Web of Things (WoT) Binding Templates

W3C Editor's Draft 08 June 2020

This version:
https://w3c.github.io/wot-binding-templates/

Latest published version:
https://www.w3.org/TR/wot-binding-templates/

Latest editor's draft:
https://w3c.github.io/wot-binding-templates/

Editors:
Michael Koster (SmartThings)
Ege Korkan (Siemens AG)

Contributors:
In the GitHub repository

Repository:
We are on GitHub
File a bug

Abstract

W3C Web of Things enables applications to interact with and orchestrate connected Things at Web scale. The standardized abstract interaction model exposed by the WoT Thing Description enables applications to scale and evolve independently of the individual Things.

Many network-level protocols and standards for connected Things have already been developed, and have millions of devices deployed in the field today. These standards are converging on a common set of transport protocols and transfer layers, but each has peculiar content formats, payload schemas, and data types.

Despite using unique formats and data models, the high-level interactions exposed by most connected things can be modeled using the Property, Action, and Event interaction affordances of the WoT Thing Description.

Binding Templates enable a Thing Description to be adapted to the specific protocol or data payload usage across the different standards. This is done through additional descriptive vocabulary that is used in the Thing Description.

This document describes the initial set of vocabulary extensions to the WoT Thing Description that make up the Binding Templates. It is expected over time that additional protocols and payload structures will be accommodated by further extending the Binding Templates.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

> EDITOR'S NOTE: The W3C WoT WG is asking for feedback
>
> Please contribute to this draft using the GitHub Issue feature of the WoT Binding Templates repository. For feedback on security and privacy considerations, please use the WoT Security and Privacy Issues, as they are cross-cutting over all our documents.

This document was published by the Web of Things Working Group as an Editor's Draft.

Comments regarding this document are welcome. Please send them to public-wot-wg@w3.org (archives).

Publication as an Editor's Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 1 March 2019 W3C Process Document.

Table of Contents

1. Introduction §

Binding Templates consist of reusable vocabulary and extensions to the WoT Thing Description[WOT-THING-DESCRIPTION] format that enable an application client (a Consumer) to interact, using a consistent interaction model, with Things that expose diverse protocols and protocol usage.

Binding Templates enable Consumers to adapt to the underlying protocol and network-facing API constructions. Once the base protocol (e.g., HTTP[RFC7231], CoAP[RFC7252], MQTT[MQTT], etc.) is identified, the following adaptions specify the particular use within the given platform.

> EDITOR'S NOTE: Additional Protocol Bindings
>
> This document contains examples of Protocol Bindings for HTTP, CoAP, and MQTT. Other protocols may be added following the same design style and using payload mappings that can be expressed as JSON compatible entities. Future extensions to other payload definition formats are also contemplated.

1.1 Protocol Methods and Options §

Most protocols have a relatively small set of methods that define the message type, the semantic intention of the message. REST and PubSub architecture patterns result in different protocols with different methods. Common methods found in these protocols are GET, PUT, POST, DELETE, PUBLISH, and SUBSCRIBE. Binding Templates describe how these existing methods and vocabularies can be described in a Thing Description.

This is done by mapping the protocol methods to the abstract WoT Interaction Affordance terms `readproperty`, `writeproperty`, `observeproperty`, `unobserveproperty`, `invokeaction`, `subscribeevent`, `unsubscribeevent`, `readallproperties`, `writeallproperties`, `readmultipleproperties`, `writemultipleproperties`.

Possible protocol options are also specified in the Protocol Binding. They are used to select transfer modes, to request notifications from observable resources, or otherwise extend the semantics of the protocol methods.

1.2 Media Types §

Maximum use should be made of IANA-registered Media Types [IANA-MEDIA-TYPES] (e.g., `application/json`) in order to decouple applications from connected Things. Standard bridges and translations from proprietary formats to Web-friendly languages such as JSON and XML are part of the adaptation needed.

Correct indication of Media Types enables proper processing of the serialized documents. This way, the documents can be exchanged in any format and allow the upper layers of an application to adapt to different formats.

1.3 Payload Structure §

Data serialized to a standard Media Type still remains in a structure specific to the platform data model and needs to be understood by Consumers (cf. various types of JSON documents).

The data definition language of `DataSchema` elements, described in [WOT-THING-DESCRIPTION], allows for describing arbitrary structures by nesting of arrays and objects. Constants and variable specifications may be intermixed.

1.4 Data Types and Value Constraints §

Simple data types and value constraints are currently used in a layered and descriptive way in [WOT-THING-DESCRIPTION]. Additional forms of constraints are available to help adapt to the underlying data types. A platform-specific 8-bit unsigned integer, for instance, can be defined as Integer with a minimum of 0 and maximum of 255; the system-specific representation (e.g., exact number of bits) on the Thing and the Consumer is not relevant for interoperability.

2. Conformance §

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MUST*, *MUST NOT*, *SHOULD*, and *SHOULD NOT* in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Terminology §

This section is non-normative.

The fundamental WoT terminology such as ***Thing***, ***Consumer***, ***Thing Description*** (***TD***), ***Interaction Model***, ***Interaction Affordance***, ***Property***, ***Action***, ***Event***, ***Protocol Binding***, ***Servient***, ***WoT Interface***, ***WoT Runtime***, etc. is defined in Section 3 of the WoT Architecture specification [WOT-ARCHITECTURE].

In addition, this specification introduces the following definitions:

TD Context Extension
: A mechanism to extend Thing Descriptions with additional Vocabulary Terms using `@context` as specified in JSON-LD[json-ld11]. It is the basis for semantic annotations and extensions to core mechanisms such as Protocol Bindings, Security Schemes, and Data Schemas.

Vocabulary
: A collection of Vocabulary Terms, identified by a namespace IRI.

Term* and *Vocabulary Term
: A character string. When a Term is part of a Vocabulary, i.e., prefixed by a namespace IRI[RFC3987], it is called a Vocabulary Term. For the sake of readability, Vocabulary Terms present in this document are always written in a compact form and not as full IRIs.

4. Binding Templates Overview §

This section is non-normative.

This section describes the mechanisms of binding templates with examples.

4.1 Data Schema §

A data schema describes the payload structure and included data items that are passed between the Consumer and the Thing during interactions.

4.1.1 Payload Structure §

Payload Structure is determined by `DataSchema` elements of a Thing Description. DataSchema elements should be used by an instance of a `PropertyAffordance`, `input`/`output` of `ActionAffordance`, `data`/`subscription`/`cancellation` of an `EventAffordance` or by a `uriVariable` of the `InteractionAffordance`. As indicated in the [WOT-THING-DESCRIPTION], `DataSchema` Vocabulary is a subset of JSON Schema [json-schema]

In the case of Action Affordances, the additional keywords `input` and `output` are used to provide two different schemas when data might be exchanged in both directions, such as in the case of invoking an Action Affordance with input parameters and receiving status information.

In the case of Event Affordances, the additional keywords `data`, `subscription` and `cancellation` are used to describe the payload when the event data is delivered by the Exposed Thing, the payload needed to subscribe to the event and the payload needed to cancel receiving event data from the Exposed Thing, respectively.

In addition to the example pattern in [WOT-THING-DESCRIPTION] of an object with name/value constructs or simple arrays, Protocol Bindings for existing standards may require nested arrays and objects, and some constant values to be specified.

Below are examples of different payloads and their corresponding `DataSchema`.

For example, a simple payload structure may use a map:

EXAMPLE 1: Simple JSON Object Payload

```
{
  "level": 50,
  "time": 10
}
```

EXAMPLE 2: DataSchema for Simple JSON Object Payload

```
{
  "type": "object",
  "properties": {
    "level": {
      "@type": ["iot:LevelData"],
      "type": "integer",
      "minimum": 0,
      "maximum": 255
    },
    "time": {
      "@type": ["iot:TransitionTimeData"],
      "type": "integer",
      "minimum": 0,
```

```
      "maximum": 65535
    }
  }
}
```

SenML [RFC8428] might use the following construct:

EXAMPLE 3: SenML Example

```
[
  {
    "bn": "/example/light/"
  },
  {
    "n": "level",
    "v": 50
  },
  {
    "n": "time",
    "v": 10
  }
]
```

EXAMPLE 4: DataSchema for SenML Payload

```
{
  "type": "array",
  "items": [
    {
      "type": "object",
      "properties": {
        "bn": {
          "type": "string",
          "const": "example/light"
        }
      }
    },
    {
      "type": "object",
      "properties": {
        "n": {
          "type": "string",
          "const": "level"
        },
        "v": {
          "@type": ["iot:LevelData"],
          "type": "integer",
          "minimum": 0,
          "maximum": 255
        }
      }
    },
    {
      "type": "object",
      "properties": {
        "n": {
          "type": "string",
          "const": "time"
        },
        "v": {
          "@type": ["iot:TransitionTimeData"],
          "type": "integer",
          "minimum": 0,
          "maximum": 65535
        }
      }
```

```
    }
  ]
}
```

A Batch Collection according to OCF[OCF] may be structured like this:

EXAMPLE 5: OCF Batch Example

```
[
  {
    "href": "/example/light/level",
    "rep": {
      "dimmingSetting": 50
    }
  },
  {
    "href": "/example/light/time",
    "rep": {
      "rampTime": 10
    }
  }
]
```

EXAMPLE 6: DataSchema for OCF Batch Payload

```
{
  "type": "array",
  "items": [
    {
      "type": "object",
      "properties": {
        "href": {
          "type": "string",
          "const": "/example/light/level"
        },
        "rep": {
          "type": "object",
          "properties": {
            "dimmingSetting": {
              "@type": ["iot:LevelData"],
              "type": "integer",
              "minimum": 0,
              "maximum": 255
            }
          }
        }
      }
    },
    {
      "type": "object",
      "properties": {
        "href": {
          "type": "string",
          "const": "/example/light/time"
        },
        "rep": {
          "type": "object",
          "properties": {
            "rampTime": {
              "@type": ["iot:TransitionTimeData"],
              "type":"integer",
              "minimum": 0,
              "maximum": 65535
            }
          }
        }
      }
```

```
    }
  ]
}
```

And an IPSO Smart Object on LWM2M [LWM2M] might look like the following:

EXAMPLE 7: IPSO/LWM2M Example

```
{
  "bn": "/3001/0/",
  "e": [
    {
      "n": "5044",
      "v": 0.5
    },
    {
      "n": "5002",
      "v": 10.0
    }
  ]
}
```

EXAMPLE 8: DataSchema for IPSO/LWM2M Payload

```
{
  "type": "object",
  "properties": {
    "bn": {
      "type": "string",
      "const": "/3001/0/"
    },
    "e": {
      "type": "array",
      "items": [
        {
          "type": "object",
          "properties": {
            "n": {
              "type": "string",
              "const": "5044"
            },
            "v": {
              "@type": ["iot:LevelData"],
              "type": "number",
              "minimum": 0.0,
              "maximum": 1.0
            }
          }
        },
        {
          "type": "object",
          "Properties": {
            "n": {
              "type": "string",
              "const": "5002"
            },
            "v": {
              "@type": ["iot:TransitionTimeData"],
              "type": "number",
              "minimum": 0.0,
              "maximum": 6553.5
            }
          }
        }
      ]
```

```
    }
  }
}
```

4.1.2 Data Types and value constraints §

Note that in Example 7 above, the values are floating point (`double`) while the other examples have integer values. In general, Consumers should follow the data schemas strictly, not generating anything not given in the WoT Thing Description, but should accept additional data from the Thing not given explicitly in the WoT Thing Description. This means that a Consumer sending the payload of the Example 7 should use floating points in the payload.

