Blink Dynamic Schema Exchange Specification

beta4 - 2013-06-05

This document specifies a method for encoding a Blink schema into a sequence of binary Blink messages. It also specifies a set of basic rules for exchanging encoded schemas.

Contents

1 Overview

The Blink Encoding Specification [BLINK] defines the concept of a Blink schema. It also defines a concrete syntax for representing a schema in plain text. This specification defines an alternative representation where a Blink schema is encoded as a sequence of Blink messages. This representation is suitable for dynamic, in-stream, exchange of schemas.

This specification does not mandate a single unique way of exchanging encoded schema messages, but defines a basic set of rules that can be applied in a number of scenarios.

Applications can range from fully static to fully dynamic. A schema received in-stream can still be useful to a static implementation to check for structural consistency. A semi-static implementation can for example use the schema to discover new types that are compatible through inheritance with its predefined types. Other applications will be fully dynamic and will base the decoding fully on the in-stream schema.

The following example shows a schema that defines a message type **Logon** which is then encoded in the same stream as a subsequent instance of that message.

This schema:

```
Logon/1 ->
  string User,
  string Password
```

and a message instance:

```
Logon: User: George Password: abracadabra
```

can be encododed in the same stream like this:

```
2b                // Msg size: 43
81 fa             // GroupDef type ID is 16001
00                // GroupDef.Annotations: NULL
00                // GroupDef.Name.Ns: NULL
05 4c 6f 67 6f 6e // GroupDef.Name.Name: "Logon"
01                // GroupDef.Id: 1
02                // GroupDef.Fields length: 2
00                // FieldDef [0].Annotations: NULL
04 55 73 65 72    //   Name: "User"
00                //   Id: NULL
04                //   Type: Group size: 4
86 fa             //     String type ID: 16006
00                //     String.Annotations: NULL
00                //     String.MaxSize: NULL
00                //   Optional: false
00                // FieldDef [1].Annotations: NULL
08 50 61 73 73 77 //   Name: "Password"
6f 72 64          //   ...
00                //   Id: NULL
04                //   Type: Group size: 4
86 fa             //     String type ID: 16006
00                //     String.Annotations: NULL
00                //     String.MaxSize: NULL
00                //   Optional: false
00                // GroupDef.Super: NULL
14                // Msg size: 20
01                // Logon type ID is 1
06 47 65 6f 72 67 // Logon.User: "George"
65                // ...
0b 61 62 72 61 63 // Logon.Password: "abracadabra"
61 64 61 62 72 61 // ...
```

2 Basic Rules

Before a decoder can decode a message it must know how to map the encoded type identifier to the name of a group definition. It must then determine how to get the actual definition that matches that name.

These two steps can be fully or partially supported by the messages defined here depending on the context:

- If the decoder already knows about a type definition and its name, then it only needs to determine the mapping from the type identifier. The message **`Blink:GroupDecl`** can be used for this purpose. A *group declaration* only specifies a type identifier and a group name. It does not carry any information about the structure of the group.
- If the decoder does not know anything about the message type, it requires both the identifier mapping and the actual structure. This specification defines the **`Blink::GroupDef`** message that enables the required information to be passed in the stream.

The sending application must make sure that a group declaration or a group definition, depending on the context, is available to the receiving application before it sees the corresponding message for the first time.

If a reliable transport like TCP is used, then this means that the relevant parts of the schema appears before the message instance in the stream. If an unreliable transport is used, then the possible scenarios are many and this specification does not define a normative method in this case.

Type and group definitions of the schema can have interdependencies. It is not required that the definitions are transmitted in dependency order. As long as all prerequisites are in place when needed to decode a particular application message instance, the application can send the definitions in any order.

This specification does not require that the complete schema is transmitted before the first application message. A sending application may choose to do so for simplicity, but can also choose to incrementaly interleve the stream of application messages instances with schema messages as needed.

3 Reserved Type Identifiers

An application based on this specification must not use type identifiers in the range 16000 - 16383 for other messages than those defined here.

