Cloud.Haskzure

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

Haskzure containings bindings to be used with the new Azure Resource Manager APIs.

Core components:

Contents


```
class (ToJSON a, FromJSON a) =>
AzureResource r a | r -> a where                    #
```

AzureResource is the typeclass to which all AzureResource resource datatypes must comply in order to be deployed. It should be directly serializable to/from JSON.

Minimal complete definition

```
rID, rName, rLocation, rType, rProperties
```

Methods

```
rID :: r -> Text                                    #
```

rID returns the Text ID of the AzureResource:

```
rName :: r -> Text                                  #
```

rName returns the Text name of the AzureResource:

```
rLocation :: r -> Text                              #
```

rLocation returns the normalized String location of the AzureResource:

```
rType :: r -> Text                                  #
```

rType returns the String Type of the AzureResource in 'Provider/ResourceType' form:

```
rProperties :: r -> a                               #
```

rProperties returns the set of properties specific to this AzureResource:

Instances

```
(ToJSON a, FromJSON a) => AzureResource (Resource a) a
                                                    #
```

```
data Resource a                                     #
```

Resource defines the core Resource ATD which will be used to model Azure resources. Resource is an instance of AzureResource.

Resource is the basic implementation of a generic Azure resource:

Constructors

```
Resource
```

`resID :: Text`	resID is the ID of the Resource:
`resName :: Text`	resName is the Name of the Resource:
`resLocation :: Text`	resLocation is the Location of the Resource:
`resType :: Text`	resType is the Type of Resource:
`resProperties :: a`	resProperties are the properties of the Resource:

Instances

```
Eq a => Eq (Resource a)                                                    #
Show a => Show (Resource a)                                                #
ToJSON a => ToJSON (Resource a)                                            #
FromJSON a => FromJSON (Resource a)                                        #
Default (Resource Value)                                                   #
(ToJSON a, FromJSON a) => AzureResource (Resource a) a                     #
```

Instance Generation helpers and utilities:

```
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.

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

`reify`s 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.