4.1.3 XML Schema Binding §

In the previous sections, the examples showed what data, whose value is described using the Data Schema, look like when serialized to JSON. This section describes how type definitions described using the Data Schema can be mapped to XML schema definitions by using the same examples. Given these Data Schemas, providing the mapping to XML schema allows XML tools to directly validate serialized XML data, for example. The XML structure for which this mapping is designed is based on EXI4JSON [exi-for-json].

4.1.3.1 Object Definition to XML Schema §

Shown below is an example Data Schema of an Object Schema. The object consists of two named literals `id` (of type `integer`) and `name` (of type `string`) where `id` is required to be present.

EXAMPLE 9: JSON Schema description of the required JSON Object

```
{
  "type": "object",
  "properties": {
      "id": {
              "type": "integer"
      },
      "name": {
              "type": "string"
      }
  },
  "required": [
      "id"
  ]
}
```

When the `object` is anonymous (i.e. it is the root, or participates in an `array` definition), the above `object` definition transforms to the following XML Schema element definition.

EXAMPLE 10: XML Schema mapping of the above JSON Schema with an anonymous object

```
<xs:element name="object" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType>
        <xs:all>
            <xs:element name="id">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="integer" type="xs:integer" />
                  </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="name" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="string" type="xs:string" />
                  </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:all>
    </xs:complexType>
</xs:element>
```

Otherwise (i.e. the `object` is a member of another `object` definition, thus has a name), the object definition transforms to the following XML schema element definition. Note `$name` represents the name of the `object`, and needs to be replaced by the actual name of the `object`.

EXAMPLE 11: XML Schema mapping of the above JSON Schema with a non-anonymous object

```
<xs:element name="$name" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType>
        <xs:sequence>
        <!--Until the next comment, it is a copy of the previous example-->
            <xs:element name="object">
                <xs:complexType>
                    <xs:all>
                        <xs:element name="id">
                            <xs:complexType>
                              <xs:sequence>
                                <xs:element name="integer" type="xs:integer" />
                              </xs:sequence>
                            </xs:complexType>
                        </xs:element>
                        <xs:element name="name" minOccurs="0">
                            <xs:complexType>
                              <xs:sequence>
                                <xs:element name="string" type="xs:string" />
                              </xs:sequence>
                            </xs:complexType>
                        </xs:element>
                    </xs:all>
                </xs:complexType>
            </xs:element>
        <!--Until here-->
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

4.1.3.2 Array Schema to XML Schema §

Shown below is an example Data Schema of an Array Schema. The `array` consists of exactly three number literals with each value within the value range of [0 ... 2047].

EXAMPLE 12: JSON Schema description of the required JSON Array

```
{
    "type": "array",
    "items": {
        "type": "number",
        "minimum": 0,
        "maximum": 2047
    },
    "minItems": 3,
    "maxItems": 3
}
```

When the `array` is anonymous (i.e. it is the root, or participates in another `array` definition), the above `array` definition transforms to the following XML Schema element definition.

EXAMPLE 13: XML Schema mapping of the above JSON Schema with an anonymous array

```
<xs:element name="array" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="double" minOccurs="3" maxOccurs="3">
                <xs:simpleType name="minInclusive">
                    <xs:restriction base="xs:double">
                        <xs:minInclusive value="0"/>
                        <xs:maxInclusive value="2047"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Otherwise (i.e. the `array` is a member of an `object` definition, thus has a name), the `array` definition transforms to the following XML schema element definition. Note `$name` represents the name of the `array`, and needs to be replaced by the actual name of the `array`.

EXAMPLE 14: XML Schema mapping of the above JSON Schema with a non-anonymous array

```
<xs:element name="$name" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType>
        <xs:sequence>
        <!--Until the next comment, it is a copy of the previous example-->
            <xs:element name="array">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="double" minOccurs="3" maxOccurs="3" >
                            <xs:simpleType name="minInclusive">
                                <xs:restriction base="xs:double">
                                    <xs:minInclusive value="0"/>
                                    <xs:maxInclusive value="2047"/>
                                </xs:restriction>
                            </xs:simpleType>
                        </xs:element>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        <!--Until here-->
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

4.2 Forms Element §

The form elements contain the URI [RFC3986] pointing to an instance of the interaction and descriptions of the protocol settings and options expected to be used when between the Consumer and the Thing for the interaction.

4.2.1 Operation Types §

Form Operation Types describe the intended semantics of performing the operation described by the form.

For example, the Property interaction allows read and write operations. The protocol binding may contain a form for the read operation and a different form for the write operation. The value of the `op` attribute of the form indicates which form is which and allows the Consumer to select the correct form for the operation required.

EXAMPLE 15: Form Operation Types

```
"op": "readproperty"
"op": "writeproperty"
```

The vocabulary in section 4 lists the recommended set of form operations, and the full TD examples in section 5 contain example uses of form operations types.

4.2.2 Content Types §

Content Types define the serialization details and other rules for processing the payloads. The content type is used to select a serializer/deserializer and to select an additional set of rules and constraints for the protocol driver. Content type includes the media type and potential parameters for the media type.

For example, the media type `application/ocf+cbor` indicates that CBOR serialization is used, but also that OCF rules and namespaces apply to the processing of the representations.

Some special protocol drivers may be invoked by using a non-registered media type (e.g., `x-`) along with a custom URI Scheme [RFC3986] and its own set of protocol methods and options defined for that URI Scheme.

When the media type is `application/xml` (or its binary representation `application/exi`) and there is a Data Schema provided for the payload, the payloads are constrained by a XML Schema derived from the Data Schema. See § 3.1.3 XML Schema Binding Schema Binding for how to derive a XML Schema from a Data Schema definition.

Below are some examples of payloads in JSON and their corresponding equivalent payloads in XML.

EXAMPLE 16: JSON Payload of Object type for specifying the desired brightness and flashing frequency of a lamp

```
{
  "brightness": 200,
  "frequency": "fast"
}
```

EXAMPLE 17: Corresponding XML Payload for the previous JSON Payload

```
<object>
  <brightness>
    <integer>200</integer>
  </brightness>
  <frequency>
    <string>fast</string>
  </frequency>
</object>
```

EXAMPLE 18: JSON Payload of Array type for specifying the desired brightness and flashing frequency of a lamp

```
[
  520,
  184,
  1314
]
```

EXAMPLE 19: Corresponding XML Payload for the previous JSON Payload

```
<array>
  <number>520</number>
  <number>184</number>
  <number>1314</number>
</array>
```

4.2.3 Protocol Methods and Options §

Each target protocol may specify different method names for similar operations, and there may be semantic differences between similar method names of different protocols. Additionally, platforms may use different methods for realizing a particular WoT Interaction Affordance. For example, POST may be used for writing a Property value in one platform, while PUT may be used in another. For these reasons, we require the ability to specify which method to use for a particular Interaction. We also will provide vocabulary to differentiate between methods of different protocols.

The W3C RDF vocabulary for HTTP [HTTP-in-RDF10] is used to identify the methods and options specified in the HTTP protocol bindings.

For the sake of consistency, we will use the same ontology design pattern to derive a vocabulary for each target protocol, e.g. CoAP, MQTT.

The example below shows some method definitions for various protocols.

EXAMPLE 20: Vocabulary Example for Methods

```
"htv:methodName": "GET"

"mqv:controlPacketValue": "SUBSCRIBE"

"cov:methodName": "GET"
```

Header options in HTTP, CoAP, MQTT sometimes must be included in a protocol binding in order to successfully interact with the underlying protocol. The example below shows the structure of the definition for HTTP header options, according to the W3C HTTP Vocabulary in RDF.

EXAMPLE 21: HTTP Vocabulary Example for Header Options

```
"htv:headers":
  [
    {
      "htv:fieldName": "Accept",
      "htv:fieldValue": "application/json"
    },
    {
      "htv:fieldName": "Transfer-Encoding",
      "htv:fieldValue": "chunked"
    }
  ]
```

Note: different forms in a binding may need different header constructions, therefore the `htv:headers` construct is an extension of the TD "form" element.

Protocols may have defined sub-protocols that can be used for some interaction types. For example, to receive asynchronous notifications using HTTP, some servers may support long polling (`longpoll`), WebSub [WebSub] (`websub`) and Server-Sent Events [eventsource] (`sse`). The `subprotocol` item may be defined in a form instance to indicate the use of one of these protocols, for example long polling with its special use of HTTP:

EXAMPLE 22: subprotocol

