DLI Type Description Reference

Generated by Doxygen

20200818T065022Z

Contents

1 Overview

The DLI type description specifies the interface of an object in a human- and machine-readable way. It is in a way similar to XML DTDs and the various attempts to create a JSON schema, because it can be used to understand what a serialized representation of the object is supposed to look like. However, it describes not only properties of the representation, but also of the object itself, which are useful in interacting with the object. Data are presented and manipulated as an hierarchy of interdependent objects, with possibly complex constraints, traits and permissions.

1.1 Target audience

The DLI type description is the basis of the REST-style API of modern DLI controllers. Though being tailor-made for this application, the description system itself is not limited to it, and can be used in other products as well.

1.2 Design goals

- Human readability

  Basic DLI type descriptions are supposed to be readable by humans. Unfortunately this means that the same structure can have different meanings in different contexts, which slightly complicates automated processing.

- Interoperability

  The type descriptions and the objects to be described are expected to be serializable to JSON, a popular interchange format.

- Extensibility

  There are many extension points in the type description format. Clients reading unexpected description elements can understand whether the new elements are critical for their interaction with the object, or not.

1.3 Design non-goals

We explicitly choose not to consider:

- Expressing description format in terms of itself

  There is no need for describing a type for "type description" itself in our applications.

- Completely specifying behaviour and constraints

  Objects can have behaviour or constraints which are impossible or hard to model. You may need to handle error results even if your interaction with the object appears to be correct.

2 Data model

Formats of both the description and the described objects are based on the JSON data format, and most type definitions are borrowed from there.

Objects of an hierarchy are described using an hierarchical description, with the elements' descriptions nested into the parents'. Every object is described with a JSON object with at least the following mandatory fields, each having a value of JSON type "string":

- "title": a short name for the object;
- "description": a longer description of the object;
- "type": underlying representation type.

For example, the following is a valid description of an object holding a velocity reading.

```
{
   "title": "Velocity",
   "description": "Velocity of the car, in m/s",
   "type": "number"
}
```

We'll refer to keys of such descriptions as description keys.

The "title" and "description" are not special and their contents are arbitrary human-readable strings. The "type" is what determines what kind of object is described. We will start with the types present in JSON.

2.1 Scalar types

Scalar data types (having no internal structure and corresponding to simple properties) are:

- "string": a string literal;
- "number": a number literal;

JSON literal numbers cannot be equal to infinity or NaN; special values can be handled in a different manner, which will be discussed below.

Note that binary data are encoded as strings.

2.2 Binary object encoding and description

JSON is known for limited means for expressing binary data: all its strings must be valid UTF-8 code point sequences. Therefore, binary data are encoded as UTF-8 strings whose code points match the octet values of the original data. For example, octet 0x41 is unmodified, as it corresponds to U+0041 'A', and octet 0xEF gets mapped to U+00EF encoded by a two-octet UTF-8 sequence 0xC3 0xAF. The resulting string gets escaped subject to ordinary JSON rules.

If both sides of a communication agree on not treating a particular JSON string as a UTF-8 string, the aforementioned encoding and corresponding decoding can be skipped for that string.

String data are indicated as binary by having their content description key set to a binary MIME type (or a JSON Reduce expression evaluating to it).

2.3 Singleton types

Singleton data types are a special type of scalar types. They are also known as unit types, and they have only a single value:

- "const_true": the constant value true;
- "const_false": the constant value false;
- "const_null": the constant value null.

You may wonder why there's no separate boolean type. Surprisingly, the JSON specification doesn't mention it. "true" and "false" are just separate special values, like "null".

