WoT Scripting

TPAC 2020 update, October 22

Summary of changes

- Spec sweep for more modern ReSpec, referencing Web Platform etc.
- Discontinued *writeAllProperties()*.
- Resolve *DataSchema* vs *Forms*.
- Support for ***formIndex*** (consumer selecting desired form).
- Introduction of ***InteractionData*** (transformed into ***InteractionInput*** and ***InteractionOutput***) interface, supporting streams.
- Added content handling algorithms, added pictures.
- Improved almost all algorithms.
- Improving event handling in *ExposedThing*.
- Discussions on next API changes
  - Discovery API
  - JavaScript idioms and design patterns (mainly on subscription).

DataSchema vs Form: formIndex

Scripts can obtain the TD by using getThingDescription().

For certain *Form* content types, a DataSchema is defined, which helps parsing and validation.

Scripts can select a *Form* to be used by providing the *formIndex* preference in InteractionOptions. If that fails, the interaction fails (no fallback to the first *Form* that works). TODO: allow fallback in the algorithms?

How to handle interaction data in scripts and implementations:

- *InteractionInput* (used when the scripts pass data to the implementation)
- *InteractionOutput* (used when implementations provide data to scripts).

InteractionInput

```
typedef any DataSchemaValue;

typedef (ReadableStream or DataSchemaValue) InteractionInput;
```

DataSchemaValue is an ECMAScript value that is accepted for DataSchema defined in WoT-TD (i.e. *null*, *boolean*, *number*, *string*, *array*, or *object*).

ReadableStream is meant to be used for WoT Interactions that don't have a DataSchema in the Thing Description, only a Form's contentType that can be represented by a stream.

InteractionOutput

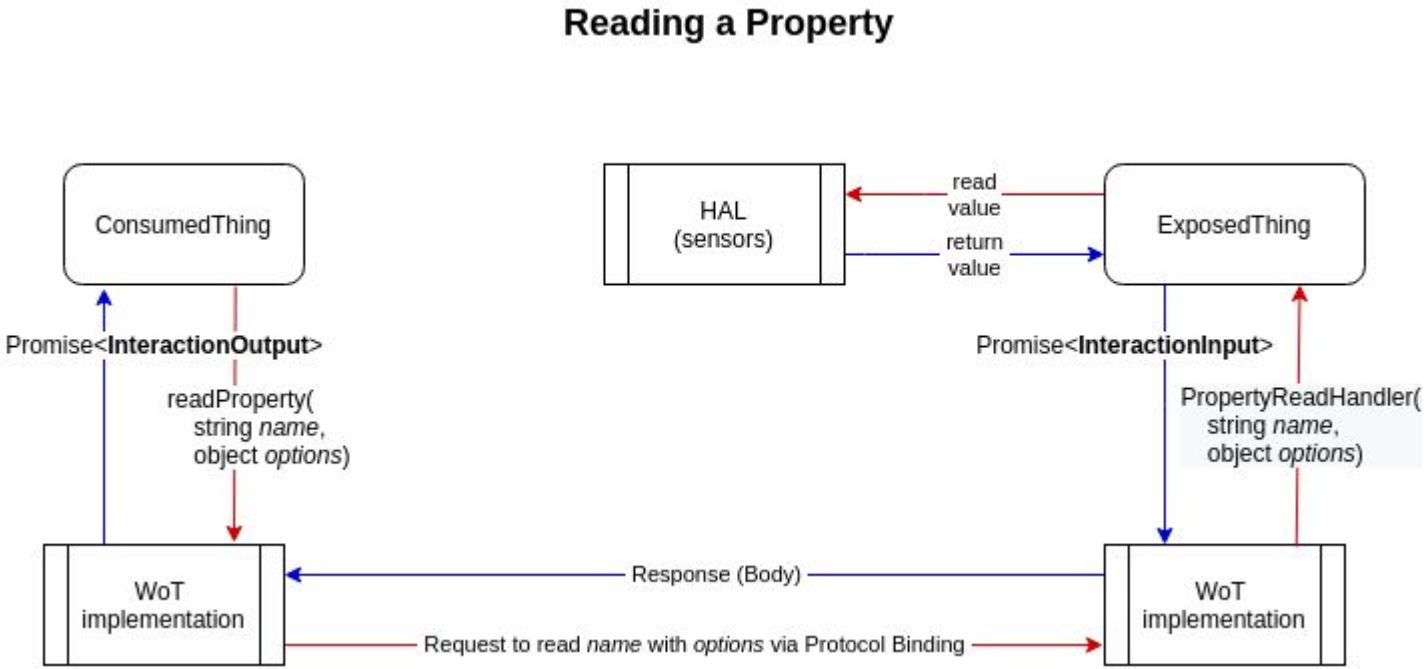
```
[SecureContext, Exposed=(Window,Worker)]
interface InteractionOutput {
  readonly attribute ReadableStream? data;
  readonly attribute boolean dataUsed;
  readonly attribute Form? form;
  readonly attribute DataSchema? schema;
  Promise<ArrayBuffer> arrayBuffer();
  Promise<any> value();
  };
```

`value()`: convenience function. If fails, scripts can revert to reading the stream.

See example with value(), fallback example for a picture and stream example for a video.