```
{
  "op": "subscribeevent",
  "href": "https://mylamp.example.com/overheating",
  "subprotocol": "longpoll"
}
```

4.2.4 URI Template Variables §

When Interaction Affordances require dynamic variables in the `href`, they can be described using `uriVariables` in the Data Schema of the interaction. For example, `p` and `d` in `http://192.168.1.25/left?p=2&d=1` can be described with a template as defined in [RFC6570]]: `http://192.168.1.25/left{?p,d}`.

In such a case, the URI Template variables *MUST* be collected in the JSON-object based `uriVariables` member with the associated (unique) variable names as JSON names.

The serialization of each value in the map assigned to `uriVariables` in an instance of `Form` *MUST* rely on the DataSchema as explained in § 3.1 Data Schema.

An action affordance snippet using a URI Template and `uriVariables` is given below:

EXAMPLE 23: Action Affordance using URI Template Variables

```
...
"actions": {
    "LeftDown": {
        ...
        "uriVariables": {
            "p" : { "type": "integer", "minimum": 0, "maximum": 16 },
            "d" : { "type": "integer", "minimum": 0, "maximum": 1 }
        },
        "forms": [{
          "href" : "http://192.168.1.25/left{?p,d}",
          "htv:methodName": "GET"
        }]
    },
    ...
},
...
```

4.3 Interaction Affordances §

This section is non-normative.

This section describes unique aspects of protocol bindings for the three WoT Interaction Affordances.

4.3.1 Bindings for Properties §

This section describes unique aspects of protocol bindings for WoT Property interactions.

The abstract operations exposed for the Property Interaction are `readproperty`, `writeproperty`, `observeproperty` and `unobserveproperty`. These are mapped by using form operations that describe how the abstract operation is performed, resulting in a semantic interpretation similar to HTML form submission.

Additionally, the abstract operations exposed for multiple Property Interactions are `readallproperties`, `writeallproperties`, `readmultipleproperties` and `writemultipleproperties`.

EXAMPLE 24: Example use of form operation for Property

```
{
  "op": "writeproperty",
  "href": "/example/level",
  "htv:methodName": "POST"
}
```

The form element in the example above conveys the statement: *"To do a `writeproperty` of the subject Property (context of the form), perform an `HTTP POST` on the resource at the target URI `/example/level`."*

Properties may be observable, defined by the TD keyword "observable". If there is an observe form and a retrieve form, the observe form may be indicated by including `op=observeproperty` in the form. The observe form may also specify header options to use, as specified in Observing in CoAP[RFC7641]for example setting the CoAP Observe option to `0` in the header, starts observation.

4.3.2 Bindings for Actions §

This section is non-normative.

This section describes unique aspects of protocol bindings for Actions.

The abstract operation on Actions is `invokeaction`. In the same way that the abstract operations on Properties are mapped using form operation types, the abstract operation of Actions is also mapped.

EXAMPLE 25: Example use of form operation for Action

```
{
  "op": "invokeaction",
  "href": "/example/levelaction",
  "http:methodName": "POST"
}
```

The form element in the example above conveys the statement: *"To do an `invokeaction` of the subject Action (context of the form), perform a `POST` on the resource at the target URI `/example/levelaction`."*

4.3.3 Bindings for Events §

This section is non-normative.

This section describes unique aspects of protocol bindings for WoT Event Interaction Affordances.

The abstract operations on Events are `subscribeevent` and `unsubscribeevent`. The `subscribeevent` operation may directly enable event instance delivery from the pre-defined URI to observable resources or pubsub topics encoded in URIs. Alternatively, it may return a location or resource URI from which event instance may be obtained, either by observation or some other mechanism, depending on the transfer protocol.

Usually, the `unsubscribeevent` only occurs when the transfer protocol has no implicit unsubscribe operation such as closing the connection. Examples are Webhooks that require particular unsubscribe requests.

If the binding offers an observable Event resource from which events are obtained, there will be a form which describes the required transfer layer operation, for example CoAP Observe or HTTP Long Polling.

EXAMPLE 26: Example use of form operation for Events

```
{
  "op": "subscribeevent",
  "href": "mqtt://wot.example.com/levelevent",
  "mqv:controlPacketValue": "SUBSCRIBE"
}
```

The form element in the example above conveys the statement: *"To do an `subscribeevent` of the subject Event (context of the form), perform an `MQTT SUBSCRIBE` on the topic `/levelevent` on the broker at `wot.example.com` using the default MQTT port."*

5. Vocabulary §

This section summarizes the vocabulary used for Binding Templates. The vocabulary is defined in other documents, in particular the WoT Thing Description [WOT-THING-DESCRIPTION]

5.1 `DataSchema` Vocabulary §

`DataSchema` elements describe the structure of the payload. The `DataSchema` class and vocabulary is defined in [WOT-THING-DESCRIPTION]. Properties and Events directly implement the `DataSchema` class (i.e., they contain the corresponding fields such as `type`), which describes the data transfer in either direction. Actions may define an `input` data schema for actuation data being sent to the Action and/or an `output` data schema for result or status data being returned from the Action.

5.2 Form Operation Type Vocabulary §

Each interaction affordance has associated form operation types (i.e. `op`) that are used to select the form element corresponding to the intended interaction from the Array of forms. For example, for one interaction, a Consumer can choose the form element corresponding to reading a Property, observing a Property or writing to Property by using the form operation type.

5.2.1 Property Forms §

Properties can provide `readproperty` and `writeproperty` operations, which map to GET and PUT/POST of a REST API, respectively. Properties may also by observed if they provide an `observeproperty` operation and the observation can be stopped if the property provides an `unobserveproperty` operation.

op Term	*Description*
`readproperty`	Read a Property. Requires `writeOnly` to be set to `false`.

op Term	*Description*
`writeproperty`	Write a Property. Requires `readOnly` to be set to `false`.
`observeproperty`	Observe a Property. Requires `observable` to be set to `true`.
`unobserveproperty`	Unobserve a Property. Requires `observable` to be set to `true`.

5.2.2 Action Forms §

Actions only provide `invokeaction` operations. For completeness, there is also a form operation type defined.

op Term	*Description*
`invokeaction`	Invoke an Action.

5.2.3 Event Forms §

Events describe subscription endpoints from which to event instances can be received and unsubscription endpoints to stop receiving event instances.

op Term	*Description*
`subscribeevent`	Subscribe to an Event.
`unsubscribeevent`	Unsubscribe from an Event.

5.3 Protocol Vocabulary §

Extensions to the Thing Description core vocabulary can inform the Consumer about protocol-specific message configurations such as methods, options, and status codes. By using such information, the Consumer can build the protocol specific request that allows interaction with the Exposed Thing. Per default the Thing Description includes HTTP Vocabulary by including the HTTP RDF vocabulary definitions from HTTP Vocabulary in RDF 1.0 [HTTP-in-RDF10].

EDITOR'S NOTE: Protocol Vocabulary Definitions

The WoT Working Group is investigating good ways to also provide COAP and MQTT Vocabulary in RDF. Whether the WG will publish corresponding WG Notes is still subject to discussion.

The protocol vocabularies for each protocol are presented as two tables per protocol. The first table details the vocabulary terms, whereas the second one lists a default mapping of the `op` for the given protocol, if that `op` value is defined for the given protocol.

Other protocols can be easily integrated into a Thing Description. To do so, the vocabulary of the protocol should be linked via `@context` context extension, in which the different vocabulary terms used for the protocol should be described. A URI Scheme of the protocol is also necessary to identify it in the `href` value. A Consumer should be able to construct the appropriate requests based vocabulary terms defined in the Thing Description and should not rely on out-of-band information.

5.3.1 HTTP Vocabulary §

5.3.1.1 HTTP Vocabulary Terms §

Vocabulary term	*Description*	*Assignment*	*Type*
`htv:methodName`	HTTP method name (Literal).	optional	`string` (one of `"GET"`, `"PUT"`, `"POST"`, `"DELETE"`, `"PATCH"`)
`htv:headers`	HTTP headers sent with the message.	optional	array of `htv:MessageHeader`
`htv:fieldName`	Header name (Literal), e.g., `"Accept"`, `"Transfer-Encoding"`.	mandatory within `htv:MessageHeader`	`string`
`htv:fieldValue`	Header value (Literal).	mandatory within `htv:MessageHeader`	`string`

5.3.1.2 HTTP Default Vocabulary Terms §

op value	*Default Binding*
`readproperty`	`"htv:methodName": "GET"`
`writeproperty`	`"htv:methodName": "PUT"`
`invokeaction`	`"htv:methodName": "POST"`
`readallproperties`	`"htv:methodName": "GET"`
`writeallproperties`	`"htv:methodName": "PUT"`
`readmultipleproperties`	`"htv:methodName": "GET"`

op value	*Default Binding*
`writemultipleproperties`	`"htv:methodName": "PUT"`

5.3.2 CoAP Vocabulary §

5.3.2.1 CoAP Vocabulary Terms §

Vocabulary term	*Description*	*Assignment*	*Type*
`cov:methodName`	CoAP method name (Literal).	optional	`string` (one of `"GET"` (1), `"POST"` (2), `"PUT"` (3), `"DELETE"` (4), `"FETCH"` (5), `"PATCH"` (6), `"iPATCH"` (7))
`cov:options`	CoAP options sent with the message, e.g., `[ { "cov:optionName": "Accept", "cov:optionValue": 110 }]` to observe.	optional	array of `cov:MessageOption`
`cov:optionName`	Option name (Literal), see CoRE Parameters.	mandatory within `cov:MessageOption`	`string`
`cov:optionValue`	Header value (Literal).	mandatory within `cov:MessageOption`	`anyType`

5.3.2.2 CoAP Default Vocabulary Terms §

op value	*Default Binding*
`readproperty`	`"cov:methodName": "GET"`
`writeproperty`	`"cov:methodName": "PUT"`
`observeproperty`	`"cov:methodName": "GET","subprotocol":"cov:observe"`
`unobserveproperty`	`"cov:methodName": "GET","subprotocol":"cov:observe"`
`invokeaction`	`"cov:methodName": "POST"`

op value	*Default Binding*
`subscribeevent`	`"cov:methodName": "GET","subprotocol":"cov:observe"`
`unsubscribeevent`	`"cov:methodName": "GET","subprotocol":"cov:observe"`
`readallproperties`	`"cov:methodName": "GET"`
`writeallproperties`	`"cov:methodName": "PUT"`
`readmultipleproperties`	`"cov:methodName": "GET"`
`writemultipleproperties`	`"cov:methodName": "PUT"`

Observing Resources in CoAP should be done as specified in [RFC7641]. Since observing and unobserving need to be indicated with the Observe flag in the header, this mechanism can be also described in the `cov:options` with the value `[ { "cov:optionName": "Observe", "cov:optionValue": 0 }]`. However, this is not enough to describe the mechanism of getting asynchronous updates from the observed resource. Thus, `"subprotocol":"cov:observe"` is used to indicate the observation mechanism.

EDITOR'S NOTE

As indicated by the CoAP Default Vocabulary Terms, it is recommended to use the `subprotocol` to describe the observation. Thus, `cov:options` with the value `[ { "cov:optionName": "Observe", "cov:optionValue": 0 }]` *MUST NOT* be used together with `"subprotocol":"cov:observe"`, since it can lead to confusion.

5.3.3 MQTT Vocabulary §

5.3.3.1 MQTT Vocabulary Terms §

Vocabulary term	*Description*	*Assignment*	*Type*
`mqv:controlPacketValue`	MQTT Control Packet type (Literal).	optional	`string` (one of `"PUBLISH"` (3), `"SUBSCRIBE"` (8), `"UNSUBSCRIBE"` (10))
`mqv:options`	MQTT options sent with the message, e.g., `[ { "mqv:optionName": "qos", "mqv:optionValue": 1 }]`.	optional	array of `mqv:MessageOption`

Vocabulary term	*Description*	*Assignment*	*Type*
`mqv:optionName`	Option name (Literal).	mandatory within `mqv:MessageOption`)	`string` (one of `"qos"`, `"retain"`, `"dup"`)
`mqv:optionValue`	Header value (Literal).	mandatory within `mqv:MessageOption`)	One of `0`, `1` or `2` (only for `qos`)

5.3.3.2 MQTT Default Vocabulary Terms §

op value	*Default Binding*
`readproperty`	`"mqv:controlPacketValue": "SUBSCRIBE",`
`writeproperty`	`"mqv:controlPacketValue": "PUBLISH"`
`observeproperty`	`"mqv:controlPacketValue": "SUBSCRIBE"`
`unobserveproperty`	`"mqv:controlPacketValue": "UNSUBSCRIBE"`
`invokeaction`	`"mqv:controlPacketValue": "PUBLISH"`
`subscribeevent`	`"mqv:controlPacketValue": "SUBSCRIBE"`
`unsubscribeevent`	`"mqv:controlPacketValue": "UNSUBSCRIBE"`
`readallproperties`	`"mqv:controlPacketValue": "SUBSCRIBE"`
`writeallproperties`	`"mqv:controlPacketValue": "PUBLISH"`
`readmultipleproperties`	`"mqv:controlPacketValue": "SUBSCRIBE"`
`writemultipleproperties`	`"mqv:controlPacketValue": "PUBLISH"`

For the MQTT protocol, if an MQTT client publishes a message to a topic with the retain flag set to true, the future subscribers of the topic will also get this message. Outside of this case, it is not possible to read a property but only possible to observe it. Additionally, in a Form element with MQTT protocol, if the `op` contains `readproperty` (meaning that retain flag is set to true), it *SHOULD* also contain `observeproperty`. On the other hand, if the MQTT publisher does not set the retain flag to true, the property will be only observable. In this case, the property in the exposed Thing Description *SHOULD NOT* have Form elements with MQTT protocol containing `readproperty` operation.

5.3.4 `subprotocol` Vocabulary §

The `subprotocol` field is defined in [WOT-THING-DESCRIPTION].

Currently, the supported values are `longpoll`, `websub` and `sse` defined for HTTP. Subprotocols can be used for asynchronous event delivery or observing Properties.

For WebSockets, the IANA-registered Websocket Subprotocols [iana-web-socket-registry] may be used.

For CoAP, `"subprotocol":"cov:observe"` can be used to describe asynchronous observation operations as defined by [RFC6741]

6. Examples of Thing Descriptions including protocol bindings §

This section is non-normative.

The following TD examples uses a fictional CoAP and MQTT Protocol Bindings, as no such Protocol Binding is available at the time of writing this specification. These TD Context Extensions assume that there is a CoAP and MQTT in RDF vocabulary similar to [HTTP-in-RDF10] that is accessible via the namespace `http://www.example.org/coap-binding#` and `http://www.example.org/mqtt-binding#`, respectively. The supplemented `cov:methodName` member instructs the Consumer which CoAP method has to be applied (e.g., `GET` for the CoAP Method Code 0.01, `POST` for the CoAP Method Code 0.02, or `iPATCH` for CoAP Method Code 0.07). The supplemented `"mqv:controlPacketValue"` member instructs the Consumer which MQTT command has to be applied (e.g., `8` for the subscribing and `10` for unsubscribing).

A TD with simple payload format and protocols can be seen below. Here each interaction affordance has one form with one protocol.

EXAMPLE 27: TD with a Simple Payload