4 Schema Translation

The mapping from the schema syntax to messages in this specification is straightforward based on the names of the messages and their fields as defined here: Appendix A (page 3) . This section details parts that need special consideration.

A namespace can be specified for a schema file in the schema syntax. There is no direct counterpart to this declaration in the schema messages, but all names must be fully qualified when translated.

Given this schema:

```
namespace Eg
Msg
```

a group definition instance would look like this:

```
GroupDef:
  Name:
    Ns: Eg
    Name: Msg
  ...
```

The same holds for type references; they must be fully resolved before being encoded in a schema message.

Names that are quoted by a backslash in the schema syntax should not keep the backslash when translated:

```
\decimal -> i32 exp, i64 mant
```

translates to

```
GroupDef:
  Name:
    Name: decimal
  ...
```

Schema annotations are translated to **`SchemaAnnotation`** messages. It is recommended that the **`Ns`** field of the translated message is supplied with the namespace that was declared in the schema where the annotation appears:

```
namespace Eg
schema <- @version="1.0"
```

translates to

```
SchemaAnnotation:
  Annotations: [Name: version Value: 1.0]
  Ns: Eg
```

Types in type definitions and in fields are translated to matching dynamic subgroups. The constraints that are specified for the schema syntax must still hold when translated to these groups. This means that:

- a **`Sequence`** subgroup cannot have a **`Type`** field that directly or indirectly resolves to another Sequence.
- an **`Enum`** subgroup can only appear in the **`Type`** field of a **`Define`** message.
- the **`Type`** field of a **`DynRef`** subgroup must resolve to a group definition.
- the **`Super`** field of a **`GroupDef`** message must resolve to a group definition.

The symbols in an enumeration definition can have implicit values in the schema syntax. In the correspoding **`Symbol`** subgroup this is not supported and the value must always be explicitly specified.

Allthough an **`Enum`** component can hold a set of annotations since it inherits from **`TypeDef`**, this set must always be empty. This is because annotations cannot be associated with the enumeration as a whole in the schema syntax.

A Schema for Blink Schemas

```
namespace Blink

GroupDecl / 16000 : Annotated ->
  NsName Name, u64 Id

GroupDef / 16001 : Annotated ->
  NsName Name, u64 Id?, FieldDef [] Fields,
  NsName Super?

FieldDef : Annotated ->
  string Name, u32 Id?, TypeDef* Type, bool Optional

Define / 16002 : Annotated ->
  NsName Name, u32 Id?, TypeDef* Type

TypeDef : Annotated

Ref / 16003 : TypeDef ->
  NsName Type

DynRef / 16004 : TypeDef ->
  NsName Type

Sequence / 16005 : TypeDef ->
  TypeDef* Type

String / 16006 : TypeDef ->
  u32 MaxSize?

Binary / 16007 : TypeDef ->
  u32 MaxSize?

Fixed / 16008 : TypeDef ->
  u32 Size

Enum / 16009 : TypeDef ->
  Symbol [] Symbols

Symbol : Annotated ->
  string Name, i32 Value

U8             / 16010 : TypeDef
I8             / 16011 : TypeDef
U16            / 16012 : TypeDef
I16            / 16013 : TypeDef
U32            / 16014 : TypeDef
I32            / 16015 : TypeDef
U64            / 16016 : TypeDef
I64            / 16017 : TypeDef
F64            / 16018 : TypeDef
Bool           / 16019 : TypeDef
Decimal        / 16020 : TypeDef
NanoTime       / 16021 : TypeDef
MilliTime      / 16022 : TypeDef
Date           / 16023 : TypeDef
TimeOfDayMilli / 16024 : TypeDef
TimeOfDayNano  / 16025 : TypeDef
Object         / 16026 : TypeDef

SchemaAnnotation / 16027 ->
  Annotation [] Annotations,
  string Ns?

Annotated ->
  Annotation [] Annotations?

Annotation ->
  NsName Name, string Value

NsName ->
  string Ns?, string Name
```

B References

BLINK http://blinkprotocol.org/spec/BlinkSpec-beta4.pdf