# Cloud.Haskzure.Gen


| | |
|---|---|
| **Copyright** | (c) Nashwan Azhari, 2016 |
| **License** | Apache 2.0 |
| **Maintainer** | aznashwan@yahoo.com |
| **Stability** | experimental |
| **Portability** | POSIX, Win32 |
| **Safe Haskell** | None |
| **Language** | Haskell2010 |


Exposes various helpers useful for generation of instance declarations.

## Main instance generation functions:



```
azureResourceInsts :: Name -> Q [Dec]
```

Generates `Monoid`, `ToJSON`, `FromJSON`, and `AzureResource` instances for the datatype given by its name. For more details, see `monoidInst`, `toJSONInst`, `fromJSONInst` and `azureResourceInst` for more details.

```
mkJSONInsts :: Name -> Q [Dec]
```

Generates instances for `ToJSON`, `FromJSON` and `Monoid` provided a type which is an instance of `Generic`. See `toJSONInst`, `fromJSONInst` and `monoidInst` for more details.

```
azureResourceInst :: Name -> Q [Dec]
```

Generates an `AzureResource` instance for the datatype provided by its `Name`. The fields of the datatype have their names matched to the fields of `AzureResource` by dropping the prefix which is the name of the datatype as opposed to the common field prefix in the names of the AzureResource fields. For Example:

```
data SomeData = SomeData {
     someDataID :: String,
     someDataName :: String,
     someDataType :: String,
     someDataLocation :: String
 }


instance AzureResource SomeData where
 rID = someDataID
 rName = someDataName
 rType = someDataType
 rLocation = someDataLocation
```

```
toJSONInst :: Name -> Q [Dec]
```

Generates a `ToJSON` instance provided a datatype given by its `Name`.

The given datatype MUST have a single value constructor of record type. Also, the data structure MUST be an instance of `Generic`.

The generated instance relies on `toEncoding`, and all of its fields will be named following the convention that they are named with the WHOLE name of the structure as a prefix as per example:

```
data TestData = TestData {
 testDataField1 :: Field1Type,
 testDataField2 :: Field2Type
 } deriving Generic
```

With the resulting JSON looking like:

```
{
 "field1": encodingOfField1,
 "field2": encodingOfField2
}
```

**fromJSONInst** :: Name -> Q [Dec] #

Generates a FromJSON instance provided a datatype given by its Name.

The given datatype MUST have a single value constructor of record type and be an instance of Generic. In addition, the types comprising the fields of the datatype must be an instance of Monoid in order to facilitate defaulting. The generated instance acts like the exact inverse of toJSONInst, in that the data structure must have all record fields with its name as a prefix, whilst the decoding process expects the JSON fields to be without. For example:

```
{
 "field1": encodingOfField1,
 "field2": encodingOfField2
}
```

The above is expected to be decoded into the following structure:

```
data TestData = TestData {
 testDataField1 :: Field1Type,
 testDataField2 :: Field2Type
 } deriving Generic
```

**monoidInst** :: Name -> Q [Dec] #

Generates a Monoid instance for the datatype with the provided Name.

The datatype MUST be an instance of Generic, with the type of all of its contained felds also Monoid instances themselves.

## Instance generation utilities:

**recordFieldsInfo** :: (VarBangType -> a) -> Name -> Q [a] #

reifys the simple type given by Name and returns the result of applying the given VarTypeBang (or VarStrictType in template-haskell <= 2.11.0) -applicable function to all the found records. This function makes hard presumptions about the provided type Name. Particularly, it expects it to be a datatype with a single value constructor which is of record type.