WoT Scripting

WoT F2F Lyon 2018

WebIDL or TypeScript?
Disambiguations
API harmonization with the Web Platform

Challenges for Scripting

Overloaded terms:

- Programming object vs Thing
- WoT interactions vs programmatic interactions/idioms
  - WoT Event vs programmatic events
  - WoT Properties vs object properties
  - WoT Actions vs object methods
- Type definitions/mappings

Trends in the Web Platform

- Event: DOM vs Node.js
- Promises, Observables, Streams, AbortController, …
- WebIDL breaks

“Wild” Ideas

Use a JavaScript object to represent a WoT Thing

- WoT Property → object property (getter, setter)
- WoT Action → object method
- WoT Event → programmatic event
- See Dave Raggett’s Simplified API proposal

Use a [shadow/virtual] DOM to represent WoT Things

- scripts would operate on that
- TD → HTML
- Consuming TD → DOM

WebIDL or TypeScript or ...?

Issues with WebIDL: mapping ideas to WebIDL, then implement in TypeScript.

What about using TypeScript in the spec? It has issues as well:

- No ReSpec support for highlighting and linking (needs to be added)
- TC39 is moving to use WebIDL for ECMAScript (Kenneth/TAG)
- WebIDL is coercive and is used by all Web specs. (Marcos)

Suggestions (Marcos/Kenneth):

- use WebIDL as normative and TypeScript as informative API definition.
- don't use WebIDL to describe data structures (use prose + links to tests)
- use the Infra Standard to refer to common data definitions/algorithms.
- use TAG review(s) for early feedback/helping with issues.

Programming object vs Thing

Object

- Identity
- State (properties)
    - Get [All | subset]
    - Set [All]
- Behavior
    - Methods
    - Events/Notifications

Thing

- Identity (TD)
- State (Properties)
- Behavior
    - Actions
    - Events

Traditional IoT interactions

← Request:
- ← fetch/set state (property, set of properties)
- ← Subscribe to:
  - Notifications (value change, value threshold, condition set by client)
  - Event (as determined by the server)
- ← Unsubscribe from (one subs | all subs in one object | a set of subs)

→ Response:
- → state (property or set of properties)
- → status
- → Notification
- → Event

Traditional async programming with Events

In general APIs are migrating away from explicitly passing callbacks in favor of either

- Promises or
- one-shot events or
- repeating events + event listeners.

Patterns:

- Create an object and add an event / listener to that.
- Use a `startXxx()` method and add events in order to monitor progress/errors/status changes.

Events are defined by the DOM, Node.js, WoT, …

See: https://github.com/w3c/web-nfc/issues/152#issuecomment-425179760

DOM Events

Events are created by constructor. https://dom.spec.whatwg.org/#event

EventTarget https://dom.spec.whatwg.org/#interface-eventtarget

https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener

`.removeEventListener(type, listener[, options])`

`.dispatchEvent(event)`

https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/dispatchEvent

- Calls listeners synchronously and sequentially
- As opposed, native events (fired by the DOM) invoke handlers asynchronously via the event-loop

EventHandler: https://html.spec.whatwg.org/multipage/webappapis.html#eventhandler

`.addEventListener(type, listener[, options])`

- *listener*: a function that receives an Event as argument

Node.js events

https://nodejs.org/api/events.html

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {
  // ...
};
const myEmitter = new MyEmitter();

myEmitter.on('event', (a, b) => {
  console.log('an event occurred!');
});

myEmitter.emit('event', 'first_arg', 'second_arg');
```

`emit()` will invoke all listeners synchronously and sequentially.

Promises

Modeled for one-shot asynchronous operations.

Issue: no easy way to cancel (needs declaring another function).

Cancellable Promises have been extensively discussed.

- A lot of controversy: can you [guarantee] cancel, is it needed, etc
- First the outcome was Observables (2016)
- Now there is push for using Promise + AbortController + AbortSignal.

Observables

https://w3c.github.io/wot-scripting-api/#observables

https://github.com/tc39/proposal-observable

