The Mobile JSON Wire Protocol

Architecting the Future of Mobile Automation
I’m Jonathan Lipps

I manage an engineering team at Sauce Labs and take care of the Appium project. I’m involved with moving the spec for mobile automation forward.

Find me online

@jlipps
@AppiumDevs
@saucelabs

http://saucelabs.com
http://appium.io
http://jonathanlipps.com
The JSONWP: Why?
What's the motivation behind the JSON Wire Protocol?

HTTP Protocols: A Brief Intro
How HTTP protocols (like the JSONWP) work

The JSONWP: How it works
A detailed look at how the spec is put together

The Mobile JSONWP
Extending the JSONWP for a mobile future

The WebDriver Spec
Double spec, what does it mean??
The JSONWP is…

The abstract specification of how automation behaviors are mapped to Selenium server HTTP requests and responses
The JSONWP: Why?

What’s the motivation behind the JSONWP?
We want a client/server implementation

- Write tests in any language
- Possibility of Selenium Grid, or Sauce Labs
- Possibility of multiple server implementations ("drivers"), maintained by different groups
Client/server implementation requires a communication spec

- HTTP / REST is a good base for a spec
- Every language worth writing tests in has an HTTP library
- HTTP servers easy to write in most languages as well
Anyone can implement an open protocol

- The goal of Selenium was always to go away
- Chromedriver, IEDriver, any future diver can be stitched seamlessly together
HTTP Protocols: A Brief Intro

How HTTP protocols (like the JSONWP) work
HTTP Protocols

- HTTP Client
  - REQ VERB
  - REQ ROUTE
  - REQ BODY

- HTTP Server
  - ACTION

- Response
  - RES STATUS
  - RES BODY
HTTP Protocols

HTTP Request

- HTTP Verb: general kind of thing you want to do
  - GET, POST, etc…
- HTTP Route (or endpoint): address for the action/resource
- HTTP Request Body: more (text) details to send to the server

HTTP Response

- HTTP Status: generally what happened
  - 200, 404, 500, etc…
- HTTP Response Body: more (text) response details
The JSWONP: How it Works

A detailed look at how the spec is put together
The JSONWP: **How it works**

HTTP Verbs

- **GET**: retrieve information from the server
  - get the status of the server
  - get the text of an element
- **POST**: make something happen, or get info with side effects
  - start a session
  - find an element
  - click
- **DELETE**: remove a resource
  - only used with ending a session
HTTP Routes

- Many routes
- Object-oriented structure encoded in routes
- GET /status
- POST /session
- GET /sessions
HTTP Response Status Codes

• 200  your action completed successfully
• 500  something went wrong
• 404  you tried to access a route or resource that wasn’t there
• 501  this server doesn’t implement that action
HTTP Request and Response Bodies

- Everything is JSON text!
- Every route describes the JSON it expects in the request
  - POST /session
  - `{“desiredCapabilities”: {“browserName”: “safari”}}`
## The JSONWP: How it works

<table>
<thead>
<tr>
<th>JSONWP Status</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
<td>The action completed successfully</td>
</tr>
<tr>
<td>7</td>
<td>NoSuchElement</td>
<td>Could not find an element matching your requirements</td>
</tr>
<tr>
<td>33</td>
<td>SessionNotCreatedException</td>
<td>A session could not be created</td>
</tr>
</tbody>
</table>
HTTP is stateless; your tests are not

• Client and server need to keep track of objects across multiple requests/responses
  • Sessions, elements, windows, frames, contexts…
• The server assigns IDs to these and shares them with the client. They can then be used in subsequent requests.
• POST /session/:sessionId/element
  • {“status”: 0, “value”: {“ELEMENT”: “1234”}}
• POST /session/:sessionId/element/:elementId/click
JSONWP clients

- The purpose of the client is to provide an idiomatic interface in a given language, hiding protocol details.
- The server does not know (or care) what client you are using—it only cares about the protocol.
- The server cannot reliably reverse-engineer your test code based on a session.
driver = webdriver.remote(...) 
driver.set_implicit_wait_timeout(10000) 
driver.url = "http://google.com" 
if driver.title == "google": 
    driver.quit() 
else: 
    # do some other stuff
Advanced JSONWP

• We have only scratched the surface
• Read the doc to learn more!
• Types of JSON request/response objects, complex error handling, unicode for non-text keyboard input, etc…
The Mobile JSONWP

Extending the JSONWP for a mobile future
https://github.com/SeleniumHQ/mobile-spec/blob/master/spec-draft.md
What is the MJSONWP?

- Mobile-specific extensions to the JSONWP
- New routes defining new automation behaviors
- New locator strategies
- …and that’s it!
Why do we need a MJSONWP?

