Web of Things Scripting API

Zoltan Kis, Intel
Daniel Peintner, Siemens

June 2019, München
WoT Concepts: Web vs WoT analogy

WoT is a framework to describe and integrate IoT platforms through Web technologies used in addressing, discovery, access control, data transfer, scripting.

| Web page | → Thing |
| URL | → URI |
| HTTP | → HTTP, CoAP, BLE, WS |
| HTML | → Thing Description |
| ECMAScript | → **WoT Script** |
| Web search | → Discovery |
| Served page | → Exposed Thing |
| Rendered page | → Consumed Thing |
Why a Scripting API?

• Scripting has transformed the Web
  – Marc Andreessen, the founder Netscape, “believed that HTML needed a ‘glue language’ that was easy to use by Web designers and part-time programmers to assemble components such as images and plugins, where the code could be written directly in the Web page markup.”
  – Brendan Eich wrote Java-inspired Mocha in 10 days in May 1995
  – Later called LiveScript, then JavaScript, then standardized as ECMAScript
  – 10.7 million JavaScript developers in 2018 (out of 23 million)

• WoT describes and integrates IoT platforms through Web technologies
  – addressing, discovery, access control, data transfer, and
  – scripting.
What does a WoT script do?

1. Implement behaviour
2. Control Thing interactions

Alternatives:
- Native implementations
- Visual rules (e.g. WebThings)
- Flows (e.g. Node-RED)
Coexisting Scripting APIs

- A Thing implemented with one scripting language interacts seamlessly with other Things.
- As long as they both expose a TD and implement protocol bindings to a network facing API.
Some benefits

- Simplify application development
- Portable across vendors, if all implement the API
- Language constructs and idioms are different, but all should implement:
  - consume/discover Things, produce Things,
  - read/write Properties
  - invoke Actions
  - react to Events.
Opportunities for Scripting

Use cases

Cloud Mirrors

Semantic metadata (Thing Description)

Web integration

Device Shadow

Application

Scripting API

Interaction Model

Protocol Bindings

Remote access

Direct Thing-to-Thing

Existing Device

⇒ Thing

Complementing existing devices and platforms

(Thing Description, Binding Templates)

Local Hubs

Standardized APIs for portable application logic (Scripting API)

Web Browser

Application

Scripting API

Interaction Model

Protocol Bindings

Direct Thing-to-Thing

Powerful and constrained Things

Virtual Thing

Application

Scripting API

Interaction Model

Protocol Bindings

Use cases
WoT Runtime

Script Execution Context (SEC)
- Single event loop
- Call stack, callback queue
- Runs a single script
- Serialized execution
- Hosts multiple Thing instances

SEC Definitions
- For ECMAScript (in browsers):
  - Multiple SEC
  - One SEC executed at a time
- For WoT: more like in Node.js
  - SEC may be sandboxed
  - SEC may run in separate containers
  $\Rightarrow$ SECs may run in parallel

WoT Runtime
- Process hosting all structures
  - WoT API objects
  - Bindings plugin host
  - Script Manager Thing
  - System Things
- May spawn new SECs
- Security enforcement point (trusted)

Scripting API: Discovery, Client, Server

WoT Runtime (Agent)
- Interactions implementation
- Security Configurations

Scheduling
- System API bindings
- Special Things

Servient architecture

Scripts can get to a device in the following ways:
1. Provisioned (e.g. flashed or copied to the device)
2. Using a built-in Thing that implements a Script Manager interface
3. By consuming a TD that contains a link to a script (idea for future development)
WoT Stack

Processes
- WoT Runtime
- Script execution contexts
- Bindings, System adapters
- System APIs / OS Kernel

Special Things
- Script Manager Thing
- System Access Things
  - expose API objects
  - have TDs ⇒ discoverable (introspection)

System APIs
- Protocol stack/services (CoAP, HTTP, BLE, OCF, ...)
- file system
- secure storage
- local sensors
- Local HW API
WoT Scripting API Standardization

- WOT API: client, server, discovery
- Thing Client API: interact with Things
- Thing Server API: expose local Things
- Examples
Scripting API standardization

- In the WoT IG
  - Proposals
  - Discussed in weekly calls
  - Tested on plug-fests

