Extending Splunk's REST API for Fun and Profit

James Ervin

Principal Engineer, Security and Compliance Solutions

Splunk, Inc.

.conf2016

splunk>

Disclaimer

Overview

At the conclusion of this presentation, you should be able to discuss the following:

- What is a REST API?
- How does Splunk implement REST style?
- How can I extend Splunk's REST API within my application?
- Do I have to use REST style?
- *Why would I want to do any of this?*

REST Style: Definition

REST (Representational State Transfer) is a set of architectural constraints that make a web application "RESTful"*:

- client-server interaction over HTTP
- stateless communication
- cacheable content
- etc.

REST is a way to do IPC (interprocess communication) over HTTP.

*Cf. Architectural Styles and the Design of Network-based Software Architectures; Fielding, R. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. 2000.

REST Style: Practical Considerations

REST is a style, rather than a standard or a protocol.

There is no formal protocol specification for REST, in the way that there is for XML-RPC, SOAP, etc.

- In practice, this can be both liberating and frustrating.
- As an API designer, you have many degrees of freedom to work with.
- As an API consumer, APIs you interact with will differ subtly, even within the limited degrees of freedom offered by REST. At the broadest level, note that the style does not specify a default format, although XML and JSON are commonly implemented.

Interacting with Splunk REST: direct

Splunk's REST API can be interacted with directly in two ways:

Via a request to a port: localhost://8089 (served by the splunkd process)

```
curl -k -u admin:changeme
https://127.0.0.1:8089/services/saved/searches?count=1
```

Q: How do I interact with Splunk REST on port 8089, when my browser is making requests to port 8000? Doesn't this violate same-origin policy?

A: Yes! See the next slide for the alternative access mechanism…

Interacting with Splunk REST: proxied

Via request to port localhost://8000 (“splunkweb”):

```
curl –k 'https://my_hostname:8000/en-US/splunkd/__raw/services/
saved/searches?output_mode=json&count=1' -H 'Cookie:
splunkweb_csrf_token_8000=11602893886132396046;
session_id_8000=b1cba29d67a369c9b2410c4885a0bca1da0ab6fd;
splunkd_8000=vjRt4ZFCbiyplxbUW2qDFe9EqTH3jCFciaRa^ul8RTQUDD_XN4WY4MT
nzue6frZBd^j1xS2MC8p4oUXWWuIoGDia4tNgSNntTAgfudmFLjkKI2PtiBK0xMnf6KS
afjg’
```

This is how you interact with Splunk REST from your Javascript code. Note the inclusion of the language (**en-US**) and the **splunkd/__raw** prefix – this is important!

Proxying REST Calls: History

Prior to Splunk 6.2, Splunk's own REST API endpoints were whitelisted internally and were exposed on port 8000 through a Python proxy (proxy.py), which was executed as part of the Python splunkweb process (a CherryPy web server).

This had two disadvantages:

