a Documentation

Release 1.0

a

Apr 06, 2020

Contents

API Elements is a structure for describing APIs and the complex data structures used within them. It also provides structures for defining parsing results for parsing API definitions from formats like API Blueprint and Swagger/OpenAPI Format.

CHAPTER 1

Getting Started

To start learning about and using API Elements, please check out the overview, which includes explanations and examples. If you are interested in finding libraries and tools for building and consuming API Elements, you will find this information on the tools page.

CHAPTER 2

Contributing

Feel free report problems or propose new ideas using the API Elements GitHub issues. We are always interested in Pull Requests as well for changes.

CHAPTER 3

About this Documentation

This documentation conforms to RFC 2119, which says:

> The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

MSON is used throughout this document to define elements and structures.

CHAPTER 4

Contents

4.1 API Elements Overview

Version: 1.0.0-rc1

Mime Type: TBD

4.1.1 About API Elements

The purpose of API Elements is to provide a single, unifying structure for describing APIs across all API description formats and serialization formats. There currently exists several formats one can choose when defining an API, from API Blueprint to Swagger—which is now known as the OpenAPI Format. There are also many serialization formats such as XML or JSON. Without a way to parse these formats to the same structure, developers are required to handle each format one-by-one, each in a different way and each translating to their internal domain model. This is tedious, time-consuming, and requires each maintainer to stay in step with every format they support.

API Elements solves this complex problem in a simple way. It allows parsers to parse to a single structure and allows tool builders to consume one structure for all formats.

If there is one thing API description formats have taught us, it is that a single contract provides the best and fastest way to design and iterate on an API. Developers building the API can move independently as they progress towards the defined contract found in the API Blueprint or Swagger document. Conversely, API consumers can build tools for consuming the API based on the API definition document.

This same pattern has proven to be just as valuable for building API description formats and tooling. API Elements is the contract for producing and consuming the many description and serialization formats and allows everyone to move quickly and independently.

4.1.2 What is an Element?

API Elements is made up of many small elements that have a rich semantic meaning given their value and context. An element is an individual piece that makes up an API, and can range from defining a resource to providing an example of an HTTP request.

The API Elements project defines elements used for:

1. Describing an API
2. Describing data structures used within that API
3. Describing parse results when parsing API-related documents

These elements also seek to provide a way to decouple APIs and their semantics from the implementation details.

4.1.3 Relationship of Elements

One purpose of the API Elements reference is to allow consumers to decouple their implementations from the structure of the document. Because of this, when consuming documents of API Elements, it is recommended to write code that queries the tree rather than looking for defined paths.

It is also helpful to know the relationship between elements. The list below shows the relationship between the elements in this reference, but does not specify how the structure must be built.

- Category (API)
  - Copy
  - Data Structure
  - Category (Group of Resource Elements)
  - Category (Group of Authentication and Authorization Scheme Definitions)
  - Category (Group of Resource Elements representing hosts)
  - Resource
    - Copy
    - Data Structure
    - Category (Group of Transition Elements)
    - Transition
      - Copy
      - HTTP Transaction
      - Copy
      - HTTP Request
      - Copy
      - Data Structure
      - Asset
      - HTTP Response
      - Copy
      - Data Structure
      - Asset

This main API Category element MAY also be wrapped in a Parse Result element for conveying parsing information, such as source maps, warnings, and errors.

4.1.4 Basic Element

Every element defined with API Elements has four primary pieces of data.

- `element` (string) - defines the type of element used
- `meta` (object) - an object that includes metadata about the element
- `attributes` (object) - user-specified attributes for a given element
- `content` - value of the element based on its type

This structure defined in detail in the Element Definitions document.

Here is an example of what an Element MAY look like serialized into JSON.

```
{
  "element": "string",
  "meta": {
    "id": {
      "element": "string",
      "content": "foo"
    },
    "title": {
      "element": "string",
      "content": "Foo"
    },
    "description": {
      "element": "string",
      "content": "My foo element"
    }
  },
  "content": "bar"
}
```

Additional examples are provided throughout this documentation for specific API Elements. As this shows, though, it allows for API Elements to not only define data, but also metadata as well. This is especially important when providing source maps and adding human readable titles to values.

4.1.5 Consuming Documents

As mentioned, for consumers, it is important to not couple code to the specific structure of an API Elements document. The common pitfall is to reference elements by specifying a specific and strict path to those elements, but it is recommended to try to avoid this for sake of evolvability and safety.

For example, to get the first HTTP Transaction element from an API Elements tree.

Relying on a fixed tree structure:

```
const transaction = apielements.content[0].content[0].content[0].content[0].content[0]
```

Querying the tree in a way that does not use a strict path:

```
import query from 'refract-query';
const transaction = query(apielements, {element: 'httpTransaction'})[0];
```

The structure of an API Elements document is recursive by nature. When looking for specific elements, it is best to walk the API Elements tree to look for a match. Querying the API Elements tree will decouple your code from specific API description syntax. Also, it decouples your code from the specific structure of these documents as long as they are semantically equivalent.

4.2 Element Reference

4.2.1 Element

An *Element* SHALL be a tuple (`element`, `meta`, `attributes`, `content`) where

- `element` SHALL be a non-empty, finite character string identifying the *type* of this Element
- `meta` SHALL be a set of *properties*, some of which have *reserved semantics*
- `attributes` SHALL be a set of *properties* defined by the *type* of this Element
- `content` SHALL be defined by the *type* of this Element

Entries in `meta` SHOULD be independent of Element. Entries in `attributes` MAY be Element specific.

Property

A *property* SHALL be a tuple (`key`, `value`) where

- `key` SHALL be a non-empty, finite character string
- `value` SHALL be an *Element*
- Two properties SHALL be equal if their keys are.

  The last statement defining equality on properties through their keys allows definition of *objects* as *sets* of properties.

Values

Following values can be described in API Elements:

- *null* value
- boolean values *true* and *false*
- rational numbers, i.e. floating point numbers with finite precision
- finite character strings
- finite sets of properties
- finite lists of values

Types

Types specify value categories. Every Element SHALL describe a type.

Note that because types may be restricted to exactly one value, an Element MAY match only a single value; such an Element still represents a type, not the value itself.

An Element can therefore be thought of as a predicate that holds if, and only if, given value is in its value category. Given `Number` is the predicate classifying rational numbers, `Number(42.0)` SHALL hold, whereas `Number("foobar")` SHALL NOT.

Subtypes

We say the type *S* is a *subtype* of type *T* if, and only if, all values of *S* are also values of *T*.

API Elements predefines three broad categories of Element types:

1. *Data Structure Element types* - Tools to define types, e.g. *string*, *array*, *object*
2. *API Element types* - Types specific to API description
3. *Parse Result Element types* - Types specific to document parsing, e.g. source map, parse result

Reserved meta properties

Any of the following properties MAY be an entry of any Element's `meta`:

- `id` (*String*) - Unique name of this Element; defines a named type; MUST be unique with respect to other `ids` in a document
- `ref` (*Ref*) - Pointer to referenced element or type
- `classes` (*Array*[*String*]) - Classifications for given element
- `title` (*String*) - Human-readable title of element
- `description` (*String*) - Human-readable description of element
- `links` (*Array*[*Link Element*]) - Meta links for a given element

Examples

A primitive Element representing finite character strings is *String*, of type id `string`. Serialized into JSON, an Element representing `Hello world!` interpreted as a *String* value:

```
{
  "element": "string",
  "content": "Hello world!"
}
```

A less trivial example is the following `asset` Element. The specific semantic interpretation of an `asset` Element is well defined in the API Elements Reference section. What we essentially describe here is a JSON snippet `{"foo": "bar"}` defined in the message body of documentation.

```
{
  "element": "asset",
  "meta": {
    "classes": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "content": "messageBody"
        }
      ]
    }
  },
  "attributes": {
```

(continues on next page)

(continued from previous page)

```
    "contentType": {
      "element": "string",
      "content": "application/json"
    }
  },
  "content": "{\"foo\": \"bar\"}"
}
```

4.2.2 Data Structure Element types

Overview

API Elements and *Parse Result Elements* are all defined via Data Structure Elements. The following table summarizes them very broadly.

