

WoT Scripting status

W3C WoT F2F, March 2021

Spec status

- Scripting [presentation on TPAC 2020](#) (interaction modeling etc)
- Summary of main PRs since then:
 - Improved data handling algorithms
 - ExposedThing creation from an init object (that maps to partial TD, Thing Model or Thing)
 - TD validation both on consumption and ExposedThing creation
- Summary of main outstanding issues
 - Validation of mandatory fields validation (e.g. *title*, ...)
 - Handling very large TDs vs constrained devices. See [#309](#)
 - Issues that need group discussion
 - [TD TF] TD generation from init could be more standardized. Best practices?
 - [Bindings TF] how to generate Forms?
 - [Discovery TF] Discovery API sync
 - A separate Scripting API for provisioning (incl. 1st phase discovery), script (module) management
(Note: *node-wot* has some basic script management)

InteractionInput and InteractionOutput

Modeling interaction data:

- Primarily driven by *contentType* and *DataSchema*
- When both are known, implementations can parse a *value*
- When only *contentType* is known, the app has to parse (on reads) or provide (on writes) a *ReadableStream*.

Interfaces:

- *InteractionInput* is either a JSON value or a *ReadableStream* object.
- *InteractionOutput* is a data interface that models this by conflating a *value()* function with a [Body mixin](#) (see next slide).

Data flows: <https://www.w3.org/TR/wot-scripting-api/#using-interactioninput-and-interactionoutput>

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();
};
```

See [code examples](#).

See rationale and discussion here: [#292](#), [#306](#)

Note: in HTTP binding (browser), WoT Scripting can be trivially implemented on top of the Fetch API.

Body mixin (from Fetch API)

```
interface mixin Body {
  readonly attribute ReadableStream? body;
  readonly attribute boolean bodyUsed;
  [NewObject] Promise<ArrayBuffer> arrayBuffer();
  [NewObject] Promise<Blob> blob();
  [NewObject] Promise<FormData> formData();
  [NewObject] Promise<any> json();
  [NewObject] Promise<USVString> text();
};
```

ConsumedThing

ConsumedThing changes

```
[SecureContext, Exposed=(Window,Worker)]
interface ConsumedThing {
  constructor(ThingDescription td);
  Promise<InteractionOutput> readProperty(DOMString propertyName,
    optional InteractionOptions options = null);
  Promise<PropertyReadMap> readAllProperties(
    optional InteractionOptions options = null);
  Promise<PropertyReadMap> readMultipleProperties(
    sequence<DOMString> propertyNames,
    optional InteractionOptions options = null);
  Promise<undefined> writeProperty(DOMString propertyName,
    InteractionInput value,
    optional InteractionOptions options = null);
  Promise<undefined> writeMultipleProperties(
    PropertyWriteMap valueMap,
    optional InteractionOptions options = null);
  Promise<InteractionOutput> invokeAction(DOMString actionName,
    optional InteractionInput params = null,
    optional InteractionOptions options = null);
  Promise<Subscription> observeProperty(DOMString name,
    InteractionListener listener,
    optional ErrorListener onerror,
    optional InteractionOptions options = null);
  Promise<Subscription> subscribeEvent(DOMString name,
    InteractionListener listener,
    optional ErrorListener onerror,
    optional InteractionOptions options = null);
  ThingDescription getThingDescription();};
```

```
dictionary InteractionOptions {
  unsigned long formIndex;
  object uriVariables;
  any data;
};
```

```
[SecureContext, Exposed=(Window,Worker)]
interface Subscription {
  readonly attribute boolean active;
  Promise<undefined> stop(
    optional InteractionOptions options = null);
};
```

```
[SecureContext, Exposed=(Window,Worker)]
interface PropertyReadMap {
  readonly maplike<DOMString, InteractionOutput>;
};
```

```
[SecureContext, Exposed=(Window,Worker)]
interface PropertyWriteMap {
  readonly maplike<DOMString, InteractionInput>;
};
```

