WebDriver Extension API for Generic Sensors

Wanming Lin (@Honry) <wanming.lin@intel.com>
Raphael Kubo da Costa (@rakuco) <raphael.kubo.da.costa@intel.com>
Problem

Testing Sensor APIs...

- Is a sensor being created with the right state?
- Are sensor readings correct?
- Is the sensor measuring things at the right frequency?
- Are the right errors being thrown under the right circumstances?

... Without actual sensors

- No hardware = more predictable, easier to integrate in CI workflows
- Tests must work across browsers
- Gecko, Blink, WebKit have their own WPT copies, periodically synced with upstream.
- Tests from WPT often run as part of their CI.
- The binary running the tests is often not the browser binary.
- Not launched by WPT, no WebDriver.

https://wpt.fyi

- Runs tests in WPT on Chrome, Edge, Firefox and Safari.
- Reports what passes what fails across runs.
- Uses WPT’s infrastructure to launch **unmodified** browser binaries, controls them via WebDriver (ChromeDriver, Marionette etc).

Browser CI
Sensors in web-platform-tests

- wpt.fyi 👎
- Blink CI 👍
- Chromium-specific build artifacts checked into WPT (not always used these days, see this Chromium bug)
- “Agnostic” API on top that hardcodes a dependency on the files above for Chromium
Solution

- Specify a WebDriver extension protocol for mocking sensors with specific characteristics and behavior
- Other specs following the same approach: Permissions API, Reporting API
Basic Flow

HTTP Request
- Method (POST, GET, DELETE)
- Route (URL after domain)
- Body

HTTP Response
- Status
- Body

WebDriver
- Extension Commands
  - Create mock sensor
  - Get mock sensor
  - Update mock sensor reading
  - Delete mock sensor

Browsers
- Execute WebDriver Extension Command
- JSON WebDriver Response
- Mock Sensor
API Design (Mock Sensor)

A mock sensor simulates the behavior of a platform sensor which should have following capabilities:

- Mock sensor type
- Mock sensor readings (modifiable)
- Sampling frequency (upper and lower bounds)
- Connection flag (can be used to create a failing sensor, for example)
# API Design (Extension Commands)

<table>
<thead>
<tr>
<th>Extension Commands</th>
<th>HTTP Method</th>
<th>URI Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create mock sensor</td>
<td>POST</td>
<td>/session/{session id}/sensor</td>
</tr>
<tr>
<td>Get mock sensor</td>
<td>GET</td>
<td>/session/{session id}/sensor/{type}</td>
</tr>
<tr>
<td>Update mock sensor reading</td>
<td>POST</td>
<td>/session/{session id}/sensor/{type}</td>
</tr>
<tr>
<td>Delete mock sensor</td>
<td>DELETE</td>
<td>/session/{session id}/sensor/{type}</td>
</tr>
</tbody>
</table>
Create mock sensor

To create an "accelerometer" mock sensor with session ID 21, the local end would send a **POST** request to /session/21/sensor with the body:

```json
{
    "mockSensorType": "accelerometer",
    "maxSamplingFrequency": 60,
    "minSamplingFrequency": 5,
    "connected": true
}
```

```csharp
    dictionary MockSensorConfiguration {
        required MockSensorType mockSensorType;
        boolean connected = true;
        double? maxSamplingFrequency;
        double? minSamplingFrequency;
    };
```
Get mock sensor

To get an "accelerometer" mock sensor with session ID 22, the local end would send a **GET** request to `/session/22/sensor/accelerometer` without a body. On success, the remote end returns with serialized mock sensor information:

```json
{
    "maxSamplingFrequency": 60,
    "minSamplingFrequency": 5,
    "requestedSamplingFrequency": 30
}
```
Update mock sensor reading

To update the mock sensor reading of "accelerometer" mock sensor with session ID 23, the local end would send a **POST** request to /session/23/sensor/accelerometer with a body like this:

```
{
    "x": 1.12345,
    "y": 2.12345,
    "z": 3.12345
}
```
Delete mock sensor

To delete an "accelerometer" mock sensor with session ID 24, the local end would send a **DELETE** request to /session/24/sensor/accelerometer without a body.
## Handling Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>HTTP Status</th>
<th>JSON Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>no such mock sensor</em></td>
<td>404</td>
<td><em>no such mock sensor</em></td>
<td>no mock sensor matching the given type was found.</td>
</tr>
<tr>
<td><em>mock sensor already created</em></td>
<td>500</td>
<td><em>mock sensor already created</em></td>
<td>A <a href="#">command</a> to create a mock sensor could not be satisfied because the given type of mock sensor is already existed.</td>
</tr>
</tbody>
</table>
WPT’s testdriver.js

- JS API to allow tests to perform human-like interaction via WebDriver
- Current commands include `click`, `send_keys` and a few others
- API is separate from implementation, which can be overridden by vendors
- `test_driver.click(element)`
  - When using WPT’s test runner, ends up sending a `Find Element` command, then a `Click` command, both via the WebDriver protocol
  - Blink’s testdriver-vendor.js (used in CI) implements it via `chrome.gpuBenchmarking.pointerActionSequence()`, not WebDriver
**Obstacles/Gaps**

- Long process
  - Spec
  - Browser-specific bits. For Chrome: ChromeDriver changes, Chrome DevTools Protocol changes
  - WPT changes ([testdriver.js](#))
- Does not automatically solve the Browser CI case
  - No WebDriver: same WPT testdriver API needs to be reimplemented
  - Medium/long-term plans ongoing to solve this, at least for Blink
- Security considerations? Web developer use cases? API design?
- WebDriver calls are unidirectional
- The spec introduces new WebDriver errors, but WebDriver support for new error types is pending
What’s next?

- Call for broader review of this Extension API (thanks @jugglinmike!)
- Prototyping
- See if our approach can be generalized so other specs can adopt it (e.g. Media Streams, Screen Orientation, Vibration etc.)
Merci