2.4 Container types

JSON defines two container types, which can hold values of other types:

- JSON arrays are ordered lists with integer keys starting with 0, like:

```
[1,"a",true]
```

- JSON objects are unordered maps with string keys like:

```
{"name":"fred","is_admin":true}
```

In DLI type description data model, a container can either have an arbitrary number of keys, all with the same type (called the element type, and we will call such containers homogeneous), or a set of pre-specified keys (which we will call fields), each with its own type (which we will call heterogeneous). So, we distinguish the following four types, which are based on the JSON ones but additionally constrain their values:

container	JSON object-based	JSON array-based
heterogeneous	object	tuple
homogeneous	map	array

Type descriptions themselves don't contain "type":"map" or "type":"tuple"; instead, objects are distinguished from maps, and arrays from tuples, by the existence of the "fields" or "element" description keys (see below).

An important difference is that you can only add or remove elements from homogeneous containers (maps and arrays).

Sample object:

```
{"name":"fred","is_admin":true}
```

Sample tuple:

```
["fred",true]
```

Sample map:

```
{"bob":true,"fred":false,"jane":true}
```

Sample array:

```
["bob","fred","jane"]
```

All JSON object-based values have an additional implicit constraint on the keys: they cannot be empty. For example,

```
{"":"I am empty!"}
```

is a valid JSON object, but it cannot be represented by the type system. The empty string "" is given a special meaning when it appears in the place of an JSON object key name. This is not the only case of an irrepresentable JSON object: to be representable, the object needs to have unique non-empty keys, and their order should not matter.

Homogeneous containers have an additional "element" key in the JSON object describing them, which must be a DLI type description for the element type. For example, this describes an array of strings:

```
{
    "title": "Places list",
    "description": "List of places to visit",
    "type": "array",
    "element":
    {
        "title":"Place",
        "description":"A place to visit",
        "type":"string"
    }
}
```

Heterogeneous containers have an additional "fields" key in the JSON object describing them, which must be a J↩ SON array (for tuple types) or a JSON object (for object types) whose values are DLI type descriptions for elements corresponding to the keys (or indices). For example, this describes a person:

```
{
    "title": "Person",
    "description": "Data related to a person",
    "type": "object",
    "fields":
    {
        "name":
        {
            "title":"Name",
            "description":"The person's full name",
            "type":"string"
        },
        "age":
        {
            "title":"Age",
            "description":"The person's age in years",
            "type":"number"
        }
    }
}
```

Descriptions with "type":"object" or "type":"array" which have neither "fields" or "element" keys are malformed. Likewise, descriptions of any other type which have those fields are malformed as well, because "object" and "array" are the only container types in JSON.

There are also two more types of objects, which don't correspond to JSON types.

2.5 The sum type

The sum type is something that cannot be attributed to a JSON value, it's a type of the object holding it. An object of a sum type can have any value of a specified set of types (if you are familiar with a 'restricted variant' type, or a 'union' type, this is similar).

Descriptions with "type":"sum" must contain a "variants" key which must hold a JSON array of possible descriptions for the value. Descriptions of any other type with a "variants" key are malformed.

For example, a boolean is the sum type of constant true and constant false. Each of the options will have a separate title and description. There are rarely "purely boolean" values: the two states carry additional meaning, like "on"/"off", "enabled"/"disabled", "locked"/"unlocked", etc.

An 'optional' string (string that can be said to have no value at all) can be encoded as the sum type of the string type and constant null (or constant false in some cases).

With additional constraints (see below), the sum type can be used to model enumerations from a fixed list of values.

The type is not a 'tagged variant', that is, it doesn't store the information about which variant is stored, it only stores the data. Therefore, the value must uniquely identify which of the variants it satisfies.

The sum type has the following additional restrictions:

- A sum type cannot be a variant of another sum type (that would be redundant and add unneeded complexity).
- The variant corresponding to a contained value must be uniquely determined by the value itself (no tagging);
- A sum type can only have at most one container variant (the restriction could in theory be lifted but it simplifies a lot of logic in type description processing).

2.6 The call type

The call type is a type of a function call. It stands out from the rest of the type description system because has no value that can be represented in JSON, but in practice some actions (like cycling an outlet) are more naturally modeled with a function call than with interaction with a mutable value.