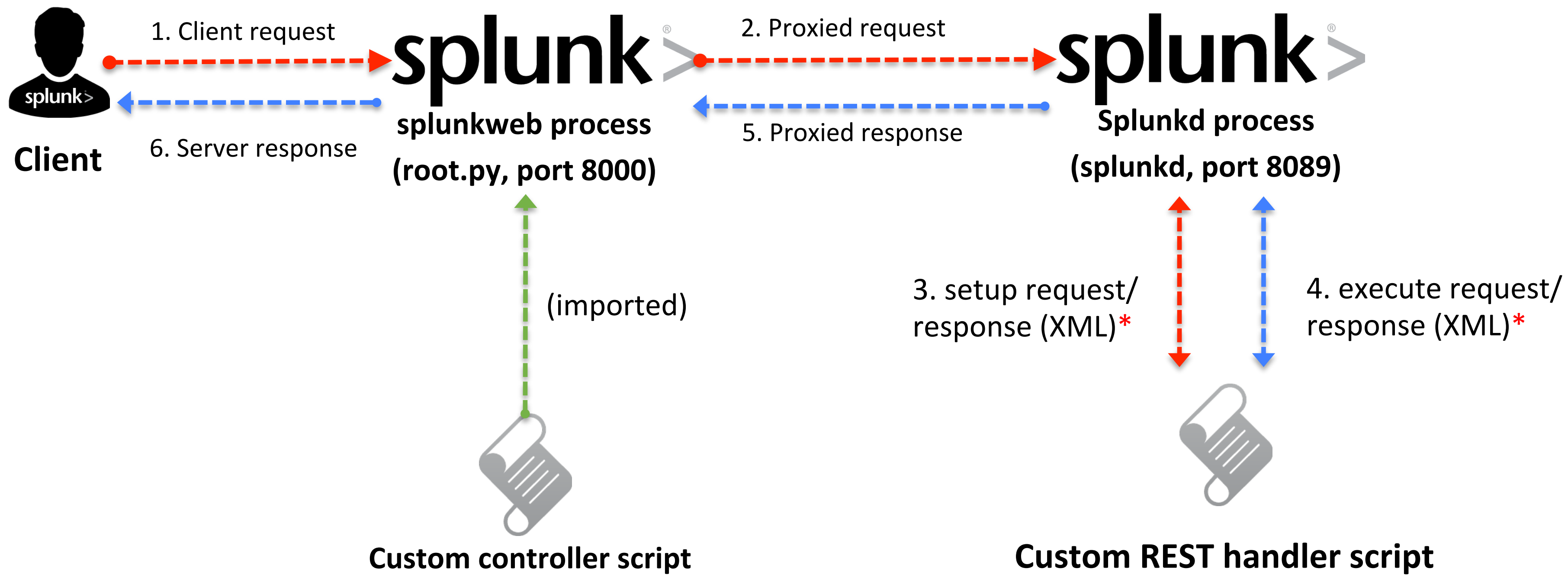
1. Python code execution was required for each REST call.
2. The set of proxied endpoints could not be extended by apps! So apps had to include a separate Python component known as a "Splunkweb controller" in order to proxy their own endpoints. This led to extensive duplication of code.

Proxying REST Calls: History

In Splunk 6.2, the “expose” keyword was introduced in web.conf. This permits direct pass-through of requests to custom REST endpoints to the C++ splunkd back-end. This has two advantages:

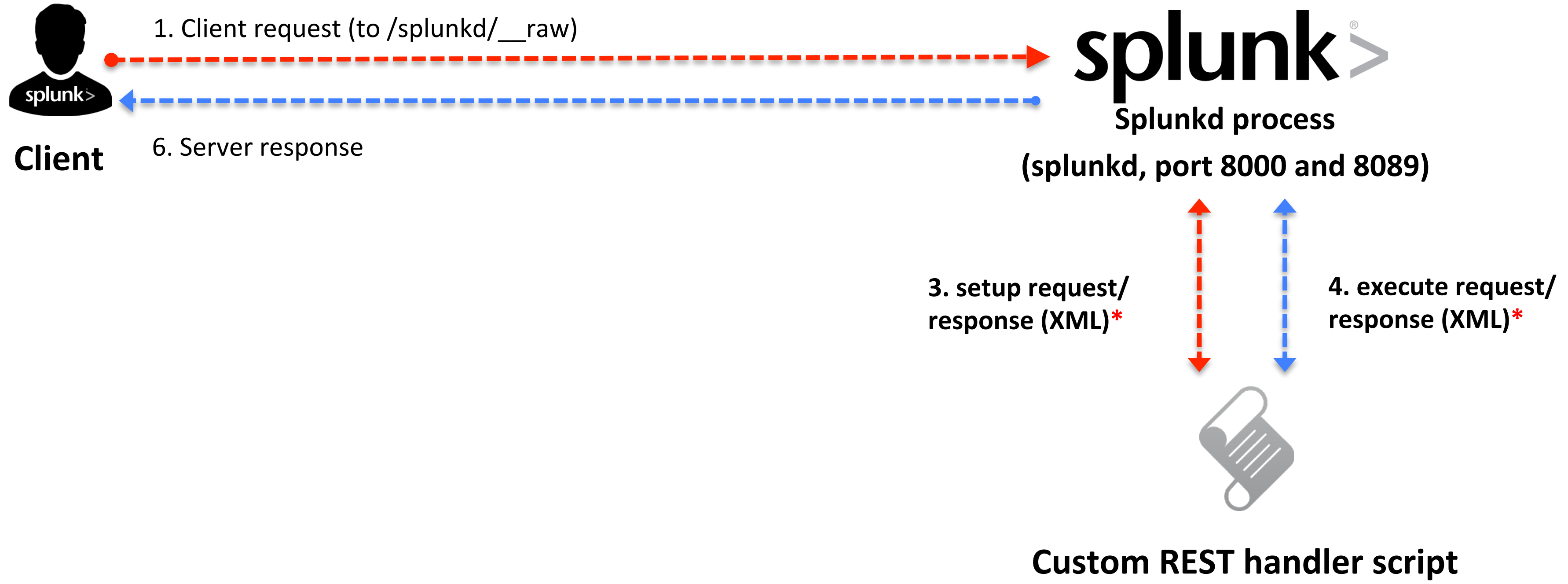
1. Python code is no longer involved in the “hot path” from client to server for custom REST endpoints **as long as access is done via the /splunkd/__raw URI**. (Other URIs are still proxied by Python and may be slower!)
2. The app developer can now expose a REST endpoint directly via configuration, without writing additional code.

