
String	`String`	`Text`

Once you have a JSON value, you can define a `Parser` like we did in `parseJSON` above, and run it using `parse`, `parseEither` or `parseMaybe`.

Decoding JSON to a specific Haskell type is actually a two-step process - first, the JSON string is converted to a `Value`, and then the `FromJSON` instance is used to convert that `Value` to the specific type.