The call type has "arguments" and "results" type description keys, which contain type descriptions for arguments and results. The arguments type is a tuple containing the function's arguments. If the function has no arguments, it is an empty tuple. The results type is a tuple containing the function's return value. If the function does not return a value, is an empty tuple. If the function returns a non-blob value, or a blob value with a fixed content type, the results type may be a one-element tuple containing the description of the result, which can neither be nor contain call types. Otherwise the results type is a two-element tuple; the first element describes a string indicating the content type, the second element describes a string encoding for the value itself and has a content type key (see below) referencing the first element. Any other argument or result types are currently invalid and reserved for future use.

3 Additional attributes

In addition to "type", "title" and "description", other keys in the JSON object describing the data object can be used to indicate its semantic properties. Such attributes can be related to the object itself (as a mutable entity), or to the particular value that the object holds.

The call type cannot have value-related attributes on it, as it is not associated with a value (the arguments and return values can, though). Most object-related attributes cannot be applied to the call type for the same reason.

The sum type cannot have value-related attributes on it, because the possible values are described in the variants array.

Types in the variants array of a sum type cannot have object-related attributes on them, because it's the sum type that is associated with the object, not the variant type.

3.1 Constraints

Data types can have additional constraints on them. Input data which don't satisfy constraints are rejected. Stored data are expected to conform to constraints as well. For example, a number can be restricted to values greater than zero, or a string can be restricted to be equal to "". Another common constraint is that one value (usually a scalar value) is an index into another value (usually of a container type). This can be seen as an alternative for a sum type for modeling enumerations, but in this case, the enumeration is 'open': the reference side doesn't enumerate what values exactly are; the referee does. This can also be seen as a 'foreign key' relation by people with a relational background.

The constraint is a value-related attribute, so it cannot appear in the call type or the sum type (it can appear in their elements though). The constraint is introduced by a "constraint" type description key. There can only be one constraint, but this does not limit its usefulness as its value is a JSON Reduce expression relative to the object described; the "and" operator can be used to combine constraints. For example, this description constrains the object's value to be non-negative:

```
{
    "title":"Age",
    "description":"The person's age in years",
    "type":"number",
    "constraint":[">=",["ref",""],0]
}
```

See below for the meaning of the "ref" function; ["ref",""] is used to refer to the value of the object (or the value to be assigned to it, if the constraint is checked on assignment).

Keys and indices of container types can also be subject to constraints. In this case, constraints are placed on the element type, and operate on the value of the "refindex" function (see below).

3.2 Traits

Objects can have different semantic traits associated with them. The "traits" key of the type description, if present, contains an object whose keys are the different traits, and the values are JSON Reduce expressions. The expression evaluates to a true value if the object has the trait, or, in general, to the value of the trait if the trait has a non-boolean value.

Some traits are outlined below. Implementations can define other traits.

Traits are object-related attributes, so they cannot appear in elements of the call type or the sum type.

Traits are not transitive; a collection may be read-only but some of its elements can be modifiable, possibly because their values are overlaid from elsewhere.

3.2.1 Generic traits

The following generic traits are defined:

- "label": a common human-readable label which can used to refer to the object (can be calculated depending on the fields of the object or its parents);
- "persistent": the object represents a value in persistent storage; if an object doesn't have this property, its value may be lost on reboot, or is derived from some other values, etc.;
- "set": the object represents an unordered set; duplicates are either invalid or ignored, and order of elements doesn't matter;
- "secret": the object's contents can contain private information and should be displayed in a secure manner;
- "advanced": the object is expected to be visible to advanced users only.

3.2.2 Manipulation traits

Manipulating an entity implies changing its state. Therefore, types whose (immediate) state cannot be changed cannot have manipulation traits associated with them:

- singleton types;
- the (proper) object type;
- the tuple type;
- the call type.