Fail Element

Type with empty domain. Attempts at instantiation of this Element SHALL fail.

Template

- `element` - `"fail"`

  Reserved for future use.

Null Element

Type with domain of a single value.

Template

- `element` - `"null"`

Example

The example below defines an Element representing only the null value.

```
{
  "element": "null"
}
```

Boolean Element

Type with domain of two values: *true* and *false*.

Template

- `element` - `"boolean"`
- `attributes`
  - `typeAttributes` (*Array*[*String*])
    * `fixed` (*String*) - The type this Element describes is restricted to the value given in `content`
  - `validation` - *reserved for future use*
  - `samples` (*Array*[*Boolean*]]) - Alternative sample values for this Element; type of items in `samples` MUST match the type this Element describes
  - `default` (*Boolean*) - Default value for this Element; type of `default` MUST match the type this Element describes
- `content` - *false* or *true*

Example

Type Element representing only boolean values (JSON `true`, `false`):

```
{
  "element": "boolean"
}
```

Type Element representing only boolean "true" (JSON `true`):

```
{
  "element": "boolean",
  "attributes": {
    "typeAttributes": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "content": "fixed"
        }
      ]
    }
  },
  "content": true
}
```

Number Element

Type with domain of all rational numbers, i.e. floating-point numbers with finite precision.

Template

- element - "number"
- attributes
  - typeAttributes (*Array*[*String*])
    - fixed (*String*) - The type this Element describes is restricted to the value given in content
  - validation - *reserved for future use*
  - samples (*Array*[*Number*]]) - Alternative sample values for this Element; type of items in samples MUST match the type this Element describes
  - default (*Number*) - Default value for this Element; type of default MUST match the type this Element describes
- content - Rational number

Example

Type Element representing only rationals. Matches JSON number values.

```
{
  "element": "number"
}
```

Type Element representing only the number 42:

```
{
  "element": "number",
  "attributes": {
    "typeAttributes": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "content": "fixed"
        }
      ]
    }
  },
  "content": 42
}
```

String Element

Type with domain of all finite character strings.

Template

- element - "string"
- attributes

  - `typeAttributes` (*Array*[*String*])
    - `fixed` (*String*) - The type this Element describes is restricted to the value given in `content`.
  - `validation` - *reserved for future use*
  - `samples` (*Array*[*String*]]) - Alternative sample values for this Element; type of items in `samples` MUST match the type this Element describes
  - `default` (*String*) - Default value for this Element; type of `default` MUST match the type this Element describes
- `content` - Finite character string

Example

Type Element representing only finite character strings. Matches JSON string values.

```
{
  "element": "string"
}
```

Type Element representing only the character string `"rocket science"`.

```
{
  "element": "string",
  "attributes": {
    "typeAttributes": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "content": "fixed"
        }
      ]
    }
  },
  "content": "rocket science"
}
```

Array Element

Type with domain of all finite lists of values.

Template

- `element` - `"array"`
- `attributes`
  - `typeAttributes` (*Array*[*String*])
    - `fixed` (*String*) - Restricts domain to a positionally typed fixed-length list over types in content. Further applies the `fixed` type attribute to nested *Array*s, *Object*s and any other type defining content or default.

    * `fixedType` (*String*) - Restricts domain to a list of types given in `content`.

  - `validation` - *reserved for future use*
  - `samples` (*Array*[*Array*]]) - Alternative sample values for this Element; type of items in `samples` MUST match the type this Element describes
  - `default` (*Array*) - Default value for this Element; type of `default` MUST match the type this Element describes

- `content` - Finite list of Elements

Examples

Type Element representing only lists (JSON `array`).

```
{
  "element": "array"
}
```

Type Element representing only pairs of (`string`, `number`).

```
{
  "element": "array",
  "attributes": {
    "typeAttributes": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "content": "fixed"
        }
      ]
    }
  },
  "content": [
    {
      "element": "string"
    },
    {
      "element": "number"
    }
  ]
}
```

Type Element representing only lists where items are either a JSON `string` or a JSON `number`.

```
{
  "element": "array",
  "attributes": {
    "typeAttributes": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "content": "fixedType"
        }
      ]
```

(continues on next page)

(continued from previous page)

```
    }
  },
  "content": [
    {
      "element": "string"
    },
    {
      "element": "number"
    }
  ]
}
```

Member Element

Type with domain of all properties.

Template

- `element` - `"member"`
- `attributes`
  - `typeAttributes` (*Array*[*String*])
    * `required` (*String*) - Property MUST be present in value represented by the containing *Object*. I.e. restricts the domain of the containing Object Element type to one containing this property.
    * `optional` (*String*) - Property MAY NOT be present in value represented by the containing *Object*. I.e. expands the domain of the containing Object Element type to one not containing this property.
  - `variable` - (*Boolean*) - Property key SHALL be interpreted as a variable name instead of a literal name
  - `validation` - *reserved for future use*
- `content`
  - `key` - An Element representing a key; MUST be set; SHOULD be a *String*
  - `value` - An Element representing the value

Examples

See *Object* for examples.

Object Element

Type with domain of all finite sets of properties.

Template

- `element` - `"object"`
- `attributes`
  - `typeAttributes` (*Array*[*String*])
    * `fixed` (*String*) - Restricts domain to a fixed sized set of properties, making them implicitly required. Further applies the `fixed` type attribute to nested *Array*s, *Object*s and any other type defining content or default.
    * `fixedType` (*String*) - Restricts domain to a fixed sized set of properties, making them implicitly required.
  - `validation` - *reserved for future use*
  - `samples` (*Array*[*Object*]]) - Alternative sample values for this Element; type of items in `samples` MUST match the type this Element describes
  - `default` (*Object*) - Default value for this Element; type of `default` MUST match the type this Element describes
- `content` - List of any of
  - *Member* - Object property
  - *Extend* - MUST type a property
  - *Select* - Contained *Option Element*s MUST type properties
  - *Ref* - MUST reference an Object Element

References in the `content` of an Object Element SHALL be semantically equivalent to their substitution by items held in the content of the referenced Object Element. Less formally, Ref Elements in the content of Object Element represent in-place mixins.

Examples

Type Element representing only property lists (JSON `object`).

```
{
  "element": "object"
}
```

Type Element representing only a specific property list instance (JSON `{"foo": false, "bar": "fun"}`).

```
{
  "element": "object",
  "attributes": {
    "typeAttributes": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "content": "fixed"
        }
      ]
    }
  },
```

(continues on next page)

(continued from previous page)

```
  "content": [
    {
      "element": "member",
      "content": {
        "key": {
          "element": "string",
          "content": "foo"
        },
        "value": {
          "element": "boolean",
          "content": false
        }
      }
    },
    {
      "element": "member",
      "content": {
        "key": {
          "element": "string",
          "content": "foo"
        },
        "value": {
          "element": "string",
          "content": "fun"
        }
      }
    }
  ]
}
```

Type Element representing only a property list with key "foo" of value type `boolean` and with the key "bar" of value type `string` (JSON `{"foo": false, "bar": "fun"}`, `{"foo": true, "bar": ""}` etc.).

```
{
  "element": "object",
  "attributes": {
    "typeAttributes": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "content": "fixedType"
        }
      ]
    }
  },
  "content": [
    {
      "element": "member",
      "content": {
        "key": {
          "element": "string",
          "content": "foo"
        },
        "value": {
          "element": "boolean",
```

(continues on next page)

(continued from previous page)

```
          "content": false
        }
      }
    },
    {
      "element": "member",
      "content": {
        "key": {
          "element": "string",
          "content": "bar"
        },
        "value": {
          "element": "string",
          "content": "fun"
        }
      }
    }
  ]
}
```

Enum Element

Type with domain of the union of values typed by Elements in the `enumerations` attribute. Also called tagged union or Σ-type.