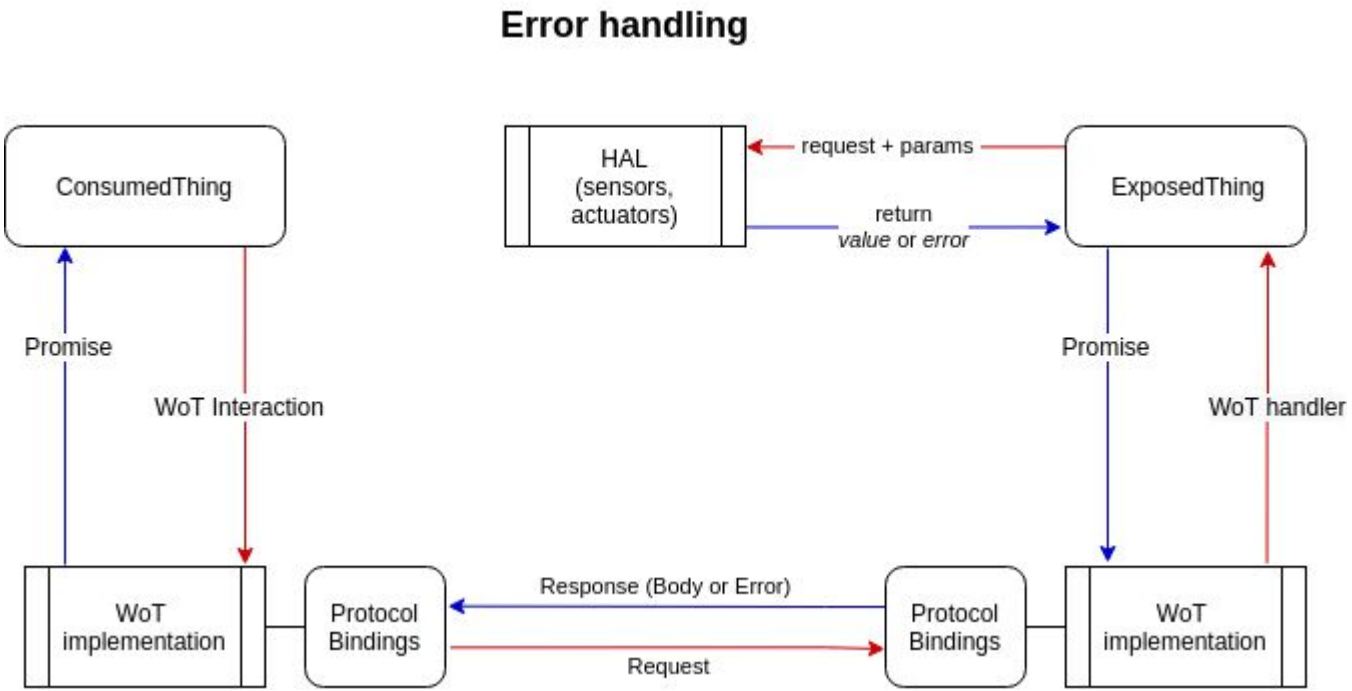
Data flow in the API

For instance, for reading with HTTP/S (see the others in the spec, [Using InteractionInput and InteractionOutput]( ).



Issue: error handling

See this section.



ExposedThing: improved event handling

```
ExposedThing setEventSubscribeHandler(DOMString name, EventSubscriptionHandler handler);
ExposedThing setEventUnsubscribeHandler(DOMString name, EventSubscriptionHandler handler);
ExposedThing setEventHandler(DOMString name, EventListenerHandler eventHandler);
Promise<undefined> emitEvent(DOMString name, InteractionInput data);

callback EventSubscriptionHandler = Promise<undefined>(optional InteractionOptions options = null);
callback EventListenerHandler = Promise<InteractionInput>();
```

Notice using the same *InteractionInput* interface in the server side hooks.

Discovery API

```
partial namespace WOT {
  ThingDiscovery discover(
      optional ThingFilter filter = null);
  };

dictionary ThingFilter {
  (DiscoveryMethod or DOMString) method = "any";
  USVString? url;
  USVString? query;
  object? fragment;
  };

[SecureContext, Exposed=(Window,Worker)]
interface ThingDiscovery {
  constructor(optional ThingFilter filter = null);
  readonly attribute ThingFilter? filter;
  readonly attribute boolean active;
  readonly attribute boolean done;
  readonly attribute Error? error;
  undefined start();
  Promise<ThingDescription> next();
  undefined stop();
  };
```

See Examples.

```
let discoveryFilter = {
  method: "directory",
  url: "http://directory.wotservice.org"
};
let discovery = new ThingDiscovery(discoveryFilter);
setTimeout( () => {
    discovery.stop();
    console.log("Discovery stopped after timeout.");
  },
  3000);
do {
  let td = await discovery.next();
  console.log("Found Thing Description for " + td.title);
  let thing = new ConsumedThing(td);
  console.log("Thing name: " +
thing.getThingDescription().title);
} while (!discovery.done);
if (discovery.error) {
  console.log("Discovery stopped because of an error: " +
error.message);
}
```

Next: Discovery API alignment and implementation

- How to spec the 1st phase of discovery? (the current API is phase 2 only)
  - Phase 1 considered a provisioning issue, managed by the runtime.
  - Use case when a script has to manage Phase 1 discovery?
  - Separate API entry point for Phase 1 discovery?
- Add “`direct`” to *DiscoveryMethods*? Currently supported by default.
- Validate API design for iteration over discovered items.
  - The current design can accommodate arbitrary buffering/paging schemes.
  - Similar pattern used in IndexedDB/IDBCursor. Here it’s much simpler, using *next()* that provides the next TD object. Should that be changed to the URL of the next fetchable TD? (That would allow handling TD fetch via a Response object using ReadableStream, for huge TDs. Has been discussed and rejected for this version, for convenience. Plus, a JS object can be exposed with properties provided on request.)

Next: script management, provisioning, runtime

Node-wot supports basic script management
(e.g., list available things, run script/thing).

Ongoing work about Edge Workers / IoT orchestration.

Packaging options:

- Script (currently possible via node-wot)
- WASM module (future)
- Container (future, but already possible when including node-wot).