The default behaviour is modeled after a memory cell. You can read from it, and if you write to it, the value stays put to be read back unless someone else changes it. In a multiprocessor system, caches get notified about the change and can catch up with it.

Manipulation semantics is described in terms of the sources, conditions and effects of changes.

The following traits related to the possible sources of change are defined:

- "readonly": client immutable: the value cannot be changed directly by any of the clients, i.e. users of the object (the default is that the value can be changed by clients);

- "mutable": server mutable: the value can be changed by the server, i.e. the owner of the object, as if by itself or indirectly because of clients' actions (the default is that the value cannot be changed by the server).

Clients may wish to track the state of an object via unspecified protocols. Some objects, however, may be impossible to track this way.

The following traits related to the possibility of change tracking are defined:

- "volatile": not trackable: the value's state cannot in principle be tracked, change can occur without the users of the object noticing.

Usually, it makes no sense for a client to modify an object if it already contains the value they wish to set; however, for some objects it may make sense to forcefully set a value which is already present there. Some objects wrap a state changes to which do not occur immediately; the state changes will be visible immediately but the effective value may be changed after some other event happens or some time passes.

The following traits related to the effects of setting an object's value:

- "effectful": if true, writing the same value which the object already has may have a significant side effect;
- "latched": changes to the object do not take effect immediately, but after a delay or possibly some other external action.

3.3 Access control

Objects can have access to them checked, to prohibit administratively forbidden actions. The "deny" key of the type description, if present, contains an object whose keys are the different actions which can be denied, and the values are JSON Reduce expressions which evaluate to a true value if the action is denied.

The following actions can be denied on objects:

- "read": retrieving the value associated with the object;
- "write": changing the value associated with the object;
- "index": using the object as an field for indexing into the grandparent collection;
- "create_element": creating a new object in this collection;
- "erase_element": removing an object from this collection;
- "delete": removing the object from its parent collection;
- "invoke": performing the function call.

Access control settings are object-related attributes, so they cannot appear in elements of the call type or the sum type.

Access control is not transitive; you may be forbidden to modify a collection, but allowed to change a property of its element. However, permissions for child and parent elements interact: for example, to remove an element from its collection, you need to pass the following checks:

- the collection must not have the "readonly" trait;
- you need "write" and "erase_element" permissions on the collection;
- you need "delete" permission on the element.

3.4 Initial value

Sometimes an entity can be created (possibly as a part of another value) but cannot be modified later (has the "readonly" trait). It then retains the original creation value. To make sure that newly created entities have a particular value (possibly opposed to values of internal entities), an "initial" description key is added. It is a JSON Reduce expression.

Entities with an "initial" description key can have their value omitted at creation time. Attempts to supply a different value will fail.

3.5 Synchronization

The "synchronize" description key can be used to indicate that agents which wish to modify an object must supply the current value of other objects (and possibly the same object as well) in the same request. Failure to do so, or value mismatch will cause the modification request to fail. This has several uses:

- "compare-and-set"-style interaction (conditional overwriting);
- requiring proof that the agent knows the current value for security (e.g. password change).

Note that some protocols may not support such operation, and the object server does not necessarily enforce it. The restriction is often enforced by middleware, and details of interaction with it are not discussed here.

The "synchronize" description key contains a JSON array of two-element JSON arrays, in each of which the first element is the synchronization condition (synchronization is not performed if the expression is false), and the second element is a JSON string containing the URI of the object to synchronize to (note that it cannot be a JSON Reduce expression).

3.6 Representation hints

Objects can have representations which can benefit from additional information (forms, etc.).

The "ui_order" description key can contain a JSON array which lists the object keys in the order in which they are to be displayed; unlisted keys go at the end of the list in alphabetical order.

The "composite" description key may only be set on a type (possibly) representing a container; if set to a true value, indicates that its fields or elements (and their descendants, if any) effectively represent a single value or are bound by a tighter relation than usual (note that it cannot be a JSON Reduce expression).