- In the WoT WG
  - GitHub repository
  - Proposals in GitHub issues
  - Several versions:
    - Editor’s Draft (ED)
    - First Public Working Draft (FPWD)
    - Working Draft (WD)
    - WG Note

Initial ED: February 2017
FPWD: 14.09.2017
WD1: 05.04.2018
WD2: 29.11.2018
WG Note: June 2019 (work can continue)
Reference implementation: node-wot
# Approaches to the Scripting API

## No externally exposed API (only WoT network interface)

- `lock = WoT.consume('https://td.my.com/lock');`
- `print(lock.status);`
- `lock.open();`

## Simple API

- `lock = WoT.consume('https://td.my.com/lock');`
- `print(lock.status);`
- `lock.open();`

## Current API (based on the TD spec)

- `lock = WoT.consume('https://td.my.com/lock');`
- `print(lock.readProperty('status'));`
- `lock.invokeAction('open');`

## A WoT gateway can encapsulate other IoT deployments:
- presents a REST-ful API towards clients
- implements IoT protocols towards IoT deployments

## Design

**Thing** = object
**Thing Property** = object property
**Thing Action** = object method
**Thing Event** = event
**WoT API object** = lifecycle methods

**Thing Description** = data object
**Thing** = TD instance + API methods
**WoT API object** = lifecycle methods
Web of Things (WoT) Scripting API

W3C Editor's Draft 12 April 2019

This version:
https://w3c.github.io/wot-scripting-api/

Latest published version:
https://www.w3.org/TR/wot-scripting-api/

Latest editor's draft:
https://w3c.github.io/wot-scripting-api/

Editors:
Zoltan Kis (Intel)
Daniel Peatiner (Siemens AG)
Johannes Hunt (Former Editor, when at Siemens AG)
Kazuaki Nimura (Former Editor, at Fujitsu Ltd.)

Repository:
On GitHub
File a bug

Contributors:
Contributors on GitHub

Copyright © 2017-2019 W3C® (MIT, ERCIM, Keio, Beihang), W3C liability, trademark and permissive document license rules apply.

Abstract

The key Web of Things (WoT) concepts are described in the WoT Architecture document. The Web of Things is made of entities (Things) that can describe their capabilities in a machine-interpretable format, the Thing Description (TD) and expose these capabilities through the WoT Interface, that is, network interactions modeled as Properties (for reading and writing values), Actions (to execute remote procedures with or without return values) and Events (for signaling notifications).

Scripting is an optional “convenience” building block in WoT and it is typically used in gateways that are able to run a WoT Runtime and script management, providing a convenient way to extend WoT support to new types of endpoints and implement WoT applications such as Thing Directory.

This specification describes a programming interface representing the WoT Interface that allows scripts to discover and operate Things and to expose locally defined Things characterized by WoT Interactions specified by a script.
interface WOT {
  ConsumedThing consume(ThingDescription td);
  ExposedThing produce(ThingDescription td);
  ThingDiscovery discover(optional ThingFilter filter);
};

typedef (USVString or object) ThingDescription;
typedef object ThingInstance;

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

typedef DOMString DiscoveryMethod;
dictionary ThingFilter {
  (DiscoveryMethod or DOMString) method = "any";
  USVString? url;
  USVString? query;
  object? fragment;
};
callback WotListener = void(any data);
dictionary InteractionOptions {
  object uriVariables;
};

[Constructor(ThingInstance instance), SecureContext, Exposed=(Window, Worker)]
interface ConsumedThing: EventTarget {
  Promise<any> readProperty(DOMString propertyName, optional InteractionOptions options);
  Promise<object> readAllProperties(optional InteractionOptions options);
  Promise<object> readMultipleProperties(sequence<DOMString> propertyNames, optional InteractionOptions options);
  Promise<void> writeProperty(DOMString propertyName, any value, optional InteractionOptions options);
  Promise<void> writeMultipleProperties(object valueMap, optional InteractionOptions options);
  Promise<any> invokeAction(DOMString actionName, optional any params, optional InteractionOptions options);
  Promise<void> subscribeProperty(DOMString name, WotListener listener);
  Promise<void> unsubscribeProperty(DOMString name);
  Promise<void> subscribeEvent(DOMString name, WotListener listener);
  Promise<void> unsubscribeEvent(DOMString name);
  readonly attribute ThingInstance instance;
  attribute EventHandler onchange;
};