Anatomy of a REST Call: Pre-Splunk 6.2



* = new Python process

Anatomy of a REST Call: Post-Splunk 6.2


1. Client request (to /splunkd/__raw)
6. Server response
Client
Splunkd process
(splunkd, port 8000 and 8089)
3. setup request/ response (XML)*
4. execute request/ response (XML)*
Custom REST handler script

* = new Python process

Proxying REST Calls: Basic Configuration

In web.conf (Splunk 6.2 and up):

```
[expose:correlation_searches]
pattern = alerts/reviewstatuses
methods = GET,POST
```

Note that the URL is what's actually "exposed" here. You can even expose Core endpoints that aren't exposed by default. The above would correspond to a URL of:

```
https://your_hostname:8000/en-US/splunkd/__raw/services/
alerts/reviewstatuses
```

Proxying REST Calls: Wildcarding

```
[expose:correlation_searches]
pattern = alerts/correlationsearches/*
methods = GET,POST
```

This exposes a URL of:

https://your_hostname:8000/en-US/splunkd/__raw/services/alerts/correlationsearches/SEARCH_NAME_HERE

But NOT:

https://your_hostname:8000/en-US/splunkd/__raw/services/alerts/correlationsearches

Splunk REST: Documentation

Splunk provides a (mostly) RESTful API. This API is served up on any running Splunk instance, usually on port 8089, and is well-documented here:

REST API Reference Manual – URI Quick Reference

http://docs.splunk.com/Documentation/Splunk/latest/RESTREF/RESTlist

REST API User Manual

http://docs.splunk.com/Documentation/Splunk/latest/RESTUM/RESTusing

restmap.conf

http://docs.splunk.com/Documentation/Splunk/latest/Admin/Restmapconf

Extending the API: Why?

Question: Why would you want to extend the REST API?

Answer(s): Several reasons, most of which are just general principles of good software design.

1. Encapsulation
2. Computation
3. Functionality
4. Abstraction
5. Performance
6. App Management
7. Cloud Compatibility

Extending the API: Encapsulation

In the Enterprise Security app, we frequently encounter product requirements that require construction of a new concept.

Example:

A “correlation search” consists of up to 3 configuration objects:

- A savedsearches.conf entry
- Metadata about the search’s related regulatory compliance settings in “governance.conf”
- Metadata about the search’s workflow in “correlationsearches.conf”

Encapsulation behind an API permits manipulation of these entities as a unit or “single concept”.*

* Note: Splunk does NOT provide transactional semantics on configuration files.

Extending the API: Computation

Certain types of computation might be unsafe to perform solely in the browser.

Usually, this means argument validation.

Example:

If you create a custom configuration file that has specialized validation requirements, a custom REST handler to provide server-side validation may be required.

Extending the API: Functionality

The Core Splunk REST API may not provide a certain feature you need.

Example:

In an earlier version of Enterprise Security, in order to propagate some configuration changes across a Search Head Cluster (SHC), we had to write a REST handler that would "fan out" modifications across a cluster so that edits made on one search head would be visible on the other search heads.

These cases are generally rare. Internally, we generally don't encourage development of significant "plumbing" of this sort at the app level, when it should really be done in the Core splunkd process.

Extending the API: Abstraction

You may need to future-proof your app by providing a layer of abstraction, so that future modifications to the app can be made without requiring significant front-end or user experience work.

Example:

ES contains a small API that provides for storage of small files in the KV store as encoded strings. By writing an API for this, instead of forcing the front-end to write to KV store APIs, we retain the flexibility to swap out the storage layer at any time without requiring significant UI work.

Extending the API: Performance

Operations that would generate many round-trips to the server, often benefit from being wrapped in a REST API.

Example:

ES contains a feature known as the “Notable Event Framework” which overlays a minimal ticketing workflow system on top of indexed Splunk events. Editing events via this framework usually requires issuing multiple calls to determine the existing status and ownership of an event, and then validating that the current user has permission to change that status (for instance: not all analysts may be allowed to “close” incidents).

