Schematiks Limited

ALFA Newsletter

1st November 2020

Introduction

Welcome to the 1st issue of the ALFA newsletter!

Given some significant and exciting upcoming developments in ALFA, we thought it is best to communicate these using a newsletter.

A major theme for version 0.9 has been comprehensive support for expressions in the language extending the support introduced in version 0.8. The following features will be released as part of version 0.9 of ALFA SDK and RTL (Runtime Libraries).

For further questions, please contact us - info@schematiks.io.

Contents:

1. Fields based on expressions

Assigning *default* values or having a field's value being *calculated* by a custom function is a requirement many modelers and developers come across. Without such a capability, the model can only document the *expected* value and it is left to the developer to assign the correct values. This can lead to gaps in implementation or mis-interpretation of requirements.

With ALFA field expressions, such default values and calculated values can now be expressed as part of the model, removing the potential for error and burden on developers.

Fields within user-defined-types will now be capable of being assigned literal values or expressions. This can be used to express default or calculated values. Expressions include calls to service functions enabling your service implementations to return values to be assigned to fields. Expressions can be literal values or values dependent on other fields, which opens the door to a host of possibilities.

Refer to the ALFA code below for an example of field expressions.

```
record Request {
   VersionTimestamp : datetime = timestamp()
   VersionHost : string = AuditHelper::getHost()
   Type : RequestType = RequestType.New
   Metadata : map< string, string > = { "source" : "unknown", "type" : "unknown" }
   TimeToLiveSecs : int = 600
   Payload : string
   MessageSize : long = len( Payload ) + 64
}

service AuditHelper() {
   getHost() : string
}

enum RequestType {
   New Cancel
}
```

ALFA data model with fields assigned to values or functions

Some default values can only be derived from custom code. The example model above requires the host name to be assigned. For that purpose, ALFA will call out to an implementation of the service `AuditHelper`.

The Alfa code generators evaluate what fields will be assigned in the constructor or the final build method. For example, given the ALFA definition above, the Java generated builder class constructor initialises fields with literal expressions.

```
  private RequestBuilder(alfa.rt.IBuilderConfig cc) {
     setVersionTimestamp(java.time.LocalDateTime.now());
     setVersionHost(
             builderConfig()
                     .getServiceFactory(datahub.AuditHelper.Factory.class)
                     .create().getHost());
     setType(datahub.RequestType.New);
     setMetadata( new java.util.HashMap<java.lang.String, java.lang.String>() { {
                     put("source", "unknown");
                     put("type", "unknown");
                 } } );
     setTimeToLiveSecs(600);
  }
```

Generated Java code snippet for default value initialisation

The generated Java `build()` method contains assignments to calculated values. Calculated value field's `set*` methods are `private`, so cannot be invoked from outside of the class.

Those paying close attention will notice that `setMessageSize` is being invoked in the build method and not the constructor as `MessageSize` has a dependency on the **`_payload`** field.

```
  public Request build() {
     setMessageSize(_payload.length() + 64);
     ...
  }
```

Generated Java code snippet for setting field based on functions

This pattern will apply to all **ALFA** generated target runtime languages.

2. Builtin functions to interrogate external data sources

In addition to the long list of existing functions, **ALFA** now includes 2 new builtin functions - **`query`** and **`lookup`**. These can be used to access external data from sources such as a database.

```
record Order key (Id : string ) {
    OrderedBy : CustomerKey
    Value : dub
}

   ...
   let highValOrders = query( Order, e -> e.Value > 100 )
   let lastOrder = lookup( Order, new OrderKey(“abc”) )
   ...
```

Example of query() and lookup() function usage

The `query` function takes a lambda expression that is converted to a **push-done predicate** to the underlying datastore. The `lookup` function simply loads an entity based on a key.

The underlying datastore implementation is pluggable, so data can be sourced from any type of data store (database, middle tier cache etc). The `query` and `lookup` functions are ideally placed to be used in `assert` or `library` definitions to express logic based on external data sources.

3. Using decision table expressions

Decision table definitions is a useful technique to express multiple input conditions and outputs produced. Such tables can run into 10s, potentially 100s of rows expressing complex inter-relationships between attributes.

Although these can be expressed as `if` and `else` expressions, such code becomes verbose and difficult to maintain. ALFA introduces a simple syntax to define decision tables.

In the example below, given 3 inputs, the output is evaluated. The output can be refined to capture a single or all matching conditions. The output can even be an expression, e.g. a function call.

The example below shows a trivial ALFA decision table. You can think of other examples such as insurance premiums, financial ratings etc.

```
...
decideSport( temperature : int, humidity : int, wind : int ) : try< string > {
     let sport =
         ( temperature,   humidity,   wind  ) match {
         ( [20 .. 35] ,   < 85    ,   < 20  ) => "Tennis"
         ( [10 .. 35] ,   < 85    ,   < 10  ) => "Cycling"
         ( > 20       ,   *       ,   >= 20 ) => "KiteFlying"
         ( > 35       ,   *       ,   *     ) => "Chess"
         }
     return sport
 }
...
```

Example decision table expression

Decision tables are generated to Java for ALFA version 0.9 with other languages to follow. The generated Java code is highly optimised to execute the table cell rules concurrently to get the result in the shortest time.

4. Populating objects using data from a relational database

For those using Java/JVM languages, and ALFA to Java/JPA generator, it is possible directly access data stored in relational databases and have relational data automatically converted to ALFA generated Java objects. The returned objects undergo the validation checks expressed in the ALFA model.

All database interactions - insert, update, delete and query are supported.

The code snippet below shows how using ALFA integration with JPA, an object is fetched from a database.

```
Map<String, String> settings = new HashMap<>();
...
EntityManagerFactory emf = Persistence.createEntityManagerFactory("AlfaHrJpa", settings);
entityMgr = emf.createEntityManager();

Key ek = EmployeeKey.newBuilder().setEmployeeId(123).build();

Optional<AlfaObject> empDetails = AlfaJpaUtils.find(entityMgr,
                                                    EmployeeDetailsDescriptor.INSTANCE, ek);
```

Interfacing JPA and generated Java objects

5. Maven plugin and Java runtime group name update

Currently the Maven Plugin and Java runtime uses `io.alfa.*` as the Maven coordinate. To be consistent with the internet domain name, this will change to **`io.alfa-lang.*`** from version 0.9 onwards.

6. Maven plugin installation update

Detailed steps on installing the ALFA Maven plugin has been added under the tools section.
https://alfa-lang.io/tools/mvnplugin.html

This now includes step-by-step instructions to install the plugin into an Artifactory repository.