[Constructor(ThingInstance instance), SecureContext, Exposed=(Window, Worker)]
interface ExposedThing: ConsumedThing {
  ExposedThing setPropertyReadHandler(DOMString name, PropertyReadHandler readHandler);
  ExposedThing setPropertyWriteHandler(DOMString name, PropertyWriteHandler writeHandler);
  ExposedThing setActionHandler(DOMString name, ActionHandler action);
  void emitEvent(DOMString name, any data);
  Promise<void> expose();
  Promise<void> destroy();
};

callback PropertyReadHandler = Promise<any>();
callback PropertyWriteHandler = Promise<void>(any value);
callback ActionHandler = Promise<any>(any parameters);
Scripting API spec highlights

(what else beyond fashion, i.e. usable in other language implementations?)

Recipe for...

• Scripting use cases
• Main API methods and algorithms
  – consume, produce, discover
  – read, write, invoke, subscribe, unsubscribe
  – define handlers for requests made to exposed Things
• How to instantiate an API object from a TD
• Security and privacy considerations
WoT object

```java
[SecureContext, Exposed=(Window, Worker)]
interface WOT {
    ConsumedThing consume(ThingDescription td);
    ExposedThing produce(ThingDescription td);
    ThingDiscovery discover(optional ThingFilter filter);
};

typedef (USVString or object) ThingDescription;
typedef object ThingInstance;
```

To create and expose a Thing, we need a TD.
Fetching and consuming a TD

```javascript
try {
    let res = await fetch('https://tds.mythings.biz/sensor11');
    // ... additional checks possible on res.headers
    let td = await res.json(); // could also be res.text()
    let thing = WOT.consume(td);
    console.log("Thing name: " + thing.instance.name);
} catch (err) {
    console.log("Fetching TD failed", err.message);
}
```

To create and expose a Thing, we need a TD. We use an external API to fetch a TD.
Client API: ConsumedThing

```javascript
[Constructor(ThingInstance instance), SecureContext, Exposed=(Window, Worker)]

interface ConsumedThing {
    Promise<any> readProperty(DOMString propertyName, optional InteractionOptions options);
    Promise<object> readAllProperties(optional InteractionOptions options);
    Promise<object> readMultipleProperties(sequence<DOMString> propertyNames, optional InteractionOptions options);
    Promise<void> writeProperty(DOMString propertyName, any value, optional InteractionOptions options);
    Promise<void> writeMultipleProperties(object valueMap, optional InteractionOptions options);
    Promise<any> invokeAction(DOMString actionName, optional any params, optional InteractionOptions options);
    Promise<void> subscribeProperty(DOMString name, WotListener listener);
    Promise<void> unsubscribeProperty(DOMString name);
    Promise<void> subscribeEvent(DOMString name, WotListener listener);
    Promise<void> unsubscribeEvent(DOMString name);
    readonly attribute ThingInstance instance;
}

callback WotListener = void(any data);
dictionary InteractionOptions {
    object uriVariables;
};
```

Once a Thing is found, scripts can
- observe properties and events
- change it using properties and actions.

The client needs access rights (provisioning is out of scope).
try {
  let res = await fetch("https://tds.mythings.org/sensor11");
  let td = res.json();
  let thing = new ConsumedThing(td);

  await thing.subscribeProperty("temperature", value => {
    console.log("Temperature changed to: " + value);
  });

  await thing.subscribeEvent("ready", eventData => {
    console.log("Ready; index: " + eventData);
    await thing.invokeAction("startMeasurement",
      { units: "Celsius" });
    console.log("Measurement started.");
  });
}
catch(e) {
  console.log("Error: " + error.message);
}

setTimeout(() => {
  console.log("Temperature: " + 
    await thing.readProperty("temperature");
  await thing.unsubscribe("ready");
  console.log("Unsubscribed from the ‘ready’ event.");
}, 10000);
Server API: ExposedThing

```javascript
[Constructor(ThingInstance instance), SecureContext, Exposed=(Window,Worker)]

interface ExposedThing: ConsumedThing {
  ExposedThing setPropertyReadHandler(DOMString name, PropertyReadHandler readHandler);
  ExposedThing setPropertyWriteHandler(DOMString name, PropertyWriteHandler writeHandler);
  ExposedThing setActionHandler(DOMString name, ActionHandler action);
  void emitEvent(DOMString name, any data);
  Promise<void> expose();
  Promise<void> destroy();
};
```

callback PropertyReadHandler = Promise<any>();
callback PropertyWriteHandler = Promise<void>(any value);
callback ActionHandler = Promise<any>(any parameters);

