LIGADATA

Fatafat 1.0 - Implementing Custom Messages and Containers using Fatafat Metadata

This document describes how to build custom messages and containers. It also covers the Fatafat data transformation and consumption methods.

Prerequisites

The developer will need to have access to the input data definitions, for example, the input database table definitions, a web log of transactions, a .csv file, or other form of structured data.

Introduction

The basic process in creating messages and containers is:

1. Identify which fields are the targets for this message operation.
2. Identify any subsetting or filtering that needs to occur.
3. Identify any mapping of fields or fields holding complex datatypes that may need to be mapped to individual fields.
4. Create the JSON input definition for the example data.
5. Add the JSON input definition to the Metatdata using a (REST) API call.

Specify the Field Name and Type

An essential part of creating a FataFat application is specifying the field name and type information for the incoming message to be consumed by the models running on the FataFat platform. The material presented here will introduce the reader to the key aspects of how this is done.

Data can be sent from a variety of sources:

- SQL database
- Web log
- Web application

Future:

- XML
- Comma delimited text files or .csv
- SPS, SAAS, Stata, Systat or other

Each source will have the requisite resources needed to identify the fields available from that source and their possible type representations. Typically the source will have far more data fields than are really desired. We manage this by using simple projection syntax that allows for the creation of just the essential fields even when all of them are being sent through the input queue system concerned.

Fixed versus Mapped Messages

Fixed messages are simple containers where every field in the resulting message must have a value. In the case of "mapped" messages, not all fields from the input data will be present in the mapped data. The list of fields in a Fixed Message describes the domain of the fields that could be present. This is quite useful where partial messages are received or a wide variety of messages can be described with the same message. This is also known as "sparse" data. Fatafat uses Fixed messages to process JSON, CSV, and XML.

Mapped messages are designed to allow for consumption of large records containing large numbers of fields. Records are converted using the TransformData section to a message or container element to be consumed by the engine. Mapped messages are designed to be a simple mapping of the incoming fields to model elements. A mapped message will have all the fields that were present in the input data. Mapped Messages can process JSON and XML.

A key benefit of Mapped Messages is the ability to consume unordered fields into the Fields section, representing all of the incoming possible fields.

At this writing, message definitions are expressed with JSON syntax, a simple key value pair syntax that is highly readable and one that many editors support with syntax and highlighting specialization. XML and CSV syntax are to be supported in the near future.

Building a Fixed Message

Let's examine the example below of a small message that illustrates the syntax of a message definition. Our example creates a message for a retail store sale. In this version, there is only one purchase item in the record:

```
{
  "Message": {
    "NameSpace": "System",
    "Name": "StoreSaleMsg",
    "Version": "00.01.00",
    "Description": "Store Sales Msg",
    "Fixed": "true",
    "TransformData": {
      "Input": [
        "prt_col",
        "ent_seg_typ",
        "sys_date",
        "sys_time",
        "state_id",
        "cust_id",
        "product_id",
```

```
      "price",
      "qty",
      "total",
      "clerkId",
      "fil2",
      "nar2",
      "nar3",
      "nar4"
    ],
    "Output": [
      "sys_date",
      "sys_time",
      "state_id",
      "cust_id",
      "product_id",
      "price",
      "qty",
      "total"
    ],
    "Keys": [
      "state_id",
      "cust_id"
    ]
  },
  "Fields": [
    {
      "Name": "sys_date",
      "Type": "System.Int"
    },
    {
      "Name": "sys_time",
      "Type": "System.Int"
    },
    {
      "Name": "state_id",
      "Type": "System.String"
    },
    {
      "Name": "cust_id",
      "Type": "System.String"
    },
    {
      "Name": "product_id",
      "Type": "System.Int"
    },
    {
      "Name": "price",
      "Type": "System.Double"
    },
    {
      "Name": "qty",
      "Type": "System.Int"
    },
    {
      "Name": "total",
      "Type": "System.Double"
    }
```

```
    ]
  }
}
```

Fixed Message Definition

Let's examine the first section.

```
{
  "Message": {
    "NameSpace": "System",
    "Name": "StoreSaleMsg",
    "Version": "00.01.00",
    "Description": "Store Sales Msg",
    "Fixed": "true",
```

This simply describes the name of the message as it will be known in the metadata manager, the repository for all message, container, type and function definitions used in the FataFat system. In this case the name of the message is "StoreSalesMsg" and has a namespace of "System". Each message has a version and a description. The field "Fixed" indicates that this message will be represented internally as a structure with a fixed number fields ("Fixed" : "true") as opposed to the alternative structure representation as a map ("Fixed" : "false").