Template

- `element` - `"enum"`
- `attributes`
  - `enumerations` (*Array*) - List of Elements
  - `typeAttributes` (*Array*[*String*])
    * `fixed` (*String*) - Elements in `enumerations` SHALL be interpreted `fixed`.
  - `validation` - *reserved for future use*
  - `samples` (*Array*[*Element*]]) - Alternative sample values for this Element; type of items in `samples` MUST match the type this Element describes
  - `default` (*Element*) - Default value for this Element; type of `default` MUST match the type this Element describes
- `content` - An Element matching one of the Elements in the `enumerations` attribute

Examples

Type Element representing strings and numbers.

```
{
  "element": "enum",
  "attributes": {
    "enumerations": {
      "element": "array",
      "content": [
        {
          "element": "string"
        },
        {
          "element": "number"
        }
      ]
    }
  }
}
```

Type Element representing a specific string and all numbers.

```
{
  "element": "enum",
  "attributes": {
    "enumerations": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "attributes": {
            "typeAttributes": {
              "element": "array",
              "content": [
                {
                  "element": "string",
                  "content": "fixed"
                }
              ]
            }
          },
          "content": "Hello world!"
        },
        {
          "element": "number"
        }
      ]
    }
  }
}
```

Select Element

Type with domain of the union of values typed by *Option Elements* in `content`. Select Element SHOULD only be used to denote alternative sets of properties in an [Object Element][].

Template

- `element` - `"select"`
- `attributes`
- `content` - Finite list of *Option Elements*

Option Element

Type with the domain of non-empty finite sets of *properties*. An Option Element MUST be contained in a *Select Element* and its items MUST type *properties*.

Template

- `element` - `"option"`
- `attributes`
- `content` - Non-empty list of *Elements* typing properties

Extend Element

Type with domain of *merged* Elements specified in `content`. All entries in `content` MUST type the same data structure type. *Ref Elements* encountered in `content` are dereferenced before merging.

Merging SHALL be defined based on the type of entries in `content` as follows:

- *Array* - List concatenation
- *Object* - Set union; if duplicit property keys are encountered during merging, all but the last SHALL be discarded; tooling SHOULD emit a warning in such a case.
- *Select Element* - Option concatenation
- *String* - Last entry in Extend Element SHALL be used, previous are ignored
- *Boolean* - Last entry in Extend Element SHALL be used, previous are ignored
- *Number* - Last entry in Extend Element SHALL be used, previous are ignored
- *Ref Element* - Substitute by referenced Element and apply one of the rules above

Extend Element SHOULD NOT be used to encode semantic inheritance; use the `id` meta property to define a named type and reference it through the child's `element` entry.

Template

- `element` - `"extend"`
- `content` - List of *Data Structure Elements* to be merged

Ref Element

Ref Element MAY be used to reference elements in remote documents or elements in the local document. The `ref` element *transcludes* the contents of the element into the document in which it is referenced.

The following rules apply:

1. When referencing an element in the local document, the `id` of the element MAY be used
2. When referencing remote elements, an absolute URL or relative URL MAY be used
3. When a URL fragment exists in the URL given, it references the element with the matching `id` in the given document. The URL fragment MAY need to be URL decoded before making a match.
4. When a URL fragment does not exist, the URL references the root element
5. When `path` is used, it references the given property of the referenced element
6. When `path` is used in an element that includes the data of the pointer (such as with `ref`), the referenced path MAY need to be converted to a refract structure in order to be valid

Transclusion of a Ref Element SHALL be defined as follows:

1. If the Ref Element is held by an *Array* Element and references an Array Element, its content entries SHALL be inserted in place of the Ref Element.
2. Else, if the Ref Element is held by an *Object* Element and references an Object Element, its content entries SHALL be inserted in place of the Ref Element.
3. Otherwise, the Ref Element is substituted by the Element it references.

Template

- `element` - `"ref"`
- `attributes`
  - `path` (enum[*String*]) - Path of referenced element to transclude instead of element itself
    - element (default) - The complete referenced element
    - meta - The meta data of the referenced element
    - attributes - The attributes of the referenced element
    - content - The content of the referenced element
  - `validation` - *reserved for future use*
- `content` - URL to an Element in this document as a string

Examples

Elements MAY be referenced in remote or local documents.

Referencing Remote Element

```
{
  "element": "ref",
  "content": "http://example.com/document#foo"
}
```

Referencing Local Elements

```
{
  "element": "ref",
  "content": "foo"
}
```

Reference Parts of Elements

Given an element instance of:

```
{
  "element": "array",
  "meta": {
    "id": {
      "element": "string",
      "content": "colors"
    }
  },
  "content": [
    {
      "element": "string",
      "content": "red"
    },
    {
      "element": "string",
      "content": "green"
    }
  ]
}
```

And given an array where a reference is used as:

```
{
  "element": "array",
  "content": [
    {
      "element": "string",
      "content": "blue"
    },
    {
      "element": "ref",
      "attributes": {
        "path": {
          "element": "string",
          "content": "content"
        }
      },
```

(continues on next page)

(continued from previous page)

```
      "content": "colors"
    }
  ]
}
```

The resulting dereferenced array is:

```
{
  "element": "array",
  "content": [
    {
      "element": "string",
      "content": "blue"
    },
    {
      "element": "string",
      "content": "red"
    },
    {
      "element": "string",
      "content": "green"
    }
  ]
}
```

Link Element

Hyperlinking MAY be used to link to other resources, provide links to instructions on how to process a given element (by way of a *profile* or other means), and may be used to provide meta data about the element in which it's found. The meaning and purpose of the hyperlink is defined by the link relation according to RFC 5988.

Template

- `element`: `"link"`
- `attributes`
  - `relation` (*String*) - Link relation type as specified in RFC 5988.
  - `href` (*String*) - The URI for the given link
  - `validation` - *reserved for future use*

Example

The following shows a link with the relation of `foo` and the URL of `/bar`.

```
{
  "element": "link",
  "attributes": {
    "relation":  {
      "element": "string",
      "content": "foo"
```

(continues on next page)

(continued from previous page)

```
    },
    "href": {
      "element": "string",
      "content": "/bar"
    }
  }
}
```

4.2.3 API Element Types

Href (String)

Subtype of *String* with domain of all URI-References per RFC 3986.

Template

- `element` - `"href"`

Templated Href (String)

Subtype of *String* with domain of all URI-Templates per RFC 6570.

Template

- `element` - `"templatedHref"`

Href Variables (Object)

Subtype of *Object* representing an object where each property's key is a `varname` (Commonly described as URI Template variable) per RFC 6570.

Template

- `element` - `"hrefVariables"`

Data Structure

Data structure definition using Data Structure elements.

Template

- `element` - `"dataStructure"`
- `content` - *Data Structure Element*

Asset (String)

Subtype of *String* with domain of all `message-body` as per RFC 2616.