A server Thing can
- programmatically create a TD
- define behavior for client requests:
  - get/set Property
  - invoke Action
  - observe Events.
ExposedThing bad example

// Typically a TD is obtained from somewhere, but let's write it now.
let thingDescription = '{
  "name": "mySensor",
  "@context": [ "http://www.w3.org/ns/td",
    "https://w3c.github.io/wot/w3c-wot-common-context.jsonld" ],
  "@type": [ "Thing", "Sensor" ],
  "geo:location": "testspace",
  "properties": {
    "prop1": {
      "type": "number",
      "@type": [ "Property", "Temperature" ],
      "saref:TemperatureUnit": "degree_Celsius"
    }
  }
};

try {
  // note that produce() fails if the TD contains an error
  let thing = WOT.produce(thingDescription);
  // Interactions were added from TD
  // WoT adds generic handler for reading any property
  // Define a specific handler for a Property
  thing.setPropertyReadHandler("prop1", () => {
    return new Promise((resolve, reject) => {
      let examplePropertyValue = 5;
      resolve(examplePropertyValue);
    }, e => {
      console.log("Error");
    });
  });
  await thing.expose();
} catch(err) {
  console.log("Error creating ExposedThing: "+ err);
}
ExposedThing with a simple property

```javascript
let temperaturePropertyDefinition = {
  type: "number",
  minimum: -50,
  maximum: 10000
};

let tdFragment = {
  properties: {
    temperature: temperaturePropertyDefinition
  },
  actions: {
    reset: {
      description: "Reset the temperature sensor",
      input: {
        temperature: temperatureValueDefinition
      },
      output: null,
      forms: []
    },
  },
  events: {
    onchange: temperatureValueDefinition
  }
};

try {
  let thing1 = WOT.produce(tdFragment);
  // Here add customized service handlers
  await thing1.expose();
  catch (err) {
    console.log("Error creating ExposedThing: " + err);
  }

  // The Thing can be used right away.
  setInterval( async () => {
    let mock = Math.random()*100;
    let old = await thing1.readProperty("temperature");
    if (old < mock) {
      await thing1.writeProperty("temperature", mock);
    }
  }, 1000);
```
Add an object Property

```javascript
try {
    // Create a deep copy of thing1 instance
    let instance = JSON.parse(JSON.stringify(thing1.instance));
    catch (err) {
        console.log("Error cloning Thing: " + err);
    }
}
```

Create an object that describes a Property

```javascript
const statusValueDefinition = {
    type: "object",
    properties: {
        brightness: {
            type: "number",
            minimum: 0.0,
            maximum: 100.0,
            required: true
        },
        rgb: {
            type: "array",
            "minItems": 3,
            "maxItems": 3,
            items: {
                "type": "number",
                "minimum": 0,
                "maximum": 255
            }
        }
    }
};
```

Customize the instance of the Thing

```javascript
instance["name"] = "mySensor";
instance.properties["brightness"] = {
    type: "number",
    minimum: 0.0,
    maximum: 100.0,
    required: true,
};
instance.properties["status"] = statusValueDefinition;
instance.actions["getStatus"] = {
    description: "Get status object",
    input: null,
    output: {
        status: statusValueDefinition
    },
    forms: [...]}
instance.events["onstatuschange"] = statusValueDefinition;
instance.forms = [...] // update
```

Create a new Thing based on instance.

```javascript
try {
    var thing2 = WOT.produce(instance);
    // Add customized service handlers here.
    // thing2.instance is now different than instance
    await thing2.expose();
} catch (err) {
    console.log("Error creating ExposedThing: " + err);
}
```
### Discovery API