Doing the status check completely in the browser would generate possibly thousands of calls to and from the server, which would be prohibitively expensive.

Extending the API: App Management

1. Using the “triggers” stanza in app.conf, you have the ability to force REST calls to your handler to occur (or not occur) upon app state changes (install, update, enable, disable).
2. Splunk’s “layered conf” system is a very simple data persistence mechanism. You can use this when you need to store a bit of data and don’t want to be restricted to indexing it and only being able to get at it via search.

Example:

In the Enterprise Security app, we utilize this to force the customer to go through the setup process again following an app upgrade, so that they receive the newest configurations.

```
[triggers]
reload.ess_setup = access_endpoints /admin/ess_configured
```

Extending the API: Cloud Compatibility

In Splunk Cloud, you can't make the same assumptions about your storage or local environment.

Nor does the customer have shell access to the server!

This means that any operation you used to do by hand via direct edits to configuration files, or via other direct filesystem access, is better done by exposing the function in a REST API.

This is probably the most important reason to begin utilizing custom REST handlers in your app.

Cf. Steve Yegge's infamous google+ rant: https://plus.google.com/+RipRowan/posts/eVeouesvaVX on the importance of interfaces as they pertain to platform development.

REST APIs

How do I write these things?

REST Interfaces

You may be surprised to discover that Splunk offers 4 distinct methods for writing REST APIs, each with unique behavior. They are shown below on two axes: the ***interface*** that the API is written in, and the ***lifetime*** of the process that executes the REST handler code.

		Process Lifetime	
		non-persistent	persistent
Interface	EAI (admin_external)	All versions	6.4 and up
	Non-EAI (script)	All versions	6.4 and up

REST Interfaces: EAI

EAI – Extensible Administration Interface

Designed to facilitate more rapid development of REST interfaces on the C++ backend. Utilizing this interface provides some additional services such as:

- Automatic pagination
- Automatic output formatting (XML, JSON)
- Access control
- Filtering
- Limited argument validation via Splunk "eval" syntax

EAI is typically associated with management of custom Splunk configuration files.

REST Interfaces: EAI (example)

Custom EAI handlers are indicated by the presence of the "admin_external" stanza in restmap.conf. The highlighted parameters are only valid with this setting.

```
[admin_external:correlationsearches]
handlertype =    python
handlerfile =    correlationsearches_rest_handler.py
handleractions = list,edit,create,remove,_reload
```

** Only Python scripts are supported.

REST Interfaces: Mapping EAI Handlers

```
[admin:alerts_threatintel]
match=/alerts
members=correlationsearches
```

Maps the handler “correlationsearches” to the URI “services/alerts/correlationsearches”

```
## Correlation Searches Handler
[admin_external:correlationsearches]
handlertype = python
handlerfile = correlationsearches_rest_handler.py
handleractions = list,edit,create,remove,_reload
```

Endpoint-specific settings

```
[eai:conf-correlationsearches]
capability.write = edit_correlationsearches
```

Assign role-based access controls on the handler

REST Interfaces: script

A “raw” interface for writing REST interfaces.

Services such as pagination, support for multiple output formats, etc. are the responsibility of the developer. Conformance to REST style is also the responsibility of the developer.

Using this interface, you have absolute freedom.

REST Interfaces: script (example)

A “script” handler is indicated by the presence of the “script” setting in restmap.conf. Highlighted attributes are only valid with this type:

```
[script:notable_update]
match = /notable_update
scripttype = python
script = notable_update_rest_handler.py
handler = notable_update_rest_handler.NotableEventUpdate
requireAuthentication = true
capability=edit_notable_events
output_modes=json
```

REST Interfaces: Mapping Script Handlers

```
###### REST notable update ######
[script:notable_update]
match = /notable_update
script = notable_update_rest_handler.py
handler=notable_update_rest_handler.NotableEventUpdate
requireAuthentication = true
capability=edit_notable_events
output_modes=json
```

Maps the handler to the URI "services/notable_update"

The class that serves requests

Assign role-based access controls on the handler

REST Interfaces: Segue: What is “Persistence?”

Before we can talk about how to write handlers, we need to understand the other axis on our chart: what is “persistence”?