Template

- `element` - `"asset"`
- `attributes`
  - `contentType` (*String*) - Optional media type of the asset. When this is unset, the content type SHOULD be inherited from the `Content-Type` header of a parent HTTP Message Payload
  - `href` (*Href*) - Link to the asset
- `content` - A textual representation of the asset

Classifications

- `"messageBody"` - Asset is an example of message-body
- `"messageBodySchema"` - Asset is a schema for message-body

Resource

The Resource representation with its available transitions and its data.

Template

- `element` - `"resource"`
- `attributes`
  - `hosts` (*Array*[*Resource*]).

    Optional list of host resources. Every entry SHALL be interpreted as if classified as `host`. *See* `host` *classification in* Resource *for further semantics.*

    Overrides any otherwise relevant `hosts` definitions.
  - `href` (*Templated Href*) - URI Template for this resource.
  - `hrefVariables` (*Href Variables*) - URI Template variables.
- `content` (array)
  - (*Copy*) - Textual information of this resource in API Description.
  - (*Category*) - A group of Transition elements.
  - (*Transition*) - State transitions available for this resource.

    The `content` MAY include multiple `Transition` elements.
  - (*Data Structure*) - Data structure representing the resource.

    The `content` MUST NOT include more than one `Data Structure`.

Classifications

- `"host"` - A host resource represents the "root" of the API resource. The resource href MAY be append to the host href to create a absolute URI. A resource that has a `host` classification MUST be a root component of a URI.

Example

```
{
  "element": "resource",
  "meta": {
    "title": {
      "element": "string",
      "content": "Question"
    },
    "description": {
      "element": "string",
      "content": "A Question object has the following attributes."
    }
  },
  "attributes": {
    "href": {
      "element": "string",
      "content": "/questions/{question_id}"
    },
    "hrefVariables": {
      "element": "hrefVariables",
      "content": [
        {
          "element": "member",
          "content": {
            "key": {
              "element": "string",
              "content": "question_id"
            }
          }
        }
      ]
    }
  },
  "content": [
    {
      "element": "dataStructure"
    }
  ]
}
```

Transition

A transition is an available progression from one state to another state. Exercising a transition initiates a transaction.

The content of this element is array of protocol-specific transactions.

Note: At the moment only the HTTP protocol is supported.

Template

- `element` - `"transition"`
- `attributes`
  - `contentTypes` (*Array*[*String*]) - A collection of content types that MAY be used for the transition.
  - `data` (*Data Structure*) - Data structure describing the transition's `Request message-body` unless overridden.

    Definition of any input message-body attribute for this transition.
  - `hosts` (*Array*[*Resource*]).

    Optional list of host resources. Every entry SHALL be interpreted as if classified as `host`.

    See `host` *classification in* Resource *for further semantics.*

    All *Resource*s nested under the *Transition*'s `content` SHALL interpret this `hosts` definition as their own, unless it is overridden by another `hosts` definition on the path to the *Resource* element.
  - `href` (*Templated Href*) - URI template for this transition.

    If present, the value of the `href` attribute SHOULD be used to resolve the target URI of the transition.

    If not set, the parent `resource` element `href` attribute SHOULD be used to resolve the target URI of the transition.
  - `hrefVariables` (*Href Variables*) - URI Template variables.

    Definition of any input URI path segments or URI query parameters for this transition.

    If `href` and `hrefVariables` attributes aren't set, the parent `resource` element `hrefVariables` SHOULD be used to resolve the transition input parameters.
  - `relation` - (*String*) - Link relation type as specified in RFC 5988.

    The value of `relation` attribute SHOULD be interpreted as a link relation between transition's parent resource and the transition's target resource as specified in the `href` attribute.
- `content` (array)
  - (*Copy*) - Textual information of this transition in API Description.
  - (*HTTP Transaction*)

    Transaction examples are protocol-specific examples of a REST transaction that was initialized by exercising a transition.

    For the time being this reference document defines only HTTP-specific transaction data structures.

Example

```
{
  "element": "transition",
  "attributes": {
    "relation": {
      "element": "string",
      "content": "update"
    },
    "href": {
      "element": "string",
```

(continues on next page)

(continued from previous page)

```
      "content": "https://polls.apiblueprint.org/questions/{question_id}"
    }
  },
  "content": []
}
```

API Metadata (Member)

Subtype of *Member* representing a property whose key and value are strings.

Classifications

- `"user"` - User-specific metadata. Metadata written in the source.
- `"adapter"` - Serialization-specific metadata. Metadata provided by adapter.

Category

Grouping element – a set of elements forming a logical unit of an API such as group of related resources or data structures.

A category element MAY include additional classification of the category. The classification MAY hint what is the content or semantics of the category. The classification MAY be extended and MAY contain more than one classes.

For example a `category` element may be classified both as `resourceGroup` and `dataStructures` to denote it includes both resource and data structures. It may also include the `transitions` classification to denote it includes transitions.

Template

- `element` - `"category"`
- `attributes` - is intended for place element specific info
  - `metadata` (*Array*[*API Metadata*]) - Arbitrary metadata
  - `version` (*String*) - reserved for API documentation version info, if presented MUST be placed on API Category (top-level group)
- `content` (array)

Classifications

- `"api"` - Category is a API top-level group.
- `"authSchemes"` - Category is a group of authentication and authorization scheme definitions.
- `"dataStructures"` - Category is a set of data structures.
- `"hosts"` - Category of [Resource][]s interpreted as a list of host resources of an API. Every entry SHALL be interpreted as if classified as `host`.

See `host` classification in [Resource][] for further semantics.

- "resourceGroup" - Category is a set of resources.
- "scenario" - Category is set of steps.
- "transitions" - Category is a group of transitions.

Example


```
{
  "element": "category",
  "meta": {
    "classes": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "content": "api"
        }
      ]
    },
    "title": {
      "element": "string",
      "content": "Polls API"
    }
  },
  "attributes": {
    "metadata": {
      "element": "array",
      "content": [
        {
          "element": "member",
          "meta": {
            "classes": {
              "element": "array",
              "content": [
                {
                  "element": "string",
                  "content": "user"
                }
              ]
            }
          },
          "content": {
            "key": {
              "element": "string",
              "content": "HOST"
            },
            "value": {
              "element": "string",
              "content": "http://polls.apiblueprint.org/"
            }
          }
        }
      ]
    }
  },
  "content": [
```


(continues on next page)

(continued from previous page)

```
    {
      "element": "category",
      "meta": {
        "classes": {
          "element": "array",
          "content": [
            {
              "element": "string",
              "content": "resourceGroup"
            }
          ]
        },
        "title": {
          "element": "string",
          "content": "Question"
        }
      },
      "content": [
        {
          "element": "copy",
          "content": "Resources related to questions in the API."
        }
      ]
    }
  ]
}
```

Copy (String)

Subtype of *String* which represents a textual information in API description.

Its content is a string and it MAY include information about the media type of the copy's content.

Unless specified otherwise, a copy element's content represents the description of its parent element and SHOULD be used instead of parent element's description metadata.

Template

- `element` - `"copy"`
- `attributes`
  - `contentType` (*String*) - Optional media type of the content.
- `content` - Text

Example

Given an API description with following layout:

- Group
  - Copy "Lorem Ipsum"
  - Resource "A"

– Resource "B"

– Copy "Dolor Sit Amet"

```
{
  "element": "category",
  "content": [
    {
      "element": "copy",
      "content": "Lorem Ipsum"
    },
    {
      "element": "resource"
    },
    {
      "element": "resource"
    },
    {
      "element": "copy",
      "content": "Dolor Sit Amet"
    }
  ]
}
```

Protocol-specific Elements

HTTP Transaction (Array)

Example of an HTTP Transaction.