```
{
  "@context": [
    "https://www.w3.org/2019/wot/td/v1",
    {
      "iot": "http://iotschema.org/",
      "cov": "http://www.example.org/coap-binding#",
      "mqv": "http://www.example.org/mqtt-binding#"
    }
  ],
  "@type": [ "Thing", "iot:Light", "iot:LevelCapability", "iot:BinarySwitchCapability" ],
  "base": "http://example.com",
  "title": "Lamp",
  "id": "urn:dev:ops:32473-WoTLamp-1234",
  "securityDefinitions": {"basic_sc": {
    "scheme": "basic",
    "in": "header"
  }},
  "security": ["basic_sc"],
  "properties": {
    "switchState": {
      "@type": ["iot:SwitchStatus", "iot:SwitchData"],
      "type": "boolean",
      "writeOnly": false,
      "readOnly": false,
      "observable": false,
      "forms": [
        {
          "href": "/example/light/currentswitch",
          "op": ["readproperty", "writeproperty"],
          "contentType": "application/json"
        }
      ]
    },
    "brightness": {
      "@type": ["iot:CurrentLevel", "iot:LevelData"],
      "type": "number",
      "writeOnly": false,
      "readOnly": false,
      "observable": false,
      "forms": [
        {
          "href": "coap://example.com/example/light/currentdimmer",
          "op": ["readproperty", "writeproperty"],
          "contentType": "application/json"
        }
      ]
    }
  },
  "actions": {
    "switchOn": {
```

```
      "@type": ["iot:SwitchOnAction"],
      "input":  {
        "type": "boolean",
        "const": true
      },
      "forms": [
        {
          "href": "/example/light/currentswitch",
          "op": ["invokeaction"],
          "contentType": "application/json"
        }
      ]
    },
    "switchOff": {
      "@type": ["iot:SwitchOff"],
      "input":  {
        "type": "boolean",
        "const": false
      },
      "forms": [
        {
          "href": "/example/light/currentswitch",
          "op": ["invokeaction"],
          "contentType": "application/json"
        }
      ]
    },
    "setBrightness": {
      "@type": ["iot:SetLevelAction"],
      "input":  {
        "@type": ["iot:LevelData"],
        "type": "number"
      },
      "forms": [
        {
          "href": "/example/light/currentdimmer",
          "op": ["invokeaction"],
          "contentType": "application/json"
        }
      ]
    }
  }
}
```

Another version of the previous TD with complex payload and multiple protocol options is shown below. Notably, the `brightness` property can be read via HTTP, written to via CoAP and observed via MQTT.

EXAMPLE 28: TD with protocol options and complex payload

```
{
  "@context": [
    "https://www.w3.org/2019/wot/td/v1",
    {
      "iot": "http://iotschema.org/",
      "cov": "http://www.example.org/coap-binding#",
      "mqv": "http://www.example.org/mqtt-binding#"
    }
  ],
  "base": "http://example.com/",
  "@type": [ "Thing", "iot:Light", "iot:LevelCapability", "iot:BinarySwitch" ],
  "title": "Lamp",
  "id": "urn:dev:ops:32473-WoTLamp-1234",
  "securityDefinitions": {"basic_sc": {
    "scheme": "basic",
    "in": "header"
  }},
  "security": ["basic_sc"],
  "properties": {
    "switchState": {
      "@type": ["iot:SwitchStatus"],
      "type": "object",
      "properties": {
        "switch": {
          "@type": ["iot:SwitchData"],
          "type": "boolean"
        }
      },
      "writeOnly": false,
      "readOnly": false,
      "observable": true,
      "forms": [
        {
          "href": "/example/light/currentswitch",
          "contentType": "application/json",
          "op": ["readproperty"],
          "htv:methodName": "GET"
        },
        {
          "href": "/example/light/currentswitch",
          "contentType": "application/json",
          "op": ["writeproperty"],
          "htv:methodName": "POST"
        },
        {
          "href": "mqtt://example.com/example/light/currentswitch",
          "op": ["observeproperty"],
          "mqv:controlPacketValue": "SUBSCRIBE"
        }
      ]
```

```
    },
    "brightness": {
      "@type": ["iot:CurrentLevel"],
      "type": "object",
      "properties": {
        "brightness": {
          "@type": ["iot:LevelData" ],
          "type": "integer",
          "minimum": 0,
          "maximum": 255
        }
      },
      "writeOnly": false,
      "readOnly": false,
      "observable": true,
      "forms": [
        {
          "href": "coap://example.com/example/light/currentdimmer",
          "contentType": "application/json",
          "op": ["readproperty"],
          "cov:methodName": "GET"
        },
        {
          "href": "/example/light/currentdimmer",
          "contentType": "application/json",
          "op": ["writeproperty"],
          "htv:methodName": "POST"
        },
        {
          "href": "mqtt://example.com/example/light/currentdimmer",
          "op": ["observeproperty"],
          "mqv:controlPacketValue": "SUBSCRIBE"
        }
      ]
    },
    "transitionTime": {
      "@type": ["iot:TransitionTime"],
      "type": "object",
      "properties": {
        "transitionTime": {
          "@type": ["iot:TransitionTimeData" ],
          "type": "integer",
          "minimum": 0,
          "maximum": 255
        }
      },
      "writeOnly": false,
      "readOnly": false,
      "observable": false,
      "forms": [
        {
          "href": "/example/light/transitiontime",
          "contentType": "application/json",
```