Recall the execution model for a Splunk REST call on the previous diagram:

1. The splunkd process receives request on port 8089.
2. This python script is invoked: $SPLUNK_HOME/bin/python runScript.py <setup|execute>
3. This script loads the REST handler using Python’s execfile() method, handing off STDIN and STDOUT as needed.

It does this ***twice*** for every REST call: once to ***setup*** the REST handler, once to ***execute*** it.

That’s two invocations of Python for ***every REST call***.

REST Interfaces: Segue: What is "Persistence?"

"Persistent" mode means that the splunkd process will only execute one process per REST call. Additionally, this process will ***persist*** until it is idle for a period of time (60 seconds), at which point it will be reaped by the primary splunkd process (no developer action required). During the non-idle interval, it can service multiple requests.

This is the execution model for a Splunk persistent REST call:

1. The splunkd process receives request on port 8089.
2. The python script is invoked directly:

   ```
   $SPLUNK_HOME/bin/python <YOUR_SCRIPT HERE> persistent
   ```

(subsequent requests get passed to the same process directly)

REST Interfaces: Handler Base Classes

Python classes are distributed with Splunk that you can inherit from to write your own handlers:

		Process Lifetime	
		non-persistent	persistent
Interface	EAI (admin_external)	MConfigHandler	MConfigHandler
	Non-EAI (script)	BaseRestHandler	PersistentServerConnectionApplication

REST Interfaces: Handler Base Classes

Python classes are distributed with Splunk that you can inherit from to write your own handlers:

		Process Lifetime	
		non-persistent	persistent
Interface	EAI (admin_external)	**MConfigHandler**	**MConfigHandler**
	Non-EAI (script)	BaseRestHandler	PersistentServerConnectionApplication

REST Interfaces: Adding EAI Mode Persistence

Q: What did we notice about the preceding slide?

A: The classes providing EAI support are the same!

That's correct: enabling persistence on a custom handler written using the EAI specification is *simply a configuration change.* To add persistence to an EAI handler, simply add this to your restmap.conf:

```
handlerpersistentmode = true
```

However… this is not to say that your handler is guaranteed to work properly. Why? If you were doing work in the *__init__()* method of your handler, and were depending on that work being done to properly serve requests, when in persistent mode this work will NOT be redone – because __init__() is never called again!

REST Interfaces: Adding Script Mode Persistence

Enabling persistence on a "script" custom REST handler requires:

1. Add this to your restmap.conf:

```
scripttype = persist
```

2. Rewrite your handler to use the new protocol specification. This is the hard part.

Gold star question: Persistent scripts execute only once. What does this imply for discoverability?

REST Interfaces: Classes

EAI, persistent and non-persistent: MConfigHandler

```
$SPLUNK_HOME/lib/python2.7/site-packages/splunk/admin.py
```

Script, non-persistent (two competing implementations): BaseRestHandler

```
$SPLUNK_HOME/lib/python2.7/site-packages/splunk/rest/__init__.py
$SPLUNK_HOME/etc/system/bin/sc_rest.py
```

Script, persistent: PersistentServerConnectionApplication

```
$SPLUNK_HOME/lib/python2.7/site-packages/splunk/persistconn/application.py
```

REST Interfaces: Recommendations

The non-persistent interfaces should be avoided.

		Process Lifetime	
		non-persistent	persistent
Interface	EAI (admin_external)	MConfigHandler	MConfigHandler
	Non-EAI (script)	BaseRestHandler	PersistentServerConnectionApplication

AVOID

REST Interfaces: Recommendations

1. Avoid versions of Splunk prior to 6.2 so that you can make use of the “expose” web.conf directive.
2. Non-persistent interfaces should be avoided unless your app requires compatibility with pre-Splunk 6.4 versions.

Reasons for recommendation #2:

- Persistent interfaces offer all the flexibility of the non-persistent interfaces.
- Performance of persistent REST handlers is vastly improved.
- “script” handlers using non-persistent mode can actually conflict with REST scripts running in unrelated apps.

REST Interfaces: Recommendations

AND…

- Persistent REST handlers can now be written ***in compiled languages*** using the “driver” directive:

```
[script:my_handler_written_in_go]
match = /test
driver = echo
driver.arg.1 = <whatever>
script = echo
scripttype=persist
requireAuthentication = true
output_modes=json
```

REST Interfaces: Sample Code

Sample code for simplistic REST handlers using all the interfaces detailed in this presentation (including the ill-advised ones) is available at:

https://github.com/jrervin/splunk-rest-examples

REST Interfaces: Demo

THANK YOU

.conf2016

splunk>