Template

- `element` - `"httpTransaction"`
- `attributes`
  - `authSchemes` (*Array*) - An array of authentication and authorization schemes that apply to the transaction
- `content` (array) - Request and response message pair (tuple).
  - (*Copy*) - Textual information of this transaction in API Description.
  - (*HTTP Request Message*)

    The `content` MUST include exactly one `HTTP Request Message` element.
  - (*HTTP Response Message*)

    The `content` MUST include exactly one `HTTP Response Message` element.

Example

```
{
  "element": "httpTransaction",
  "content": [
```

(continues on next page)

(continued from previous page)


```
    {
      "element": "httpRequest",
      "attributes": {
        "method": {
          "element": "string",
          "content": "GET"
        },
        "href": {
          "element": "string",
          "content": "/questions/{question_id}"
        },
        "hrefVariables": {
          "element": "hrefVariables",
          "content": [
            {
              "element": "member",
              "content": {
                "key": {
                  "element": "string",
                  "content": "question_id"
                }
              }
            }
          ]
        }
      },
      "content": []
    },
    {
      "element": "httpResponse",
      "attributes": {
        "statusCode": {
          "element": "number",
          "content": 200
        }
      },
      "content": [
        {
          "element": "asset",
          "meta": {
            "classes": {
              "element": "array",
              "content": [
                {
                  "element": "string",
                  "content": "messageBody"
                }
              ]
            }
          },
          "attributes": {
            "contentType": {
              "element": "string",
              "content": "application/json"
            }
          },
          "content": "{\"name\": \"John\"}"
```


(continues on next page)

(continued from previous page)

```
          }
        ]
      }
    ]
  }
```

HTTP Headers (Object)

Subtype of *Object* representing an object where each property's key is a `field-name` (Commonly known as HTTP Header name) per RFC 822 and the property's key is a `field-value` (Commonly known as HTTP Header value) per RFC 2616.

Template

- `element` - `"httpHeaders"`

Example

```
{
  "element": "httpHeaders",
  "content": [
    {
      "element": "member",
      "content": {
        "key": {
          "element": "string",
          "content": "Content-Type"
        },
        "value": {
          "element": "string",
          "content": "application/json"
        }
      }
    }
  ]
}
```

HTTP Message Payload (Array)

Subtype of *Array* representing a `HTTP-message` (Commonly known as Payload) per RFC 2616.

Template

- `attributes`
  - `headers` (*HTTP Headers*)
- `content` (array)
  - (*Copy*) - Textual information of this payload in API Description.

- (*Data Structure*) - Data structure describing the payload's `message-body`.

  The `content` MUST NOT contain more than one `Data Structure`.

- (*Asset*) - An asset associated with the payload's `message-body`.

  This asset MAY represent payload body or body's schema.

  The `content` SHOULD NOT contain more than one asset of its respective type.

HTTP Request Message (HTTP Message Payload)

Subtype of *HTTP Message Payload* representing a `Request` (Commonly known as HTTP Request) per RFC 2616.

Template

- `element` - `"httpRequest"`
- `attributes`
  - `method` (*String*) - `Method` part of HTTP Request per RFC 2616.

    The method value SHOULD be inherited from a parent transition if it is unset.

  - `href` (*Templated Href*) - URI Template for this HTTP request. Combined with `hrefVariables` forms the `Request-URI` part of HTTP Request per RFC 2616.

    If present, the value of the `href` attribute SHOULD be used to resolve the target URI of the http request.

    If not set, the `href` attribute which was used to resolve the target URI of the parent transition SHOULD be used to resolve the URI of the http request.

  - `hrefVariables` (*Href Variables*) - URI Template variables.

    Definition of any input URI path segments or URI query parameters for this transition.

    If `href` and `hrefVariables` attributes aren't set, the `hrefVariables` attribute which was used to resolve the input parameters of the parent transition SHOULD be used to resolve the http request input parameters.

HTTP Response Message (HTTP Message Payload)

Subtype of *HTTP Message Payload* representing a `Response` (Commonly known as HTTP Response) per RFC 2616.

Template

- `element` - `"httpResponse"`
- `attributes`
  - `statusCode` (*Number*) - `Status-Code` part of HTTP Response per RFC 2616.

4.2.4 Parse Result Element types

Parse Result (Array)

A result of parsing of an API description document.

Template

- element - "parseResult"
- content (array)
  - (*Category*)
  - (*Annotation*)

Example

Given following API Blueprint document:

```
# GET /1
```

The parse result is (using null in category content for simplicity):


```
{
  "element": "parseResult",
  "content": [
    {
      "element": "category",
      "meta": {
        "classes": {
          "element": "array",
          "content": [
            {
              "element": "string",
              "content": "api"
            }
          ]
        }
      }
    },
    {
      "element": "annotation",
      "meta": {
        "classes": {
          "element": "array",
          "content": [
            {
              "element": "string",
              "content": "warning"
            }
          ]
        }
      },
      "attributes": {
        "code": {
          "element": "number",
          "content": 6
        },
        "sourceMap": {
          "element": "array",
          "content": [
            {
```


(continues on next page)

(continued from previous page)

```
              "element": "sourceMap",
              "content": [
                {
                  "element": "array",
                  "content": [
                    {
                      "element": "number",
                      "content": 0
                    },
                    {
                      "element": "number",
                      "content": 9
                    }
                  ]
                }
              ]
            }
          ]
        }
      },
      "content": "action"
    }
  ]
}
```

Annotation (String)

Annotation for a source file. Usually generated by a parser or adapter.

Template

- `element` - `"annotation"`
- `attributes`
  - `code` (*Number*) - Parser-specific code of the annotation. Refer to parser documentation for explanation of the codes.
  - `sourceMap` (*Array*[*Source Map*]) - Locations of the annotation in the source file.
- `content` - Textual annotation.

  This is – in most cases – a human-readable message to be displayed to user.

Classifications

- `"error"` - Annotation is an error
- `"warning"` - Annotation is a warning

Example

```
{
  "element": "annotation",
  "meta": {
    "classes": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "content": "warning"
        }
      ]
    }
  },
  "attributes": {
    "code": {
      "element": "number",
      "content": 6
    },
    "sourceMap": {
      "element": "array",
      "content": [
        {
          "element": "sourceMap",
          "content": [
            {
              "element": "array",
              "content": [
                {
                  "element": "number",
                  "content": 4
                },
                {
                  "element": "number",
                  "content": 12
                }
              ]
            },
            {
              "element": "array",
              "content": [
                {
                  "element": "number",
                  "content": 20
                },
                {
                  "element": "number",
                  "content": 12
                }
              ]
            }
          ]
        }
      ]
    }
  },
```

(continues on next page)

(continued from previous page)

```
  "content": "action is missing a response"
}
```

Source Map

Source map of an Element.

Every Element MAY include a `sourceMap` attribute. Its content MUST be an array of `Source Map` elements. The Source Map elements represent the location(s) in source file(s) from which the element was composed.

If used, it represents the location of bytes in the source file. This location SHOULD include the bytes used to build the parent element.

The Source Map element MUST NOT be used in its normal form unless the particular application clearly implies what is the source file the source map is pointing in.

A source map is a series of byte-blocks. These blocks may be non-continuous. For example, a block in the series may not start immediately after the previous block. Each block, however, is a continuous series of bytes.

Template

- `element` - `"sourceMap"`
- `content` (array) - Array of byte blocks.
  - (*Array*) - Continuous bytes block. A pair of byte index and byte count.
    - `content` (array)
      - (*Number*) - Zero-based index of a byte in the source document.
      - attributes
      - line (*Number*) - The line number the source map starts on.
      - column (*Number*) - The column number of the line that the source map starts on.
      - (*Number*) - Count of bytes starting from the byte index.
      - attributes
      - line (*Number*) - The line number the source map ends on.
      - column (*Number*) - The column number of the line that the source map ends on.

