# Johannesburg Stock Exchange

## Post-trade and Information Services

## JSE Services Documentation
## EMAPI TagWire

| | |
|---|---|
| **Version** | 1.1 |
| **Release Date** | 17 Feb 2017 |
| **Number of Pages** | 10 (Including Cover Page) |

# 1 Document Control

## 1.1 Table of Contents


**1 DOCUMENT CONTROL ..... 2**

1.1 Table of Contents ..... 2
1.2 About this Document ..... 3
1.3 Intended Audience ..... 3
1.4 Typographical Conventions ..... 3
1.5 Document Information ..... 3
1.6 Revision History ..... 3
1.7 Related Documents ..... 4
1.8 Contact Details ..... 4

**2 INTRODUCTION ..... 5**

**3 ENCODING ..... 6**

3.1 Encoding of Primitive Type Values ..... 6
3.1.1 Integers ..... 6
3.1.2 Fixed-point Number ..... 6
3.1.3 Boolean ..... 6
3.1.4 String ..... 6
3.2 Encoding of Derived Scalar Type Values ..... 6
3.3 Encoding of Binary Values ..... 7
3.4 Encoding of Structures ..... 7
3.4.1 Messages ..... 7
3.4.2 Fields ..... 7

**4 GRAMMAR IN BNF NOTATION ..... 10**

## 1.2 About this Document

This document describes the syntax of the TagWire encoding of EMAPI messages body.

The purpose of this document, together with its related documents (see 1.7), is to serve as a specification of the EMAPI protocol when implementing an EMAPI client to interface to the JSE's real-time clearing (RTC) system.

## 1.3 Intended Audience

The information in this document is aimed towards EMAPI client software developers who will implement EMAPI message body encoding and decoding.

## 1.4 Typographical Conventions

Messages, fields or enumerators are presented in the `computer style font or courier new.`

## 1.5 Document Information

| | |
|---|---|
| **Drafted By** | Post-trade and Information Services |
| **Status** | Draft |
| **Version** | 1.1 |
| **Release Date** | 17 Feb 2017 |

## 1.6 Revision History

| Date | Version | Description |
|---|---|---|
| 11 May 2016 | 1.0 | Initial draft created. |
| 17 Feb 2017 | 1.1 | 1. Corrected error in BNF grammar. Value 'F' was missing from <hex-digit> definition<br><br>2. Added clarification note on encoding for constants.<br><br>3. Added note on use of extended generic record types – see Section 3.4.2. |

## 1.7 Related Documents

**Note:** The documents in the table below are published on the ITaC website: https://www.jse.co.za/services/itac

| Name | Description |
|---|---|
| Volume PT01 – Post-trade EMAPI Common.pdf | Describes the semantics and syntax of the common or session/admin EMAPI protocol messages. |
| Volume PT02 – Post-trade EMAPI Clearing.pdf | Describes the semantics and syntax of the clearing or application messages of the EMAPI protocol. |
| EmapTransactionsForMember.xml | XML definition of all EMAPI protocol messages for market participants, i.e. clearing and trading members. |
| EmapiTransactionsForMember.html | HTML file describing the syntax of all EMAPI protocol messages for market participants, i.e. clearing and trading members. |
| EmapiTransactions.xsd | The XML Schema that EmapiTransactionsForMember.xml must conforms to. |

## 1.8 Contact Details


| | |
|---|---|
| **JSE Limited**<br>One Exchange Square<br>Gwen Lane, Sandown<br>South Africa<br>Tel: +27 11 520 7000<br><br>**www.jse.co.za** | **Post Trade and Information Services**<br><br>ITAC Queries<br>**Email: CustomerSupport@jse.co.za** |

# 2 Introduction

TagWire is the supported encoding of EMAPI message body used in the JSE's real-time Clearing (RTC) system.

> **Note:** All EMAPI message bodies are preceded by an EMAPI message header, which among other things specifies the encoding used.

Features of the format are:

- Non-binary and human readable; the encoding results in a sequence of characters
- UTF-8 is used as character encoding
- Tag-based as opposed to fixed position (parse data is compact)
- Uses separator/delimiters instead of length indicators
- Only 6 reserved characters

# 3 Encoding

## 3.1 Encoding of Primitive Type Values

### 3.1.1 Integers

An integer is encoded as a sequence of ASCII digit characters. If the number is negative, the minus (i.e. '-') character is appended as the first character in the character sequence.

### 3.1.2 Fixed-point Number

A fixed-point number is encoded as an integer, i.e. without being scaled according to the scaling factor.

**Example:**
If a fixed-point number field's value is 12.3 and it has a scaling factor of 1/1 000 000 it will be encoded as `12300000`

### 3.1.3 Boolean

The Boolean values `true` and `false` will be encoded with the characters `'T'` and `'F'` respectively.