```
interface Observable {
  Subscription subscribe(EventHandler handler,
                         optional ErrorHandler errorHandler,
                         optional OnComplete onComplete);
};
interface Subscription {
  void unsubscribe();
  readonly attribute boolean closed;
};
callback EventHandler = void (any value);
callback ErrorHandler = void (Error error);
callback OnComplete = void ();
```

AbortController/AbortSignal

https://dom.spec.whatwg.org/#aborting-ongoing-activities
https://developers.google.com/web/updates/2017/09/abortable-fetch
https://developer.mozilla.org/en-US/docs/Web/API/AbortController
https://www.npmjs.com/package/abortcontroller-polyfill

```
const controller = new AbortController();
const signal = controller.signal;
let options = { filter = “none”, signal };

obj2.addEventListener(eventname, { signal });  // already supports AbortSignal
obj1.startLongOperation(data, options)  // has to support AbortSignal
    .then( () => { console.log(“success” }; )
    .catch( err => { if (err == AbortError) console.log(“aborted”); }

// after 3 seconds abort both operations
setTimeout(() => controller.abort(), 3000);
```

AbortController/AbortSignal (rationale)

Used for aborting [multiple] operations that return a Promise.

https://github.com/w3c/web-nfc/issues/147#issuecomment-425601613

I can understand how it can seem complex in isolation. But as with promises, streams, etc., reusing the same primitive everywhere has multiplicative effects throughout the platform. In particular, there's a common pattern of using a single AbortSignal for a bunch of ongoing operations, and then aborting them (with the corresponding AbortController) when e.g. the user presses cancel, or a single-page-app navigation occurs, or similar. So the minor extra complexity for an individual API leads to a large reduction in complexity when used with multiple APIs together.

Right now AbortController is only used in **Fetch** and **Web Locks** to my knowledge. But we're soon going to be using it in **Streams**, a promise-returning version of **setTimeout** (whatwg/html#617), and probably other upcoming APIs like **writable files**. Being able to cancel all these things together using a single tool is the dream, just like today you can use a single tool (promises) for async operations.

How to use AbortController/AbortSignal in specifications

For a function that returns a Promise and takes a dictionary parameter:

- Add a dictionary member ‘signal’ of type AbortSignal
- Add algorithmic steps to check on abort state and listening on abort signal.

See: https://dom.spec.whatwg.org/#abortcontroller-api-integration

Subscriber (proposed)

AbortController-compatible interface to be included instead of Observables (by ConsumedThing, ThingProperty, ThingEvent).

```
interface Subscriber {  // does NOT implement EventTarget
  void subscribe(NotificationHandler handler, optional SubscribeOptions options);
  attribute NotificationHandler? onerror; // simple callback, not an event!
};
dictionary SubscribeOptions { // includes subscription data/filters
  AbortSignal signal;
};
callback NotificationHandler = void (any value); // DOM EventHandler gets Event
interface WoTAbortController: AbortController {
  void abort(optional any cancellationData);
};
```

Updated ConsumedThing

```
interface ConsumedThing : ThingFragment {
  readonly attribute DOMString id;
  readonly attribute DOMString name;
  readonly attribute DOMString? base;
  readonly attribute PropertyMap properties;
  readonly attribute ActionMap actions;
  readonly attribute EventMap events;
  // getter for ThingFragment properties
  getter any (DOMString name);
};

ConsumedThing includes Subscriber;
```

Updated ThingProperty

```
interface ThingProperty : Interaction {
  // getter for PropertyFragment properties
  getter any (DOMString name);
  // get and set interface for the Property
  Promise<any> read();
  Promise<void> write(any value);
};
ThingProperty includes PropertyFragment;

ThingProperty includes Subscriber;
```

Discussion

Should we:

- Use [a subset of] DOM Event and [subset of] EventTarget?
  - But then it’s confusing to have TD Events vs scripting events
  - For subscribe use addEventListener options extended with WoT vocabulary (via TAG) + use AbortController/AbortSignal for unsubscribe
- Provide a generic sub/unsub interface (e.g. Observable)?
  - Independent from fashion trends in JS/Browser APIs
  - WoT-specific, needs implementing a shim in browser implementations
- Use the AbortController-compatible Subscriber proposal?
  - Adheres to the Web Platform requirements
  - Avoids using Events (still controversial)
  - Allows WoT extensions (additional data for subscription/unsubscribe).