```
          "op": ["readproperty"],
          "htv:methodName": "GET"
        },
        {
          "href": "/example/light/transitiontime",
          "contentType": "application/json",
          "op": ["writeproperty"],
          "htv:methodName": "POST"
        }
      ]
    }
  },
  "actions": {
    "switchOn": {
      "@type": ["iot:SwitchOnAction"],
      "input":  {
        "type": "object",
        "properties": {
          "type": "boolean",
          "const": true
        }
      },
      "forms": [
        {
          "href": "/example/light/currentswitch",
          "contentType": "application/json",
          "op": ["invokeaction"],
          "htv:methodName": "POST"
        }
      ]
    },
    "switchOff": {
      "@type": ["iot:SwitchOffAction"],
      "input": {
            "type": "boolean",
            "const": false
        }
      },
      "forms": [
        {
          "href": "/example/light/currentswitch",
          "contentType": "application/json",
          "op": ["invokeaction"],
          "htv:methodName": "POST"
        }
      ]
    },
    "setBrightness": {
      "title": "Set Brightness Level",
      "@type": ["iot:SetLevelAction"],
      "input": {
        "type": "object",
        "properties": {
```

```
          "brightness": {
            "@type": ["iot:LevelData"],
            "type": "integer",
            "minimum": 0,
            "maximum": 255
            }
          },
          "transitionTime": {
            "@type": ["iot:TransitionTimeData"],
            "type": "integer",
            "minimum": 0,
            "maximum": 65535
          }
      },
      "forms": [
        {
          "href": "/example/light/",
          "contentType": "application/json",
          "op": ["invokeaction"],
          "htv:methodName": "POST"
        }
      ]
    }
  }
```

7. Security and Privacy Considerations §

EDITOR'S NOTE

Security and privacy considerations are still under discussion and development; the content below should be considered preliminary. Due to the complexity of the subject we are considering producing a separate document containing a detailed security and privacy considerations discussion including a risk analysis, threat model, recommended mitigations, and appropriate references to best practices. A summary will be included here. Work in progress is located in the WoT Security and Privacy repository. Please file any security or privacy considerations and/or concerns using the GitHub Issue feature.

Security is a cross-cutting issue that needs to be taken into account in all WoT building blocks. The W3C WoT does not define any new security mechanisms, but provides guidelines to apply the best practices from Web security, IoT security, and information security for general software and hardware considerations.

The WoT Thing Description must be used together with integrity protection mechanisms and access control policies. Users must ensure that no sensitive information is included in the TDs themselves.