Example

```
{
  "element": "sourceMap",
  "content": [
    {
      "element": "array",
      "content": [
        {
          "element": "number",
          "content": 4
        },
```

(continues on next page)

(continued from previous page)

```
        {
          "element": "number",
          "content": 12
        }
      ]
    },
    {
      "element": "array",
      "content": [
        {
          "element": "number",
          "content": 4
        },
        {
          "element": "number",
          "content": 12
        }
      ]
    }
  ]
}
```

This reads, “The location starts at the 5th byte of the source file. It includes the 12 subsequent bytes including the starting one. Then it continues at the 21st byte for another 12 bytes.”

Example

```
{
  "element": "sourceMap",
  "content": [
    {
      "element": "array",
      "content": [
        {
          "element": "number",
          "attributes": {
            "line": {
              "element": "number",
              "content": 3
            },
            "column": {
              "element": "number",
              "content": 2
            }
          },
          "content": 4
        },
        {
          "element": "number",
          "attributes": {
            "line": {
              "element": "number",
              "content": 3
            },
            "column": {
```

(continues on next page)

(continued from previous page)

```
                "element": "number",
                "content": 10
              }
            },
            "content": 12
          }
        ]
      }
    ]
}
```

This reads, "The location starts at the 5th byte (the 2nd byte of line 3) of the source file. It includes 12 bytes, until column 10 on line 3".

NOTE *`line` and `column` are optional and may not always be available*.

Link Relations

In addition to conforming to RFC 5988 for link relations, there are also additional link relations available for parse results.

Origin Link Relation

The `origin` link relation defines the origin of a given element. This link can point to specific tooling that was used to parse or generate a given element.

Inferred Link Relation

The `inferred` link relation gives a hint to whether or not an element was inferred or whether it was found in the originating document. The presence of the `inferred` link tells the user that the element was created based on some varying assumptions, and the URL to which the link points MAY provide an explanation on how and why it was inferred.

4.2.5 Authentication and Authorization Schemes

Authentication and authorization schemes MAY be defined within an API Elements document. These schemes are then used within the context of a resource to define which schemes to apply when making a transaction.

Basic Authentication Scheme (Object)

This element may be used to define a basic authentication scheme implementation for an API as described in RFC 2617.

The element MAY have a username and password defined as member elements within the scheme, but these are not required.

- `username` (string, optional)
- `password` (string, optional)

Template

- `element` - `"Basic Authentication Scheme"`

Example

This example shows a custom basic authentication scheme being defined as `Custom Basic Auth`. This scheme is then used on an HTTP transaction within a resource. Please note this example is incomplete for the sake of keeping it short.


```
{
  "element": "category",
  "meta": {
    "classes": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "content": "api"
        }
      ]
    }
  },
  "content": [
    {
      "element": "category",
      "meta": {
        "classes": {
          "element": "array",
          "content": [
            {
              "element": "string",
              "content": "authSchemes"
            }
          ]
        }
      },
      "content": [
        {
          "element": "Basic Authentication Scheme",
          "meta": {
            "id": {
              "element": "string",
              "content": "Custom Basic Auth"
            }
          },
          "content": [
            {
              "element": "member",
              "content": {
                "key": {
                  "element": "string",
                  "content": "username"
                },
                "value": {
                  "element": "string",
```


(continues on next page)

(continued from previous page)


```
                  "content": "john.doe"
                }
              }
            },
            {
              "element": "member",
              "content": {
                "key": {
                  "element": "string",
                  "content": "password"
                },
                "value": {
                  "element": "string",
                  "content": "1234password"
                }
              }
            }
          ]
        }
      ]
    },
    {
      "element": "resource",
      "attributes": {
        "href": {
          "element": "string",
          "content": "/users"
        }
      },
      "content": [
        {
          "element": "transition",
          "content": [
            {
              "element": "httpTransaction",
              "attributes": {
                "authSchemes": {
                  "element": "array",
                  "content": [
                    {
                      "element": "Custom Basic Auth"
                    }
                  ]
                }
              }
            }
          ]
        }
      ]
    }
  ]
}
```

Token Authentication Scheme (Object)

This describes an authentication scheme that uses a token as a way to authenticate and authorize. The token MAY exist as an HTTP header field, query parameter or as a cookie.

- One of
  - `httpHeaderName` (string)
  - `queryParameterName` (string)
  - `cookieName` (string)

When used as a query parameter, an HREF Variable is not required to be defined within the scope of the resource or transition, but is rather infered from the used token authentications scheme.

Template

- `element` - `"Token Authentication Scheme"`

Example

This example shows a custom token authentication scheme being defined as `Custom Token Auth` that uses a query parameter for the token. This scheme is then used on an HTTP transaction within a resource. Please note this example is incomplete for the sake of keeping it short.

```
{
  "element": "category",
  "meta": {
    "classes": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "content": "api"
        }
      ]
    }
  },
  "content": [
    {
      "element": "category",
      "meta": {
        "classes": {
          "element": "array",
          "content": [
            {
              "element": "string",
              "content": "authSchemes"
            }
          ]
        }
      },
      "content": [
        {
          "element": "Token Authentication Scheme",
```

(continues on next page)

(continued from previous page)


```
          "meta": {
            "id": {
              "element": "string",
              "content": "Custom Token Auth"
            }
          },
          "content": [
            {
              "element": "member",
              "content": {
                "key": {
                  "element": "string",
                  "content": "queryParameterName"
                },
                "value": {
                  "element": "string",
                  "content": "api_key"
                }
              }
            }
          ]
        }
      ]
    },
    {
      "element": "resource",
      "attributes": {
        "href": {
          "element": "string",
          "content": "/users"
        }
      },
      "content": [
        {
          "element": "transition",
          "content": [
            {
              "element": "httpTransaction",
              "attributes": {
                "authSchemes": {
                  "element": "array",
                  "content": [
                    {
                      "element": "Custom Token Auth"
                    }
                  ]
                }
              }
            }
          ]
        }
      ]
    }
  ]
}
```

OAuth2 Scheme

This describes an authentication scheme that uses OAuth2 as defined in RFC 6749.

The element MAY have the following members to define additional information about the OAuth2 implementation.

- `scopes` (*Array*[*String*])
- `grantType` (enum[*String*])
  - `authorization code`
  - `implicit`
  - `resource owner password credentials`
  - `client credentials`

Transition elements are used to define the URLs for the authorize, token and refresh endpoints for the OAuth2 schemes. When including these endpoints, the following link relations SHOULD be used.

- `authorize` - URL for the authorization endpoint
- `token` - URL for the token endpoint
- `refresh` - URL for the refresh endpoint

The HREF values for these transitions MAY be either relative or absolute URLs.

Template

- `element` - `"OAuth2 Scheme"`
- `content` (array)
  - (*Member*)
  - (*Transition*)

Example

This example shows a custom OAuth2 scheme being defined as `Custom OAuth2`. This scheme is then used on an HTTP transaction within a resource. There are a couple of things to note about this example:

1. There are two scopes defined within the scheme, but only one is used within the context of the transaction.
2. Transitions are used to define the authorize and token endpoints.