```
callback InteractionListener =
  undefined(InteractionOutput data);
```

```
callback ErrorListener = undefined(Error error);
```

Thing Description validation when consuming

Current status: uses JSON Schema algorithm

However, some rules defined in the specification document are not covered. The validation playground does additional checkings. Some of those checkings are not functional (e.g., checks whether all titles and descriptions have the same language fields).

Functional validation is enough?

ExposedThing

ExposedThing developer use case

To create a new *ExposedThing*, a developer needs to use tools or scripts outside of WoT Scripting API as well:

1. Fetch Thing Models (TM) or Thing Descriptions (TD).
Resolve/fetch all dependencies, produce a *flat* TM.
2. Optionally, use text or TD tools to assemble a desired flat TM for the new *ExposedThing*.
3. Create a combined JSON object out steps 1 and 2 (in case of a single fetch, it's trivial).
4. Write a WoT script and feed that JSON object (or a ConsumedThing object) to *produce()* or the *ExposedThing* constructor. It will validate the input object and create a new *ExposedThing* from it.
5. Add SW request handlers for each request type: property read/write, action, event, event subscribe/unsubscribe, property observe/unobserve.
6. Call *expose()*, which will generate the bindings and start to serve requests to this *ExposedThing* (including starting any needed dependent service, request dispatching, parsing etc.).
7. A TD representation is generated (or returned) when *getThingDescription()* is called.

ExposedThing changes

- ExposedThing does not inherit ConsumedThing any more. They belong to separated conformance classes now, i.e. they can be implemented separately.
- Since the TD spec does not specify how to *create* TDs, the Scripting TF had to come up with algorithms for initialization, default values, validation.
 - An init dictionary is used for providing the interaction descriptions.
 - Applications specify the request handler functions.
 - The source for the init dictionary can be a TD, or Thing Model, or ConsumedThing.
- Open issues:
 - Using Thing Models (single and multiple) as inputs for ExposedThing creation.
[#304](#)

ExposedThing

```
[SecureContext, Exposed=(Window,Worker)]
interface ExposedThing extends ConsumedThing {
    ExposedThing setPropertyReadHandler(DOMString name,
        PropertyReadHandler handler);
    ExposedThing setPropertyWriteHandler(DOMString name,
        PropertyWriteHandler handler);
    ExposedThing setPropertyObserveHandler(DOMString name,
        PropertyReadHandler handler);
    ExposedThing setPropertyUnobserveHandler(DOMString name,
        PropertyReadHandler handler);
    Promise<undefined> emitPropertyChange(DOMString name);

    ExposedThing setActionHandler(DOMString name,
        ActionHandler action);
    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);
    Promise<undefined> expose();
    Promise<undefined> destroy();
    ThingDescription getThingDescription();};
```

```
callback PropertyReadHandler =
    Promise<any>(  
    optional InteractionOptions options = null);

callback PropertyWriteHandler =
    Promise<undefined>(  
    InteractionOutput value,  
    optional InteractionOptions options = null);

callback ActionHandler =
    Promise<InteractionInput>(  
    InteractionOutput params,  
    optional InteractionOptions options = null);