The WoT Binding Templates must correctly cover the security mechanisms employed by the underlying IoT platform. Due to the automation of network interactions necessary in the IoT, operators need to ensure that Things are exposed and consumed in a way that is compliant with their security policies.

The WoT Runtime implementation for the WoT Scripting API must have mechanisms to prevent malicious access to the system and isolate scripts in multi-tenant Servients.

A. Example Sequences of Interaction Affordances §

This section is non-normative.

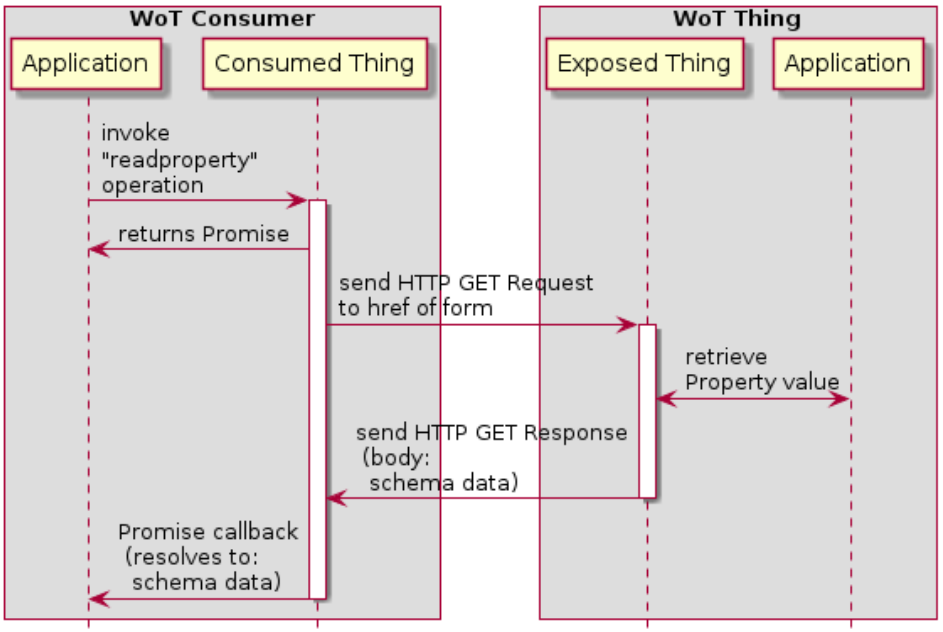
This section illustrates example sequences of application and protocol transactions that correspond to operations (defined in the Thing Description Specification) implementing various interactions among WoT Consumer and WoT Things. The illustrations show both the concrete protocol transactions and the interactions between the applications running inside the WoT Consumer and WoT Thing and the Consumed Thing and Exposed Thing abstractions.

For the sake of simplicity, remote and local proxies between the Consumer and the Thing are omitted from the following sequences. We also assume HTTP as the concrete protocol and omit any additional transactions for implementing security, such as those that would be used for authentication or to set up a secure connection for HTTPS. Other concrete protocols and the addition of security transactions however would only affect the concrete protocol transactions, not the application-level interactions with the Consumed Thing and Exposed Thing abstractions.

A.1 Property Interactions §

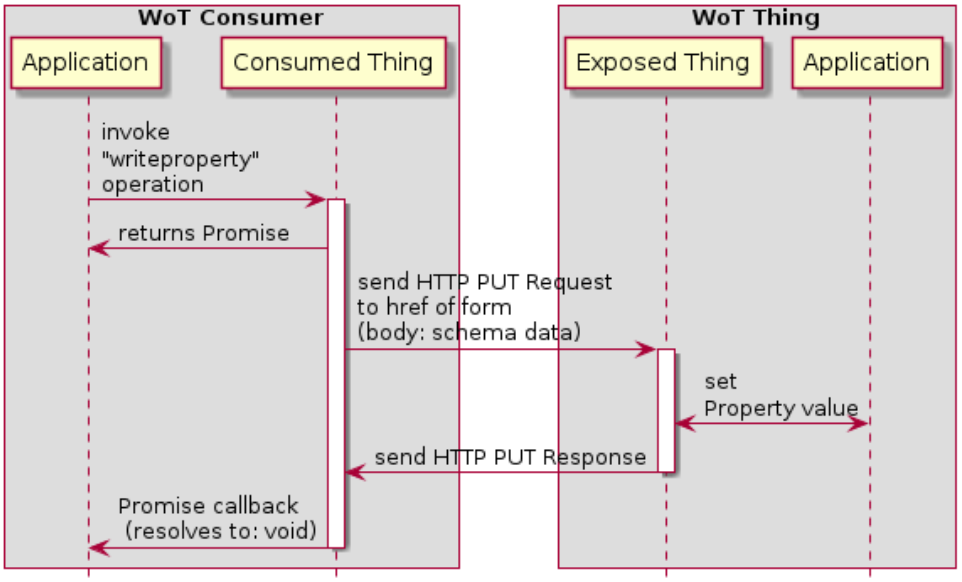
A.1.1 Read property (HTTP binding) §

The following sequence illustrates application and network transactions to implement the `readproperty` operation with an HTTP protocol binding.



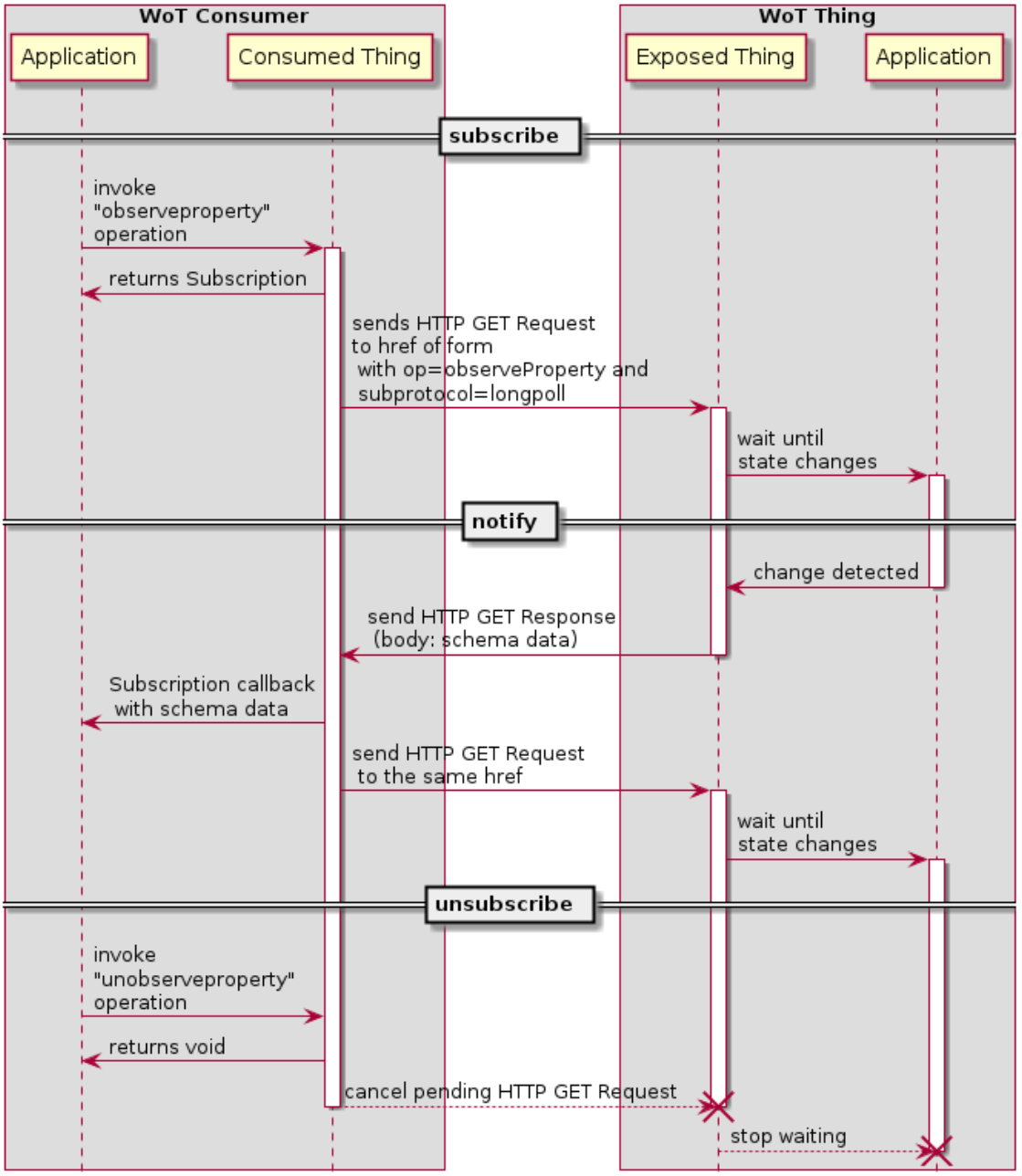
A.1.2 Write property (HTTP binding) §

The following sequence illustrates application and network transactions to implement the `writeproperty` operation with an HTTP protocol binding.



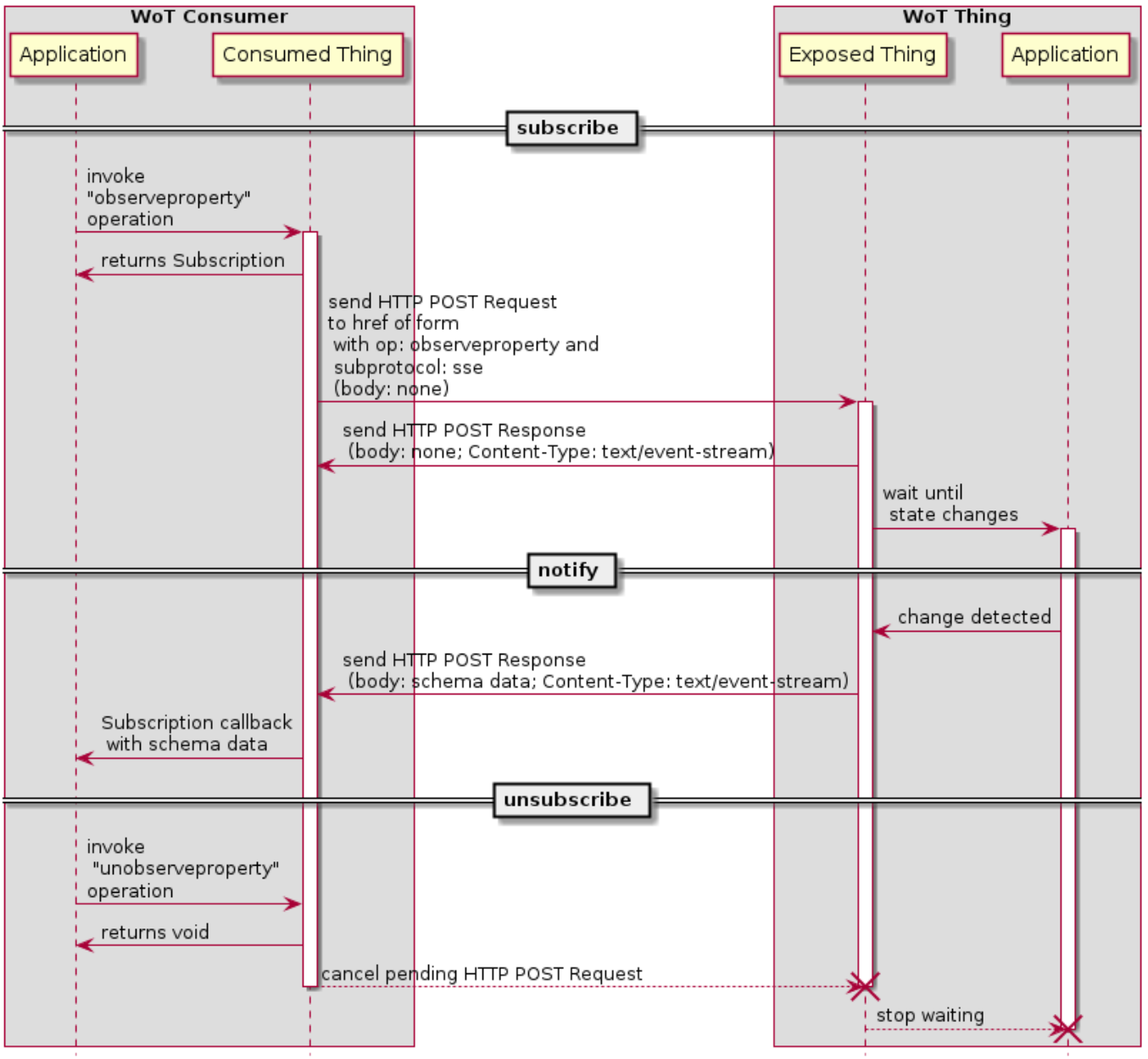
A.1.3 Observe property (HTTP binding with Long Polling subprotocol) §

The following sequence illustrates application and network transactions to implement the `observeproperty` operation with an HTTP protocol binding using the "longpolling" (Long Polling) subprotocol.



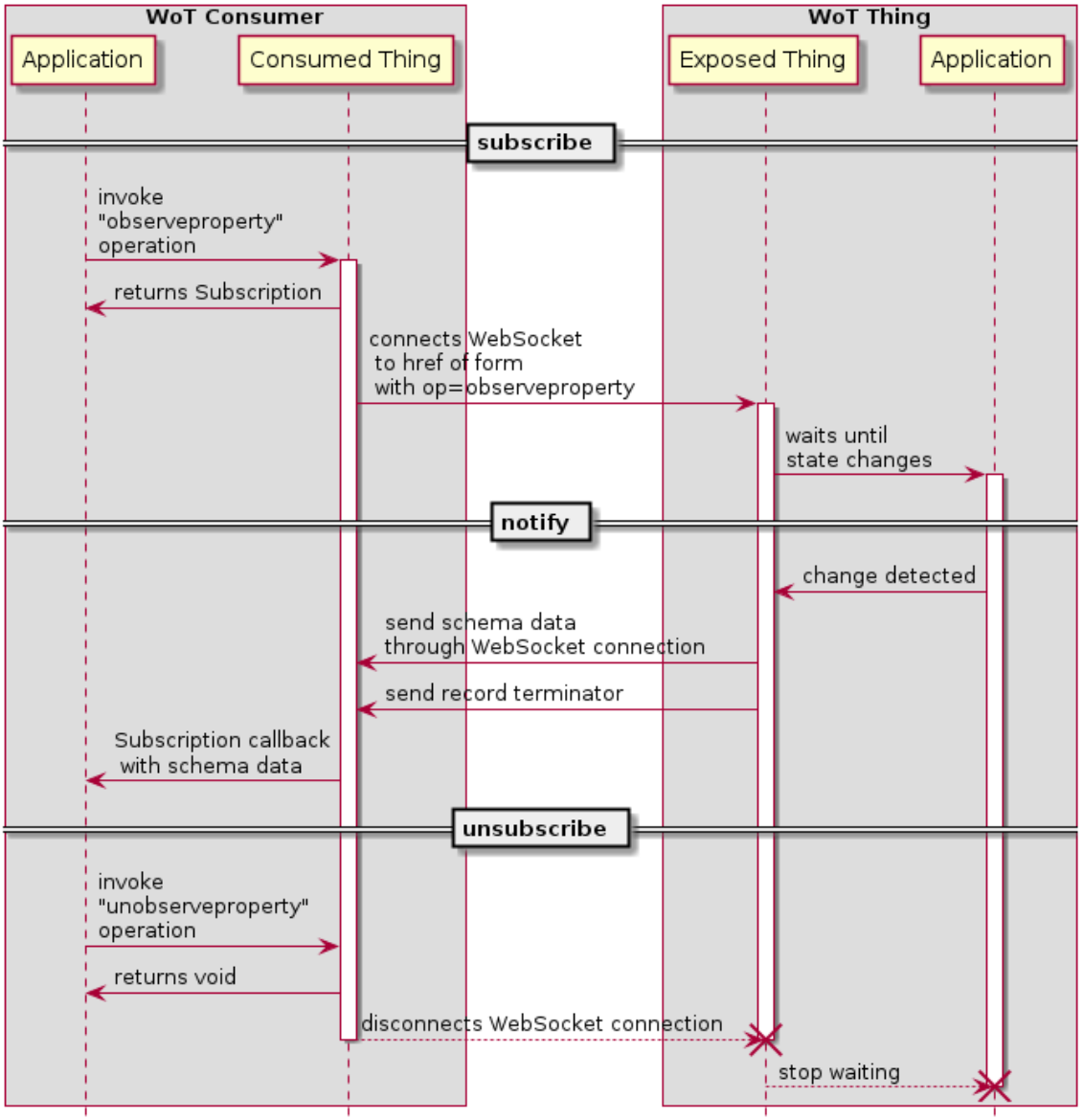
A.1.4 Observe property (HTTP binding with Server Sent Event subprotocol) §

The following sequence illustrates application and network transactions to implement the `observeproperty` operation with an HTTP protocol binding using the "sse" (Server Sent Event) subprotocol.



A.1.5 Observe property (HTTP binding with WebSocket subprotocol) §

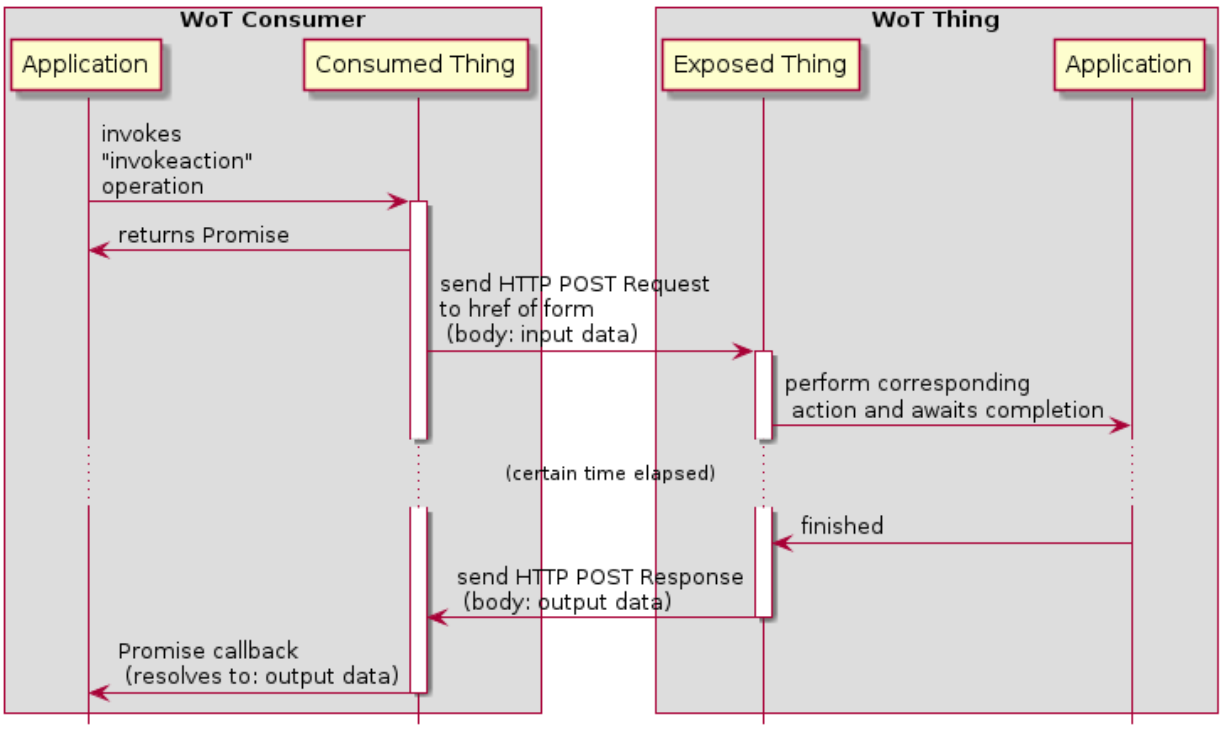