### 3.1.4 String

A string is encoded using the UTF-8 encoding.

An empty string is encoded using the '"' character.

**Reserved characters:**
'=', '[', ']', '|', '"' and '%' are reserved as protocol characters.

When any of these reserved characters occurs in a string, they must be replaced with the character '%' followed by one other character (as specified below) before the string is encoded:

- '=' is replaced with "%1"
- '[' is replaced with "%2"
- ']' is replaced with "%3"
- '|' is replaced with "%4"
- '"' is replaced with "%5"
- '%' is replaced with "%%"

**Note:** After a string has been decoded, the reverse replacement will be done.

**Example:**
The string `"abc=[]|"%123"` is encoded as the character sequence:

`"abc%1%2%3%4%5%%123"`

## 3.2 Encoding of Derived Scalar Type Values

An enumerator of an enumeration type is encoded by encoding the value as done for the type which the enumeration is derived from.

**Example:**
Enumeration Yes with value 1 in the enumeration type YesOrNo (which is derived from the integer type) is encoded as:

`"1"`

## 3.3 Encoding of Binary Values

Binary value (i.e. a sequence of bytes that for example are encoded in non-EMAPI encoding) is encoded as a sequence of two ASCII hex digit characters.

**Note:** ASCII hex digit characters that are letters must be in upper-case.

**Example:**
The binary data `{0, 7, 15, 16, 42, 255}` is encoded as the character sequence:
`"00070F102AFF"`

## 3.4 Encoding of Structures

### 3.4.1 Messages

Messages are started (tagged) with an encoded positive integer ID (which is unique among all messages). The ID is followed by the '=' character that in turn is followed by the start delimiter '[' character. Then the fields are encoded as described in Section 3.4.2 below, and these are separated by the '|' character. The message is terminated by the end delimiter ']' character.

**Note:** The fields of a message are allowed to be encoded in any order.

**Note:** The start/end delimiter characters are always present (even though the message has no fields or the encoding of the fields did not result in any characters).

**Example:**
A message with ID 100 (with no fields) is encoded as:
`100=[]`

### 3.4.2 Fields

**Note:** Fields that have no value set (null value) must not be encoded.

Fields are started (tagged) with an encoded positive integer ID (which is unique among the fields in a message). The ID is followed by the '=' character that in turn is followed by the encoded field's value encoded as described below.

**Note:** Fields inside a (generic) record (array) field have their own ID namespace, i.e., fields inside a record may have the same ID as fields that are "siblings" to a (generic) record (array) field.

#### a) Primitive type, derived scalar type and binary fields

Primitive, derived scalar type and binary field values are encoded as described in sections 3.1, 3.2 and 3.3.

**Examples:**
**1)** A message with ID 100 and with following fields:

- Integer type field with ID 1 which value is 42
- Boolean type field with ID 2 which value is true
- Enumeration type field with ID 3 which value is Yes (which derived type value is of the enumeration type YesOrNo (which is derived from the integer type)

is encoded as:
`100=[1=42|2=T|3=1]`

**2)** A message with ID 100 with an integer type field with ID 1 which value is not set (a.k.a. is null) is encoded as:

```
100=[]
```

### b) Record type fields

Record type field values are started with the start delimiter '[' character and terminated by the end delimiter ']' character. Fields inside a record field are encoded as fields in a message.

**Examples:**

**1)** A message with ID 100 and with following fields:

- integer type field with ID 1 which value is 42
- record type field with ID 2 which in its turn has an integer type field with ID 1 which value is -3

is encoded as:

```
100=[1=42|2=[1=-3]]
```

**2)** A message with ID 100 with a record type field with ID 2 which value is not set (a.k.a. is null) is encoded as:

```
100=[]
```

### c) Generic record type fields

**Note:** Not applicable to EMAPI for JSE.

Generic record type field values are encoded as a message.

**Examples:**

**1)** A message with ID 100 and with a generic record type field with ID 1 which value is a message with ID 200 which in its turn has the following fields:

- integer type field with ID 1 which value is 42
- string type field with ID 2 which value is "" (i.e. empty string)

is encoded as:

```
100=[1=200=[1=42|2="]]
```

**2)** A message with ID 100 and with a generic record type field with ID 1 which value is a message with ID 300 which extends the above message with ID 200 with one field and thereby has following fields:

- integer type field with ID 1 which value is -3
- string type field with ID 2 which value is "Hello"
- boolean type field with ID 3 which value is true

is encoded as:

```
100=[1=300=[1=-3|2=Hello|3=T]]
```

**3)** A message with ID 100 and with a generic record type field with ID 1 which value is not set (a.k.a. is null) is encoded as:

```
100=[]
```

**Note:** Generic Record Type fields are not currently used in JSE EMAPI, i.e. a sub-object in a message will always be of the exact type stated in the message spec, it will not be of an "extending" type. So the sequence "100=[1=200=[..." will not occur, it will always be "100=[1=[..."