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

callback EventListenerHandler =
    Promise<InteractionInput>();
```

New dictionary: ExposedThingInit

A JSON object that is used as initialization dictionary for an ExposedThing object. Currently, we describe it as runtime representation of a **PartialTD** or **ThingModel**.

Algorithms:

- Expansion:
 - Define the steps to create a valid ThingDescription from an ExposedThingInit object
 - Uses runtime information to fill missing fields such as forms, securitySchemas, hrefs
 - See: <https://w3c.github.io/wot-scripting-api/#expand-an-exposedthinginit>
- Validation
 - Define a transformation algorithm that processes the latest TD json schema and removes mandatory fields.
 - Similar to Thing Model schema, but it does not allow place holders
 - See <https://w3c.github.io/wot-scripting-api/#validating-an-exposedthinginit>

How to generate forms

In the current expand ExposedThingInit algorithm forms generation is left open to specific runtime logic.

It might be beneficial to describe **default form generation** process in the Protocol Bindings document, so that each runtime creates the same default forms given a particular protocol binding.

Should *title* be mandatory?

When producing an `ExposedThing`, *title* might be the minimum viable information that a developer must supply to create a web Thing.

Pro:

- Developers are encouraged to think twice before creating an `ExposedThing`. Titles might be used in UIs and therefore should be human-readable
- Automatically generated values can still be used (i.e., developers define their own generation logic)

Cons:

- `ExposedThingInit` is not a `PartialTD` (title is not mandatory)
- User defined generated values might be difficult to standardize

See: <https://github.com/w3c/wot-scripting-api/issues/300>

Discovery

Discovery API

- **The Discovery TF could have input/suggestions for the Scripting API.**
- The current Discovery API in WoT Scripting:
 - Only 2nd phase in Discovery is represented
 - A separate conformance class: optional to implement the *discover()* method.
 - Discovery in various IoT protocols might be very diverse, but the API can be simple.
 - Basically a generic observe pattern:
 - send a request with options (filters)
 - wait for replies
 - until stopped or otherwise finished.
 - Filters may contain TD-related and protocol-related clauses.
 - Protocol-related filters:
 - Discovery method (local, directory, multicast, any) - disputed, currently *any* is assumed
 - TD-related filters:
 - TD fragment: an object with property names (*existence* of properties is used)
 - SPARQL query string (no support for now, theoretical feature in the API).
- Discussion for alternative design: [#222](#)

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.thingDescription.title);
} while (!discovery.done);
if (discovery.error) {
  console.log("Discovery stopped because of an error: " +
error.message);
}
```

Next: Discovery API alignment

- 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?
- Should we use [URL object](#) or *USVString*?
- Add “direct” to *DiscoveryMethods*? Currently supported by default
 - When a *url* is specified without the *method* being “directory”
- 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.

TPAC 2020 “Next”: script management, provisioning, runtime

- node-wot supports basic script management (e.g., list available things, run script/thing).
- Ongoing work on Edge Workers / IoT orchestration.

Packaging options:

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

Note: WASM modules perhaps the [Transferable](#) compute units of the future

(e.g. transferable through workers' *postMessage()*..)

Issues for “next iteration”

See [issues](#):

#309 [Handle ThingDescriptions as streams](#) for next iteration

#304 [Produce ExposedThing from a ThingModel instance](#) for next iteration

#303 [Separate ExposedThing API](#) for next iteration

#299 [Chose a particular security schema for an ExposedThing](#) for next iteration

#298 [Requirements for Managment APIs](#) for next iteration

#274 [Add 'once' to subscription options](#) enhancement for next iteration

#192 [Discuss the modularization of WoT scripts](#) Runtime Script Management for next iteration use case

#190 [Create testing plan](#) for next iteration

#107 [Very high frequency updates](#) enhancement for next iteration use case




Implementations

Thingweb *node-wot*

- Current release v0.7.5
 - o Implements the *old* API (see changes described before)
 - o Live things available, see <http://plugfest.thingweb.io>
- Upcoming release v0.8.x
 - o Implements the *new* API
 - o See [v0.8.x branch](#) on GitHub
- No support yet for Discovery
- Current support for management API (Servient)
 - o List available things
 - o Start script (expose thing)
 - o Shutdown Servient, set log level, ...
 - o Note: improvements possible, e.g., decide which modules to be loaded







Thingweb *node-wot* - Features

Protocol Support

- HTTP/HTTPS 
- CoAP/ CoAPS 
- MQTT 
- Websocket (Server only)
- OPC-UA (Client only)
- NETCONF (Client only)
- Modbus (Client only)

Note: other protocols can be easily added by implementing `ProtocolClient`, `ProtocolClientFactory`, and `ProtocolServer` interface.

MediaType Support

- JSON 
- Text (HTML, CSS, XML, SVG) 
- Base64 (PNG, JPEG, GIF) 
- Octet stream 
- CBOR 
- EXI 

Note: other media types can be easily added by implementing `ContentCodec` interface.

Thingweb *node-wot* - NPM Statistics

E.g., @node-wot/core ([last 4 years](#))

Downloads per month