The following sequence illustrates application and network transactions to implement the `observeproperty` operation with an HTTP protocol binding using a WebSocket-based subprotocol.



A.2 Action Interactions §

A.2.1 Invoke action (HTTP binding) §

The following sequence illustrates application and network transactions to implement the `invokeaction` operation with an HTTP protocol binding, where the operation is synchronous and the response from the server is delayed until after the action completes.



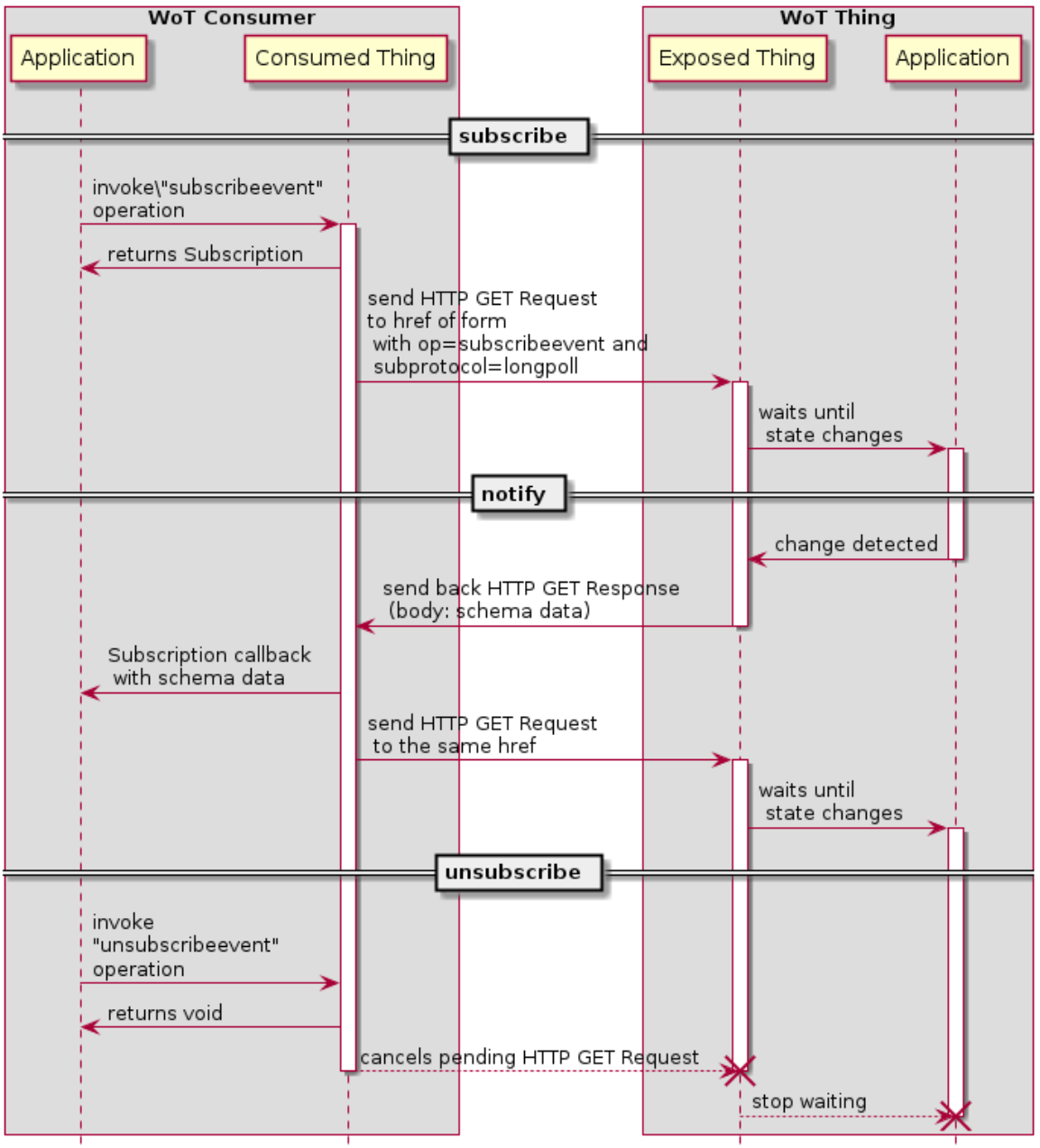
A.3 Event Interactions §

In the following, note that there is no explicit operation defined for event notification itself. The subprotocol used for notification is associated with the `subscribeevent` operation, and any necessary concrete protocol transactions are managed by the Protocol Binding subsystem.

There are also several subprotocols possible for event notification using WebSockets. The interaction diagrams show only one of several possible implementations.

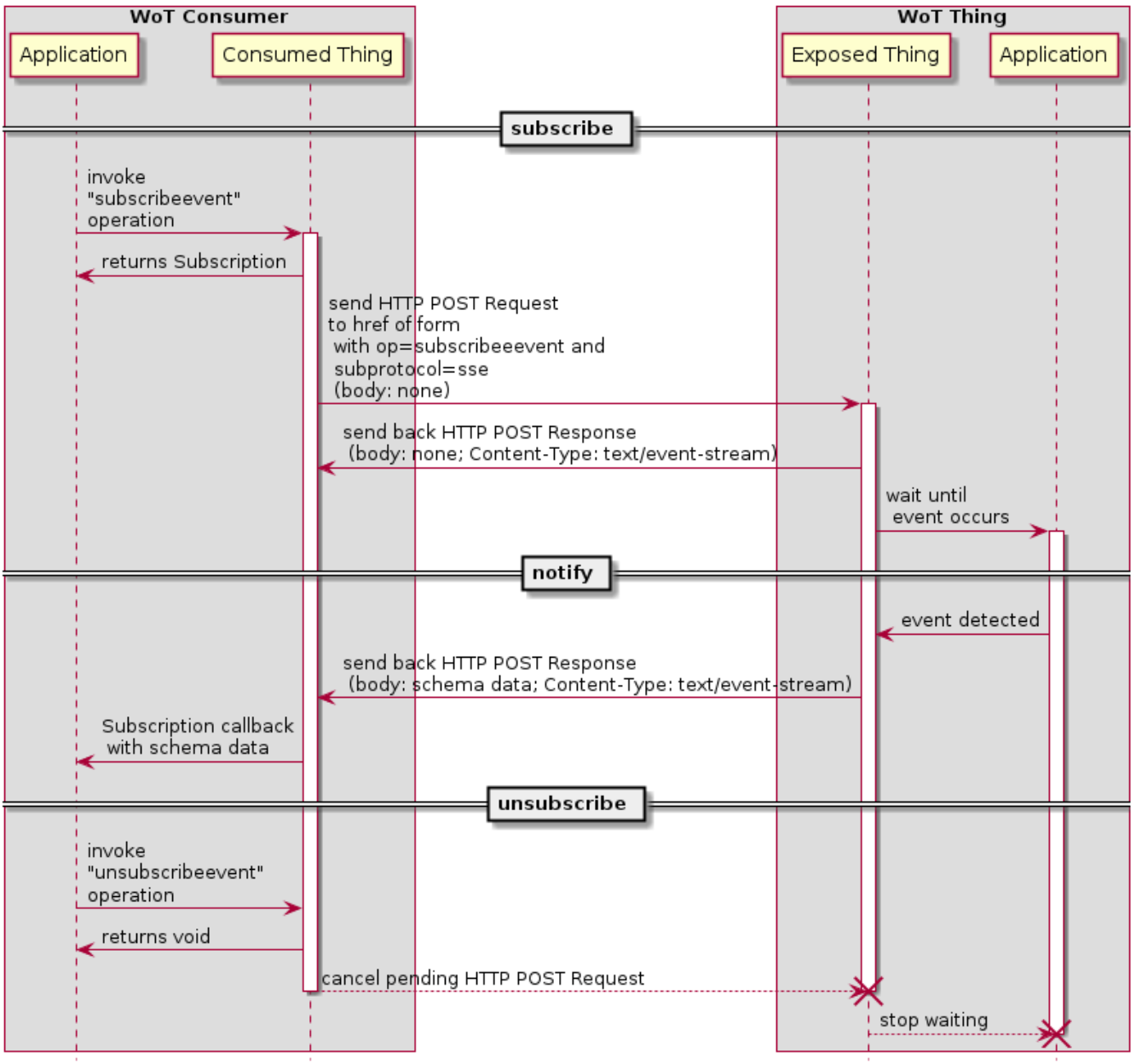
A.3.1 Subscribe, notify and unsubscribe event (HTTP binding with Long Polling subprotocol) §

The following sequence illustrates application and network transactions to implement the `subscribeevent` and `unsubscribeevent` operations with an HTTP protocol binding using the Long Polling subprotocol.



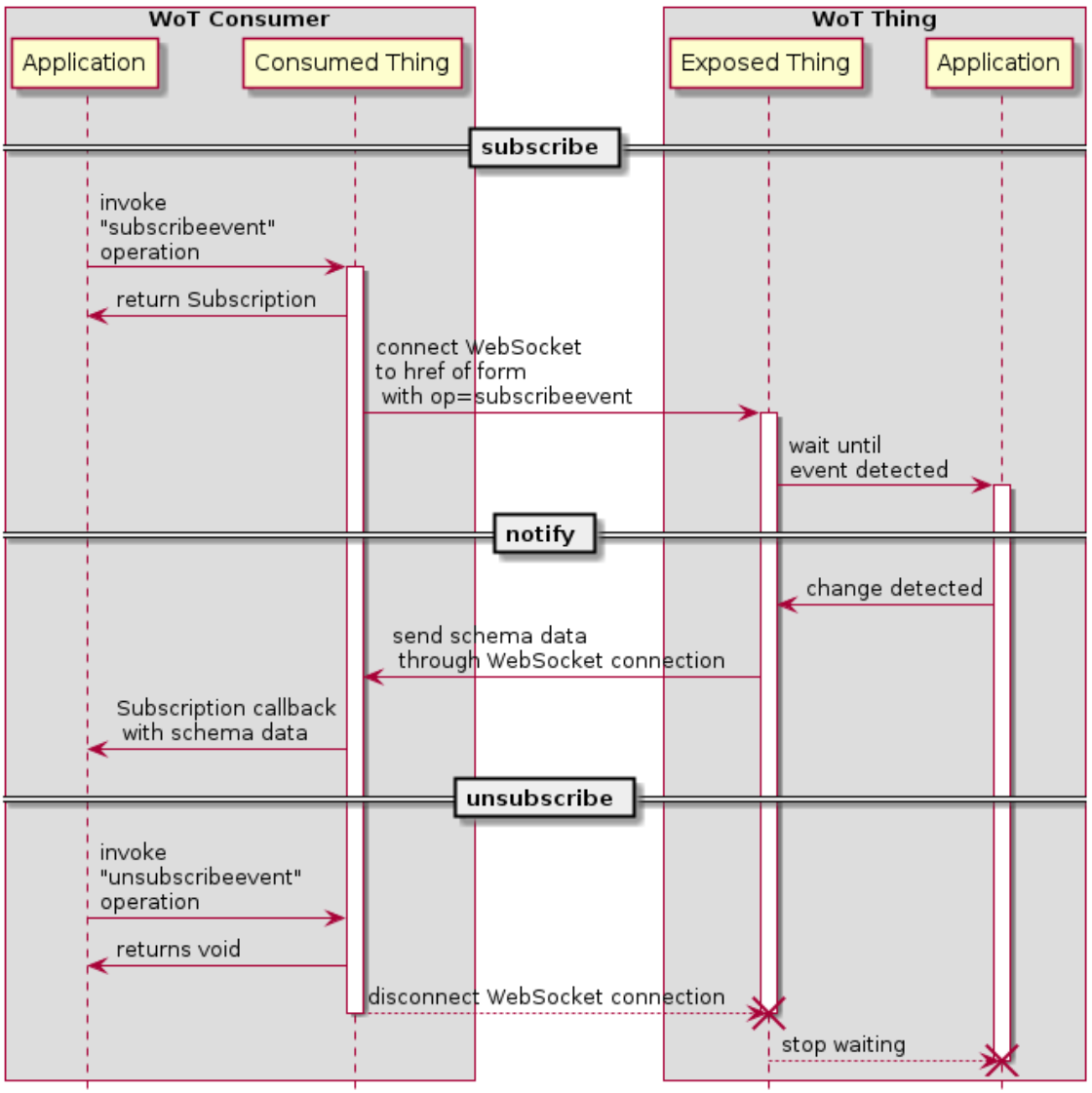
A.3.2 Subscribe, notify and unsubscribe event (HTTP binding with Server Sent Event subprotocol) §

The following sequence illustrates application and network transactions to implement the `subscribeevent` and `unsubscribeevent` operations with an HTTP protocol binding using the Server Sent Event subprotocol.



A.3.3 Subscribe, notify and unsubscribe event (HTTP binding with WebSocket subprotocol) §

The following sequence illustrates application and network transactions to implement the `subscribeevent` and `unsubscribeevent` operations with an HTTP protocol binding using a WebSocket subprotocol.



B. Acknowledgements §

Special thanks to all active participants of the W3C Web of Things Interest Group and Working Group for their technical input and suggestions that led to improvements to this document.

C. References §

C.1 Normative references §

[IANA-MEDIA-TYPES]
Media Types. IANA. URL: https://www.iana.org/assignments/media-types/

[iana-web-socket-registry]
IANA Registry for Websocket Subprotocols. IANA. 24 May 2019. URL: https://www.iana.org/assignments/websocket/websocket.xml#subprotocol-name

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC6741]

Identifier-Locator Network Protocol (ILNP) Engineering Considerations. RJ Atkinson; SN Bhatti. IETF. November 2012. Experimental. URL: https://tools.ietf.org/html/rfc6741

[RFC7641]

Observing Resources in the Constrained Application Protocol (CoAP). K. Hartke. IETF. September 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7641

[RFC8174]

Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. B. Leiba. IETF. May 2017. Best Current Practice. URL: https://tools.ietf.org/html/rfc8174

[WOT-THING-DESCRIPTION]

Web of Things (WoT) Thing Description. Sebastian Käbisch; Takuki Kamiya; Michael McCool; Victor Charpenay; Matthias Kovatsch. W3C. 9 April 2020. W3C Recommendation. URL: https://www.w3.org/TR/wot-thing-description/

C.2 Informative references §

[eventsource]

Server-Sent Events. Ian Hickson. W3C. 3 February 2015. W3C Recommendation. URL: https://www.w3.org/TR/eventsource/

[HTTP-in-RDF10]

HTTP Vocabulary in RDF 1.0. Johannes Koch; Carlos A. Velasco; Philip Ackermann. W3C. 2 February 2017. W3C Note. URL: https://www.w3.org/TR/HTTP-in-RDF10/

[json-ld11]

JSON-LD 1.1. Gregg Kellogg; Pierre-Antoine Champin; Dave Longley. W3C. 5 March 2020. W3C Candidate Recommendation. URL: https://www.w3.org/TR/json-ld11/

[json-schema]

JSON Schema: core definitions and terminology. K. Zyp. Internet Engineering Task Force (IETF). 31 January 2013. Internet-Draft. URL: https://tools.ietf.org/html/draft-zyp-json-schema

[LWM2M]

Lightweight M2M. Open Mobility Alliance. URL: https://www.omaspecworks.org/what-is-oma-specworks/iot/lightweight-m2m-lwm2m/

[MQTT]

MQTT Version 5.0. Andrew Banks; Ed Briggs; Ken Borgendale; Rahul Gupta. MQTT. 07 March 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[OCF]

OCF Specification. Open Connectivity Foundation. Last accessed: February 2020. URL: https://openconnectivity.org/developer/specifications/

[RFC3986]

Uniform Resource Identifier (URI): Generic Syntax. T. Berners-Lee; R. Fielding; L. Masinter. IETF. January 2005. Internet Standard. URL: https://tools.ietf.org/html/rfc3986

[RFC3987]

Internationalized Resource Identifiers (IRIs). M. Duerst; M. Suignard. IETF. January 2005. Proposed Standard. URL: https://tools.ietf.org/html/rfc3987

[RFC6570]

URI Template. J. Gregorio; R. Fielding; M. Hadley; M. Nottingham; D. Orchard. IETF. March 2012. Proposed Standard. URL: https://tools.ietf.org/html/rfc6570

[RFC7231]

Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. R. Fielding, Ed.; J. Reschke, Ed.. IETF. June 2014. Proposed Standard. URL: https://httpwg.org/specs/rfc7231.html

[RFC7252]

The Constrained Application Protocol (CoAP). Z. Shelby; K. Hartke; C. Bormann. IETF. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7252

[RFC8428]

Sensor Measurement Lists (SenML). C. Jennings; Z. Shelby; J. Arkko; A. Keranen; C. Bormann. IETF. August 2018. Proposed Standard. URL: https://tools.ietf.org/html/rfc8428

[WebSub]

WebSub. Julien Genestoux; Aaron Parecki. W3C. 23 January 2018. W3C Recommendation. URL: https://www.w3.org/TR/websub/

[WOT-ARCHITECTURE]

Web of Things (WoT) Architecture. Matthias Kovatsch; Ryuichi Matsukura; Michael Lagally; Toru Kawaguchi; Kunihiko Toumura; Kazuo Kajimoto. W3C. 9 April 2020. W3C Recommendation. URL: https://www.w3.org/TR/wot-architecture/

↥