Also, please note this example is incomplete for the sake of keeping it short.

```
{
  "element": "category",
  "meta": {
    "classes": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "content": "api"
        }
      ]
```

(continues on next page)

(continued from previous page)


```
    }
  },
  "content": [
    {
      "element": "category",
      "meta": {
        "classes": {
          "element": "array",
          "content": [
            {
              "element": "string",
              "content": "authSchemes"
            }
          ]
        }
      },
      "content": [
        {
          "element": "OAuth2 Scheme",
          "meta": {
            "id": {
              "element": "string",
              "content": "Custom OAuth2"
            }
          },
          "content": [
            {
              "element": "member",
              "content": {
                "key": {
                  "element": "string",
                  "content": "scopes"
                },
                "value": {
                  "element": "array",
                  "content": [
                    {
                      "element": "string",
                      "content": "scope1"
                    },
                    {
                      "element": "string",
                      "content": "scope2"
                    }
                  ]
                }
              }
            },
            {
              "element": "member",
              "content": {
                "key": {
                  "element": "string",
                  "content": "grantType"
                },
                "value": {
                  "element": "string",
```


(continues on next page)

(continued from previous page)

```
                  "content": "implicit"
                }
              }
            },
            {
              "element": "transition",
              "attributes": {
                "relation": {
                  "element": "string",
                  "content": "authorize"
                },
                "href": {
                  "element": "string",
                  "content": "/authorize"
                }
              }
            },
            {
              "element": "transition",
              "attributes": {
                "relation": {
                  "element": "string",
                  "content": "token"
                },
                "href": {
                  "element": "string",
                  "content": "/token"
                }
              }
            }
          ]
        }
      ]
    },
    {
      "element": "resource",
      "attributes": {
        "href": {
          "element": "string",
          "content": "/users"
        }
      },
      "content": [
        {
          "element": "transition",
          "content": [
            {
              "element": "httpTransaction",
              "attributes": {
                "authSchemes": {
                  "element": "array",
                  "content": [
                    {
                      "element": "Custom OAuth2",
                      "content": [
                        {
                          "element": "member",
```

(continues on next page)

(continued from previous page)

```
                          "content": {
                            "key": {
                              "element": "string",
                              "content": "scopes"
                            },
                            "value": {
                              "element": "array",
                              "content": [
                                {
                                  "element": "string",
                                  "content": "scope1"
                                }
                              ]
                            }
                          }
                        }
                      ]
                    }
                  ]
                }
              }
            }
          ]
        }
      ]
    }
  ]
}
```

4.2.6 Profiles

The primary means by which users can provide semantic definitions and other meta information is through a profile. A profile MAY provide semantic information about an element and its data, it MAY provide specific instructions about elements such as how inheritance should work or how elements should be processed, and it MAY be used to modify understanding of existing elements in other profiles. The usage of a profile is not limited to these uses here, and SHOULD be left up to the profile author to define its use.

To point to a profile, you MAY use the profile link relation as a meta link in your root element or in any other element. Profile links may also be found outside of the document itself in places like the HTTP Link Header. Additionally, a profile link is not required in order to use the functionality a profile provides, as a media type MAY define the same things a profile.

Below is an example of how a profile link is used as a meta link.

```
{
  "element": "foo",
  "meta": {
    "links": {
      "element": "array",
      "content": [
        {
          "element": "link",
          "attributes": {
```

(continues on next page)

(continued from previous page)

```
            "relation": {
              "element": "string",
              "content": "profile"
            },
            "href": {
              "element": "string",
              "content": "http://example.com/profiles/foo"
            }
          }
        }
      ]
    }
  },
  "content": "bar"
}
```

The example shows a `foo` element with a `profile` link. This profile link informs the parser this particular element is defined as part of the linked profile.

4.2.7 Extending API Elements

An API Elements document MAY be extended by providing a profile link that describes how non-specification elements should be handled.

Additionally, an `extension` element is provided as a way to extend API Elements documents to include additional data not expressed by the elements in this specification.

When the `extension` element is used, it SHOULD include a profile link that provides information on how the content and attributes SHOULD be handled. Additionally, the presence of an `extension` element MUST NOT change the meaning of the rest of the API Elements document in which it is found. In other words, a tool SHOULD be able to safely ignore an `extension` element.

For changes that need to make unsafe changes, a custom media type or profile SHOULD be used.

Extension

- `element` - `"extension"`
- `content` (enum) - Custom content of extension element
  - (*String*)
  - (*Number*)
  - (*Boolean*)
  - (*Array*)
  - (*Object*)

Example

This `extension` element has a custom content, and the meaning and handling instructions for this content.

```json
{
  "element": "extension",
  "meta": {
    "links": {
      "element": "array",
      "content": [
        {
          "element": "link",
          "attributes": {
            "relation": {
              "element": "string",
              "content": "profile"
            },
            "href": {
              "element": "string",
              "content": "http://example.com/extensions/info/"
            }
          }
        }
      ]
    }
  },
  "content": {
    "element": "object",
    "content": [
      {
        "element": "member",
        "content": {
          "key": {
            "element": "string",
            "content": "version"
          },
          "value": {
            "element": "string",
            "content": "1.0"
          }
        }
      }
    ]
  }
}
```


This specific extension adds an object for including information about an API that may be specific to an implementation—in this case, a version number of the API. The URL `http://example.com/extensions/info/` would then provide instructions on the meaning and structure of the `content`.

As a tool comes across this extension element, it would look at the profile URL to see if it understands this particular element. If not, it can ignore it safely, but if so, it can use it as it sees fit.

4.3 Examples

4.3.1 API Description Formats

API Blueprint Example

This shows an example of the parse results of an API Blueprint. Here is a very minimal example API Blueprint.

```
# My API
## Foo [/foo]
```

When this document is parsed, it returns a serialization of API Elements that looks like the following.


```
{
  "element": "parseResult",
  "content": [
    {
      "element": "category",
      "meta": {
        "classes": {
          "element": "array",
          "content": [
            {
              "element": "string",
              "content": "api"
            }
          ]
        },
        "title": {
          "element": "string",
          "content": "My API"
        }
      },
      "content": [
        {
          "element": "category",
          "meta": {
            "classes": {
              "element": "array",
              "content": [
                {
                  "element": "string",
                  "content": "resourceGroup"
                }
              ]
            },
            "title": {
              "element": "string",
              "content": ""
            }
          },
          "content": [
            {
              "element": "resource",
              "meta": {
                "title": {
                  "element": "string",
                  "content": "Foo"
                }
              },
              "attributes": {
                "href": {
```


(continues on next page)

(continued from previous page)

```
                  "element": "string",
                  "content": "/foo"
                }
              },
              "content": []
            }
          ]
        }
      ]
    }
  ]
}
```

Swagger / OpenAPI Format Example

This shows an example of the parse results of a Swagger document.

```
{
  "swagger": "2.0",
  "info": {
    "title": "My API",
    "version": "1.0.0"
  },
  "paths": {
    "/foo": {}
  }
}
```

When this document is parsed, it returns a serialization of API Elements that looks like:

```
{
  "element": "parseResult",
  "content": [
    {
      "element": "category",
      "meta": {
        "classes": {
          "element": "array",
          "content": [
            {
              "element": "string",
              "content": "api"
            }
          ]
        },
        "title": {
          "element": "string",
          "content": "My API"
        }
      },
      "content": [
        {
          "element": "resource",
          "attributes": {
            "href": {
```

(continues on next page)

(continued from previous page)

```
                "element": "string",
                "content": "/foo"
              }
            },
            "content": []
          }
        ]
      }
    ]
}
```

As you can see, Swagger and API Blueprint both convey the same information resulting in almost the same parse results. This shows the importance of querying the results rather than looking for specific paths in the document.

4.3.2 Data Structure

This section shows what individual and specific data structures look like when converted to API Elements. In the context of an API description or parse results, these structures will be nested within the document.