Transforming the input to a defined Message

The TransformData function is used when you are selecting only a subset of the fields available in the input data. It also converts the data to types and arrays that will be used in the processing step. In the next section the TransformData Function is described. This "Input" subsection describes all of the fields coming to FataFat in the order they are used in the created message.

```
"TransformData": {
      "Input": [
        "prt_col",
        "ent_seg_typ",
        "sys_date",
        "sys_time",
```

Ordinarily, not all fields are needed for processing. From the `TransformData`, the Fatafat ingestion component can do a simple projection of fields that are essential and discard the rest. The "Output" subsection of the TransformData describes the essential fields to be more thoroughly described in the "Fields" section of the specification.

Before looking at that, there is another important section of the `TransformData` that needs to be addressed, namely the "Keys" section. It describes the partition key for the data. This is an essential bit of information needed so that processing messages can be distributed on the Fatafat cluster, allowing for parallel processing and dramatic improvement in throughput and general overall scalability. In our example, there are two "`Keys`":

```
"Keys": [
```

```
        "state_id",
        "cust_id"
      ]
```

Evidently in this retail store, it is possible to have the same cust_id number in different states in which the store operates. This needs to be managed properly in the transform.

Finally there is the "Fields" section that defines the field names and their system types. This is how they will be represented internally by FataFat. This is also how they will look when presented to the PMML models that show interest in the message:

```
"Fields": [
    {
      "Name": "sys_date",
      "Type": "System.Int"
    },
    {
      "Name": "sys_time",
      "Type": "System.Int"
    },
    {
      "Name": "state_id",
      "Type": "System.String"
    }
```

Note that the types are qualified with the namespace and refer to types in the metadata cache that can be used to construct messages. Quite complex types can be declared in the FataFat type system cached in the metadata. For messages, you can have also have "message types", called "container types" as a field type. Arrays and ArrayBuffer of scalars as well as of messages are supported.

For example, it is highly unlikely that anyone would design a sales message that was limited to one item purchase. In a real system, there would be multiple records describing each item, the unit price, and quantity purchased.

Mapped Message Definition

Mapped messages look very similar to what you have seen above, except the "Fixed" key in the first section of the JSON has the value "false".

Referring to the definition of Mapped messages from above, please remember – Mapped Messages are designed to allow for consumption of large records containing large numbers of fields. Records are converted using the TransformData section to a message or container element to be consumed by the engine. Mapped messages are designed to be a simple mapping of the incoming fields to model elements. A mapped message will have all the fields that were present in the input data.

A key benefit of Mapped Messages is the ability to consume unordered fields into the Fields section, representing all of the incoming possible fields.

Retail Sales Example – a more complex (and realistic) version

Instead of `product_id, price`, and `qty` being part of the main message, let's permit more than one product to be sent via one message. To do this, we will create a small container type that can be used to represent these three fields. Here is what this looks like:

```
{
  "Container": {
    "NameSpace": "System",
    "Name": "SalesItem",
    "Version": "00.01.00",
    "Description": "Store Sales Item",
    "Fixed": "true",
    "Fields": [
      {
        "Name": "product_id",
        "Type": "System.Int"
      },
      {
        "Name": "price",
        "Type": "System.Double"
      },
      {
        "Name": "qty",
        "Type": "System.Int"
      }
    ]
  }
}
```

Containers are simple. Like messages they too can be mapped. In this case, they are fixed. All three fields are presented in each message.

To use this container in the `StoreSaleMsg`, it needs to change. For brevity, only the `Fields` section of the `SalesMsg` is shown. Notice that the `product_id, price` and `qty` are gone. In their place is an array of `SalesItem` is found:

```
    "Fields": [
      {
        "Name": "sys_date",
        "Type": "System.Int"
      },
      {
        "Name": "sys_time",
        "Type": "System.Int"
      },
      {
        "Name": "state_id",
        "Type": "System.String"
      },
      {
        "Name": "cust_id",
        "Type": "System.String"
      },
```

```
    {
      "Name": "sales_items",
      "Type": "System.ArrayOfSalesItem"
    },
    {
      "Name": "total",
      "Type": "System.Double"
    }
]
```

When the MetadataAPI ingests this JSON to create these messages, some things automatically happen on your behalf. Not only is the message or container compiled and a runtime module created that is used by the engine to serialize and deserialize instances of these messages and containers for use in FataFat, but it will create `ArrayOf`, `ArrayBufferOf`, `SetOf`, `ImmutableArrayOf` each message or container.

It is important to note that the messages and containers that are used in another message or container must previously exist in the metadata. This means that the `SalesItem` container would be compiled by the message compiler before it can be used in the StoreSaleMsg.

Summary

Messages and Containers are used in Fatafat to transform your data so that it can be used by the Fatafat engine. You can identify, subset and filter certain fields for a message operation. Fields can be mapped into new fields, into fields of differing types, and combined into complex datatypes. Similarly, fields can be mapped from complex datatypes to individual fields. Fatafat uses JSON in the input definitions for the your data. Once you have added the JSON input definition to the Metatdata using a (REST) API call, the engine can process your data according to the rules you have defined in the Metadata.

Its time to build some messages and containers. A number of examples are present in the github repository. We encourage you to study these examples and make use of the techniques for your own application