**Constructor** (optional `ThingFilter filter`), `SecureContext`, `Exposed=(Window, Worker)`

```javascript
interface ThingDiscovery {
  readonly attribute `ThingFilter`? filter;
  readonly attribute boolean active;
  readonly attribute boolean done;
  readonly attribute `Error`? error;
  void start();
  `Promise<object>` next();
  void stop();
}
```

typedef `DOMString` DiscoveryMethod;
// "any", "local", "directory", "multicast"

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

**Discovery provides TDs:**
- Things exposed in the local WoT Runtime
- Things listed in a directory service
- Things exposed in a local network.
Discovery examples

// Discover Things exposed by local hardware

let discovery = WOT.discover({ method: "local" });

do {
  let td = await discovery.next();
  console.log("Found Thing Description for " + td.name);
  let thing = WOT.consume(td);
  console.log("Thing name: " + thing.instance.name);
} while (!discovery.done);

// Multicast discovery

let discovery = WOT.discover({ method: "multicast" });

setTimeout(() => {
  discovery.stop();
  console.log("Stopped open-ended discovery");
}, 10000);

do {
  let td = await discovery.next();
  console.log("Found Thing Description for " + td.name);
  let thing = WOT.consume(td);
  console.log("Thing name: " + thing.instance.name);
} while (!discovery.done);

if (discovery.error) {
  console.log("Discovery stopped.");
  console.log("Discovery error: " + discovery.error.message);
}

// Discover Things via directory

let discoveryFilter = {
  method: "directory",
  url: "http://directory.wotservice.org"
};

let discovery = WOT.discover(discoveryFilter);

setTimeOut(() => {
  discovery.stop();
  console.log("Discovery timeout");
}, 3000);

do {
  let td = await discovery.next();
  console.log("Found Thing Description for " + td.name);
  let thing = WOT.consume(td);
  console.log("Thing name: " + thing.instance.name);
} while (!discovery.done);

if (discovery.error) {
  console.log("Discovery stopped.");
  console.log("Discovery error: " + discovery.error.message);
}
node-wot

One implementation of the Scripting API
Dual W3C and Eclipse license
The *de-facto* reference implementation
node-wot: a Scripting API implementation

- node-wot is an open-source implementation of the WoT Scripting API
  [http://www.thingweb.io](http://www.thingweb.io)

- The project can be fully customized using various packages
  - td-tools
  - core
  - bindings (HTTP, CoAP, MQTT, WebSockets, …)
    - Other binding protocols can be added by fulfilling a given API
    - Content codecs (besides JSON, text, and octet-stream) can be added
  - Miscellaneous: demos, command-line interface

- Facts
  - NodeJS implementation in TypeScript
  - Development on GitHub: [https://github.com/eclipse/thingweb.node-wot/](https://github.com/eclipse/thingweb.node-wot/)
  - Dual-licensed: [Eclipse Public License v. 2.0](https://www.eclipse.org/legal/epl-v20.html) and [W3C Software Notice and Document License](https://www.w3.org/Consortium/LicenseInfo/CDDL-1.0)
  - Available through NPM (packages such as [core](https://www.npmjs.com/package/core), [td-tools](https://www.npmjs.com/package/td-tools), …)
node-wot - Demos and Tools

- **Web UI**
  - node-wot can be used as a browser-side JavaScript library (~160kB JS code)
  - [http://plugfest.thingweb.io/webui/](http://plugfest.thingweb.io/webui/)

- **TD Playground**
  - Tool to check the validity of a TD
  - Performs both syntactic checks and semantic checks
  - [http://plugfest.thingweb.io/playground/](http://plugfest.thingweb.io/playground/)

- **TD Directory**
  - REST interface to add, update and query TDs for discovery purposes
  - [http://plugfest.thingweb.io](http://plugfest.thingweb.io)
Demo

- **Server script** *(ExposedThing)* example *(counter.js)*

  ```bash
  $ node packages\cli\dist\cli.js examples\scripts\counter.js
  ```

- **Client script** *(ConsumedThing)* example *(counter-client.js)*

  ```bash
  $ node packages\cli\dist\cli.js --clientonly examples\scripts\counter-client.js
  ```

- **Browser Client example**
  - Pointing to Property, Action, Event
  - Listening to events
  - Changing with different binding (e.g., CoAP) values

Thanks!

Contributions welcome!

- Implementations
- Examples
- Experiments with API styles
- Browser use cases
- Applications
  - Thing Directory
  - Discovery
  - Script Management