If you have an MSON definition like the one below.

```
# My List (array)
- 1 (number)
- 2 (number)
- 3 (number)
```

When the structure is parsed, it returns a serialization of API Elements that looks like:

```
{
  "element": "array",
  "meta": {
    "id": {
      "element": "string",
      "content": "My List"
    }
  },
  "content": [
    {
      "element": "number",
      "content": 1
    },
    {
      "element": "number",
      "content": 2
    },
    {
      "element": "number",
      "content": 3
    }
  ]
}
```

And when it is converted to JSON, it looks like:

```
[1, 2, 3]
```

4.4 Tooling

There are various API Elements tools available to interact and parse API Description Documents.

4.4.1 API Description Parsing Service

API Description Parsing Serice (formerly API Bluerpint API) is a hosted service that takes API Description documents such as API Blueprint or Swagger 2.0 as input and returns API Elements.

4.4.2 JavaScript

Fury

Fury is a library for validating and parsing API description documents, Furys API provides API Element JS objects.

API Element JS

The API Elements JS Package provides an interface for querying and interacting with API Elements. This library can be used in conjunction with Fury to handle parsing of API Description documents into API Elements.

4.4.3 Python

refract.py

A Python library for interacting with Refract and API Elements in Python.

4.4.4 API Blueprint

The API Blueprint ecosystem heavily uses API Elements under the hood. Although we would recommend interacting with API Elements using the JavaScript tooling above as it is generic and not API Blueprint specific.

Drafter

Drafter is a library for parsing API Blueprint documents and return parse results in API Elements.

Drafter JS

Drafter JS is a JavaScript interface to Drafter and can be used in Node.JS or natively in a browser.

4.5 Additional Information

This page lists links and references for additional reading.

- API Blueprint - API description format that utilizes API Elements as its parsing results
- MSON - Data structure format that utilizes API Elements as its parsing results (see the Data Structure Elements)

4.6 Extensions Registry

4.6.1 OpenAPI Specification Extensions

OpenAPI Specification Extensions

This document is a profile for storing OpenAPI Specification Extensions within an API Element extension.

The contents of an extension element with this profile will contain the contents of the extensions is an object containing any vendor extensions found in the underlying Swagger Description document.

For example, if a vendor extension with the key `x-sts` with a value true was found in an OpenAPI document, it may be represented using the following extension element:

```
{
    "element": "extension",
    "meta": {
        "links": [
            {
                "element": "link",
                "attributes": {
                  "relation": "profile",
                  "href": "https://apielements.org/extensions/oas/extensions/"
                }
            }
        ]
    },
    "content": {
        "element": "object",
        "content": [
            {
                "element": "member",
                "content": {
                    "key": {
                        "element": "string",
                        "content": "sts"
                    },
                    "value": {
                        "element": "boolean",
                        "content": true
                    }
                }
            }
        ]
    }
}
```

4.7 Migration Guide

This guide documents all of the changes between API Elements 0.6 and API Elements 1.0.

4.7.1 JSON Serialisation

In prior versions of API Elements, an Element (i.e. a type) could be occasionally serialized as a value. This behaviour has since been dropped, so that Elements are always serialized in full form (i.e. as a type, not as a value). For example:

```
{
  "element": "null",
  "meta": {
    "title": "empty"
  }
}
```

Elements must always be serialised as an element, for example the following would be valid in API Elements 1.0:

```
{
  "element": "null",
  "meta": {
    "title": {
      "element": "string",
      "content": "empty"
    }
  }
}
```

In previous versions of API Elements, both forms were valid so this is not a breaking change. However, we found multiple implementations that were fragile and could break when different forms were used.

4.7.2 Changes to Elements

Category Element

The `meta` attribute has been renamed to `metadata` in a Category Element for clarity.

Before

```
{
  "element": "category",
  "attributes": {
    "meta": {
      "element": "array",
      "content": [
        {
          "element": "member",
          "content": {
            "key": {
              "element": "string",
              "content": "HOST"
            },
            "value": {
              "element": "string",
              "content": "http://polls.apiblueprint.org/"
            }
          }
        }
      ]
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

After

```
{
  "element": "category",
  "attributes": {
    "metadata": {
      "element": "array",
      "content": [
        {
          "element": "member",
          "content": {
            "key": {
              "element": "string",
              "content": "HOST"
            },
            "value": {
              "element": "string",
              "content": "http://polls.apiblueprint.org/"
            }
          }
        }
      ]
    }
  }
}
```

Source Map Element

A Source Map Element may contain an optional line and column number to make it easier to handle source map information. Computing the line and column number can often be expensive so it may be provided by a parser. Note however that it is optional and it is down the each individual tooling on whether it is present, some tools only provide line and column number for source maps contained within Annotation Elements.

```
{
  "element": "sourceMap",
  "content": [
    {
      "element": "array",
      "content": [
        {
          "element": "number",
          "attributes": {
            "line": {
              "element": "number",
              "content": 3
            },
            "column": {
              "element": "number",
              "content": 2
            }
          },
          "content": 4
```

(continues on next page)

(continued from previous page)

```
        },
        {
          "element": "number",
          "attributes": {
            "line": {
              "element": "number",
              "content": 3
            },
            "column": {
              "element": "number",
              "content": 10
            }
          },
          "content": 12
        }
      ]
    }
  ]
}
```

Data Structure Elements

Enumeration Element

The layout of the Enum Element has been altered. The enumerations have been moved to an enumerations attribute of the element and the content now represents the given value.

Enumerations themselves are an array of the possible enumerations.

Before

```
{
  "element": "enum",
  "content": [
    {
      "element": "string",
      "content": "north"
    },
    {
      "element": "string",
      "content": "east"
    },
    {
      "element": "string",
      "content": "south"
    },
    {
      "element": "string",
      "content": "west"
    }
  ]
}
```

After

```
{
  "element": "enum",
  "attributes": {
    "enumerations": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "content": "north"
        },
        {
          "element": "string",
          "content": "east"
        },
        {
          "element": "string",
          "content": "south"
        },
        {
          "element": "string",
          "content": "west"
        }
      ]
    }
  }
}
```

The intent of the structure was that it represents an enumeration of `north`, `east`, `south` and `west`. As the enumerations do not include a `fixed` type attribute it represents an enumeration where any string is valid and not just the fixed values. This created a limitation that tooling cannot determine the difference between one of the fixed element enumerations, or a type with a value. Thus, when the values are fixed they will now include a `fixed` type attribute as follows:

```
{
  "element": "enum",
  "attributes": {
    "enumerations": {
      "element": "array",
      "content": [
        {
          "element": "string",
          "attributes": {
            "typeAttributes": {
              "element": "array",
              "content": [
                {
                  "element": "string",
                  "content": "fixed"
                }
              ]
            }
          },
          "content": "north"
        },
        {
          "element": "string",
          "attributes": {
```

(continues on next page)

(continued from previous page)

```
            "typeAttributes": {
              "element": "array",
              "content": [
                {
                  "element": "string",
                  "content": "fixed"
                }
              ]
            }
          },
          "content": "east"
        },
        {
          "element": "string",
          "attributes": {
            "typeAttributes": {
              "element": "array",
              "content": [
                {
                  "element": "string",
                  "content": "fixed"
                }
              ]
            }
          },
          "content": "south"
        },
        {
          "element": "string",
          "attributes": {
            "typeAttributes": {
              "element": "array",
              "content": [
                {
                  "element": "string",
                  "content": "fixed"
                }
              ]
            }
          },
          "content": "west"
        }
      ]
    }
  }
}
```

CHAPTER 5

License

MIT License. See the LICENSE file.