3.7 Content type

String entities can represent some structured data, i.o.w. have a content type associated with them. This is indicated by a "content" description key. It is a JSON Reduce expression. If the content type is not specified, text/plain is implied.

The content type determines, among other things, if the content data are subject to binary encoding rules. The supported text and binary types are implementation-specific. Implementations may supply a database of known content types.

3.8 Physical meaning

Numeric entities can be measurements of some physical values. This is indicated by a "quantity" description key. It is a JSON Reduce expression which can have, among others, the following values:

- "count": number of times something happens;
- "screen_length": screen length, measured in e.g. pixels;
- "time": time, measured in e.g. seconds;
- "voltage": voltage, measured in e.g. volts;
- "current": current, measured in e.g. amperes;
- "energy": energy, measured in e.g. joules;
- "temperature": temperature, measured in e.g. degrees Kelvin;
- "illuminance": illuminance, measured in e.g. lux.

Implementations may supply a database of known quantities and supported units.

The "count" unit may seem redundant, but it can be used to indicate what the number actually means, e.g. that it makes sense to increment/decrement it, to accumulate such numbers, contrary to other numbers which may correspond to bit masks or mode numbers.

Quantities like "time" or "screen_length" are actually differences between certain points (in time or on the screen). To specify an absolute position (a point in time or on the screen), a "reference" description key can be used. For time, the string "1970-01-01T00:00:00Z" specifies the Unix epoch.

4 JSON Reduce and extensions

The JSON Reduce expression language provides a way to encode side-effect-free expressions which can be evaluated to JSON types, in JSON itself, in a relatively natural manner.

Simple types evaluate to themselves in JSON reduce. JSON arrays model function calls, where the first argument is the function name. For more information, please consult the JSON Reduce reference. Only extensions are described here.

4.1 Context of evaluation

Most JSON Reduce expressions related to a type description are evaluated in a context, which corresponds to the value being described. Contexts can be used to refer to one value from another (see below).

Contexts are nested structurally, corresponding to the structural nesting of the values described, so the object and array types introduce nested contexts for their elements. The sum type does not create nested contexts for its variants: all constraints are evaluated in the context of the same value. The call type does not introduce a level of nesting for the arguments, but, as arguments are of the tuple type, they themselves do. The same goes for return values. This means arguments and return values can reference their peers in constraints or traits, but arguments cannot reference return values (that wouldn't make sense) and vice versa.

4.2 The ref function

The ref function returns the value of an object referenced by a URI. It accepts a single argument, which must be a URI string that points to the object. Such URIs are usually, but not always, relative to the context of evaluation. The empty string "" corresponds to the value of the object itself, which is useful when creating constraints. For example, the following is a description of a constant "log" string:

```
{
    "title":"Logarithmic",
    "description":"Logarithmic scale,
    "type":"string",
    "constraint":["==",["ref",""],"log"]
}
```

It may not seem useful at first, but can make sense as an element of a sum type.

Accessing (grand)parent objects can be performed using "..", for example, the following is a description of an IP address assignment protocol, identified by a string value (think "static", "dhcp", etc.) which must be an element of the "known_protocols" map, which is in turn contained in an object two hierarchy levels upper:

```
{
    "title":"IP address assignment protocol",
    "description":"Method for obtaining an IP address for this interface",
    "type":"string",
    "constraint":["[]",["ref","../../known_protocols"],["ref",""]]
}
```

This essentially describes an "open" enumeration, whose elements need to be discovered at runtime.

The mechanism for resolving such references is application-defined.

4.3 The refindex function

Sometimes it is necessary to refer to the index of an object in its parent collection. It is accessible using the refindex function. It accepts a single argument, which must be a URI string that points to the object, and evaluates to its index in the parent collection. Usually, the URI will be empty or contain a series of "../" parent references; otherwise the function would evaluate to the last URI component.