- Mobile automation is a thing now, if you haven’t heard
- Mobile automation can and should build off of the excellent architectural choices made by Selenium/WebDriver
- No need to re-invent a protocol
- The JSONWP is not complete enough for mobile automation
- Projects should compete on reliability, features, etc… *not* the spec they use
History of the MJSONWP

• 1-day meeting in London, 8/2013
• Participants included: Simon Stewart, David Burns, myself, Santi Suarez Ordoñez, François Reynaud, Dominik Dary
• Since then other Selenium and Appium community members have assisted in moving things forward, for example Luke Inman-Semerau, Matthew Edwards, and Malini Das
Status of the MJSONWP

- Very much a draft
- Lot of good ideas, some of them will change
- The automation “backends” do not always give us enough control to actually implement the spec
- Appium, Selendroid, and ios-driver have implemented all or some of the MJSONWP, server-side
- Clients getting updated as well
New desired capability structure

• The world of mobile is much more complex than the world of browsers
• We need new ways to specify new combinations of device, OS, version, and even the specific choice of automation mechanism.
• Also… hello, apps!
New desired capability structure

- **deviceName** — the kind of device you want to automate
- **platformName** — the OS you want to automate
- **platformVersion** — the version of that OS
- **automationName** — the automation mechanism
- **app** — the path to the app you want to automate
- **browserName** — the name of a mobile browser to automate
New desired capability structure

{
    platformName: "iOS",
    platformVersion: "8.4",
    deviceName: "iPhone 6",
    app: "/path/to/my.app"
}
New desired capability structure

{
    platformName: "iOS",
    platformVersion: "8.4",
    deviceName: "iPhone 6",
    browserName: "Safari"
}
New desired capability structure

```
{
    platformName: "Android",
    platformVersion: "4.4",
    deviceName: "Nexus S",
    app: "/path/to/my.apk",
    automationName: "Selendroid"
}
```
Updated locator strategies

- **class name** — the native UI class (not HTML class)
- **id** — the native resource id
- **xpath** — same as before, but applied to an XML document representing a native UI hierarchy
New locator strategies

- **accessibility id** — the string used for making a UI object accessible
- `-android uiautomator` — use the Android UiAutomator selector API to find an element
- `-ios uiautomation` — use the iOS UIAutomation selector API to find an element
Network connection

- Allows modifying the network connectivity state of the device
  - Airplane mode, data mode, wifi mode
- GET /session/:sessionId/network_connection
  - returns a NetworkConnection value
- POST /session/:sessionId/network_connection
  - expects a NetworkConnection value
Network connection

- NetworkConnection
- {“type”: <int>}
- Here, the value of type is an integer that represents any combination of network connection states
<table>
<thead>
<tr>
<th>Data</th>
<th>Wifi</th>
<th>Airplane</th>
<th>“Type”</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>Off</td>
<td>Off</td>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>Off</td>
<td>Off</td>
<td>On</td>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>Off</td>
<td>On</td>
<td>Off</td>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>Off</td>
<td>On</td>
<td>On</td>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>On</td>
<td>Off</td>
<td>Off</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>On</td>
<td>Off</td>
<td>On</td>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>On</td>
<td>On</td>
<td>Off</td>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>On</td>
<td>On</td>
<td>On</td>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>
Rotation

• More general than the current “orientation” route
• Arbitrary movement of device in a 3d space
• Automation tools do not yet provide ability to simulate arbitrary movement/position input
• GET /session/:sessionId/rotation
  • returns a DeviceRotation value
• POST /session/:sessionId/rotation
  • expects a DeviceRotation value
Rotation

- DeviceRotation
  - `{“x”: <float>, “y”: <float>, “z”: <float>}
  - Here, the values of x, y, and z represent the angle that the device is rotated about each of those axes in 3d space
Contexts

• Apps, unlike browsers, can have multiple contexts
• Hybrid apps have a ‘native’ context and one or more ‘web’ contexts
• GET /session/:sessionId/contexts
  • returns a list of available context names
• GET /session/:sessionId/context
  • returns the current context name
• POST /session/:sessionId/context
  • change the current context to the name provided
Contexts

• Some commands can only work in certain contexts
  • GET /session/:sessionId/title
  • InvalidContentException
  • NoSuchContext
What’s left to do?

• Create a set of mobile apps to test with
• Create a set of reference test scripts that can be used to validate any MJSONWP server implementation
• Define XML schemas for Android/iOS UIs
• Probably lots more once we dig in further!
The WebDriver Spec

Double spec, what does it mean??
https://w3c.github.io/webdriver/webdriver-spec.html
Parallel evolution

- The MJSONWP is based off of the JSONWP
- The JSONWP will be superseded by the WebDriver spec
- The MJSONWP will have to take the WebDriver spec into account sooner or later
A period of transition

• The WebDriver spec is subtly different in many ways than the JSONWP
  • (Sometimes) different routes
  • Different error handling
  • Different request/response expectations
• Client and server implementations will undergo a (hopefully not too long) process of supporting the new spec
• The MJSONWP will eventually merge into the WebDriver spec
The end goal

- No more Selenium
- No more Appium, Selendroid, ios-driver, etc…
- Just the protocol and the vendors’ implementations of it
- Browser vendors have (mostly) agreed on how to render a website given the same HTML, CSS, etc…
- Browser vendors are getting close to doing this for automation
- Let's make the same thing happen for mobile!
How to get involved

• Subscribe to SeleniumHQ/mobile-spec and chime in on issues
• Look for TODOs and make your own proposals
• Let us know what mobile behaviors the MJSONWP should be extended to work with
• Help client and server implementations support the MJSONWP
• Help us find conceptual gaps in the spec
Questions?

@jlipps
@saucelabs
Thank you!

@jlipps
@saucelabs