### ~~c)~~d) Array type fields

Array type field values are started with the start delimiter '[' character. Then the array elements are encoded as described above and these are separated by the '|' character. The array is terminated by the end delimiter ']' character.

**Note:** Note: The array elements are encoded in the same order as the elements are indexed in the array (i.e. the array elements have a fixed position).

Array elements that are not set (a.k.a. a null value) are encoded as '' (i.e. an empty string). An array type fields value which is empty (i.e. an array that has no elements) is encoded using the '"' character.

**Examples:**

**1)** A message with ID 100 and with following fields:

- string array type field with ID 1 which value is
  `["A", "", null]`

- record array type field
  (where the record has one integer type field with ID 1 and one boolean type field with ID 2)
  with 2 array elements where the fields values in:
  - the first array element are 42 respectively true
  - the second array element are -3 respectively false

- generic record array type field with ID 3 with 2 array elements with below values:
  - the first array element is a message with ID 200 which in turn has following fields:
    - integer type field with ID 1 which value is 72
    - string type field with ID 2 which value is "B"
  - the second array element is a message with ID 300 which extends the above message with ID 200 with one field and thereby has following fields:
    - integer type field with ID 1 which value is -1
    - string type field with ID 2 which value is "C"
    - boolean type field with ID 3 which value is false

is encoded as (with extra whitespace added between the fields of ID 100 to present it more readable):

```
100=[1=[A|"|]|
     2=[[1=42|2=T]|[1=-3|2=F]]|
     3=[200=[1=72|2=B]|300=[1=-1|2=C|3=F]]]
```

**2)** A message with ID 100 and a string array type field with ID 1 which value is [null] (i.e. the array consists of one element which value is not set (a.k.a. is null) is encoded as:

```
100=[1=[]]
```

**3)** A message with ID 100 and an array type field with ID 1 with zero array elements is encoded as:

```
100=[1="]
```

**4)** A message with ID 100 and an array type field with ID 1 which value is not set (a.k.a. is null) is encoded as:

```
100=[]
```

**Note:** The Constants have a declared `type` and are encoded according to the `value` field - NOT the `name` or `encodingId` fields.

# 4 Grammar in BNF Notation

The BNF[1] description of EMAPI TagWire is given below:

```
<message> ::= <tag> '=' <message-body>
<tag> ::= <natural-number>

<natural-number> ::= <non-zero-digit> | <non-zero-digit> <digits>
<non-zero-digit> ::= '1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
<digit> ::= '0' <non-zero-digit>
<digits> ::= <digit>
           | <digit> <digits>

<message-body> ::= '[' <fields> ']' | '[' ']'
<fields> ::= <field>
           | <field> '|' <fields>
<field> ::= <tag> '=' <value>
<value> ::= <primitive-data-value> | <composite-data-value>

<primitive-data-value> ::= <integer> | <boolean> | <string> |
                           <binary>
<integer> ::= <natural-number> | '0' | '-' <natural-number>
<boolean> ::= 'T' | 'F'
<string> ::= <characters> | <empty-string>
<characters> ::= <non-reserved-character> <characters>
               | <escape-characters> <characters>
<non-reserved-character> ::= any unicode character except;
                             '%' | '=' | '[ ' | ']' | '|' | '""'

<escape-characters> ::= '%1' | '%2' | '%3' | '%4' | '%5'| '%%'
<empty-string> ::= '""'
<binary> ::= <hex-digit> <hex-digit>
           | <hex-digit> <hex-digit> <binary>
<hex-digit> ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
              |'A'|'B'|'C'|'D'|'E'|'F'

<composite-data-value> ::= <array> | <message-body> | <message>

<array> ::= '[' <array-elements> ']' | <empty-array>
<array-elements> ::=
     <integer-array-elements> | <boolean-array-elements> |
     <string-array-elements> | <message-body-array-elements> |
     <message-array-elements>
<integer-array-elements> ::= <integer>
           | <integer> '|' <integer-array-elements>
<boolean-array-elements> ::= <boolean>
           | <boolean> '|' <boolean-array-elements>
<string-array-elements> ::= <string>
           | <string> '|' <string-array-elements>
<message-body-array-elements> ::= <message-body>
           | <message-body> '|' <message-body-array-elements>
<message-array-elements> ::= <message>
           | <message> '|' <message-array-elements>
<empty-array> ::= '""'
```

[1] Backus Normal Form (BNF) is a notation technique for context-free grammars that is often used to describe the syntax of computer programming languages and communication protocols. Context-free grammars are often parsed using the recursive descent parser pattern.