

How to query and validate Splunk logs in Service Suite

Overview

The `splunk.js` library provides a reusable integration layer between AIQ Service Suite and Splunk. It allows test scripts to query Splunk indexes using SPL (Search Processing Language), filter and retrieve log data, and assert on the results as part of a functional or end-to-end test validation flow.

Common use cases include:

- Asserting that an audit log event was written after a user action (e.g. login, purchase, payment)
- Validating specific field values in application logs (e.g. transaction status, HTTP response codes)
- Checking that no ERROR or FATAL log entries appeared after a critical scenario
- Waiting for async events to appear in Splunk before continuing a test
- Extracting values from Splunk and storing them in TheLocker for downstream test cases

Prerequisite: The AIQ TestNode executing the script must have network access to the Splunk REST API on port **8089**. Verify security group / firewall rules before using this library.

Setup

1. Add splunk.js to your repository

splunk.js file: `splunk.js`

Upload `splunk.js` to your scenario repository. A recommended location is a shared folder so it can be reused across multiple scenarios:

```
your-repo/  
  shared/  
    splunk.js <-- place it here  
  scenarios/  
    checkout-flow/  
      setup.js  
      script1.js
```

2. Configure DPL variables

Never hardcode Splunk credentials in scripts. Add the following variables to your DPL (Data Provider Library) or scenario data file:

DPL Variable	Description	Example
<code>SPLUNK_HOST</code>	Hostname or IP of the Splunk instance	<code>splunk.internal.mycompany.com</code>
<code>SPLUNK_TOKEN</code>	Splunk service account Bearer token	<code>eyJrIjoiT...</code>

To generate a Bearer token in Splunk: **Settings** → **Tokens** → **New Token**. Use a dedicated service account, not a personal user.

Do not commit credentials to the repository. Always use DPL variables or TheLocker for sensitive values.

3. Include the library in your script

Use the Service Suite `include` directive at the top of any script that needs Splunk access. The `{ds}` token resolves to the root of your scenario's data source directory at runtime:

```
/**  
 * @aiq.servicesworkbench  
 */  
include('{ds}/shared/splunk.js');
```

Initialization

Call `splunk.init()` once at the beginning of each script before making any queries:

```
splunk.init({
  host: $SPLUNK_HOST, // DPL variable
  port: 8089,
  token: $SPLUNK_TOKEN, // DPL variable
  defaultIndex: 'app_logs', // used when no index is specified in SPL
  defaultEarliest: '-15m', // default look-back window for all searches
});
```

Full init options

Option	Default	Description
<code>host</code>	<code>localhost</code>	Splunk hostname or IP
<code>port</code>	<code>8089</code>	Splunk REST API port
<code>scheme</code>	<code>https</code>	<code>https</code> or <code>http</code>
<code>token</code>	<code>null</code>	Bearer token (preferred auth method)
<code>username</code>	<code>null</code>	Basic auth username (alternative to token)
<code>password</code>	<code>null</code>	Basic auth password (alternative to token)
<code>defaultIndex</code>	<code>main</code>	Fallback Splunk index
<code>defaultEarliest</code>	<code>-15m</code>	Default time window start
<code>defaultLatest</code>	<code>now</code>	Default time window end
<code>defaultCount</code>	<code>1000</code>	Max rows returned per search
<code>pollIntervalMs</code>	<code>2000</code>	How often to poll for job completion (ms)
<code>maxPollAttempts</code>	<code>30</code>	Max poll attempts before timeout (30 × 2s = 60s)

How HTTP calls work internally

The library uses native Service Suite HTTP commands throughout. No browser-style `fetch` or external modules are used.

Operation	Service Suite command used
Submit search job	<code>createPost(url, body, "application/x-www-form-urlencoded") + process(call)</code>
Poll job status	<code>createGet(url) + process(call)</code>
Fetch results	<code>createGet(url) + process(call)</code>
Per-request auth header	<code>call.addHeader("Authorization", "Bearer ...")</code>
Read response body	<code>resp.toStringContent()</code>
Read HTTP status code	<code>resp.status</code>
Sleep between polls	<code>java.lang.Thread.sleep(ms)</code>
Base64 encode (Basic auth)	<code>Java.type("java.util.Base64").getEncoder().encodeToString(...)</code>

API Reference

Query methods

Method	Returns	Description
<code>splunk.search(spl, opts)</code>	<code>Array</code>	Run any SPL query and return all result rows
<code>splunk.searchOne(spl, opts)</code>	<code>Object null</code>	Return only the first result row, or null if none
<code>splunk.count(spl, opts)</code>	<code>Number</code>	Return the number of matching events
<code>splunk.getField(spl, field, opts)</code>	<code>String</code>	Return a single field value from the first row
<code>splunk.getFields(spl, fields[], opts)</code>	<code>Object</code>	Return multiple field values from the first row

<code>splunk.getAllValues(spl, field, opts)</code>	Array	Return one field's value across all rows
<code>splunk.waitForEvent(spl, opts)</code>	Array	Poll until at least one matching event appears

All query methods accept an optional `opts` object: `{ earliest, latest, count }`. `waitForEvent` also accepts `pollMs` and `maxAttempts`.

Assertion methods

Method	Description
<code>splunk.assertCount(rows, n, msg)</code>	Fails if row count $\neq$ n
<code>splunk.assertAtLeast(rows, n, msg)</code>	Fails if fewer than n rows returned
<code>splunk.assertFieldEquals(rows, field, value, msg)</code>	Fails if the field on row[0] $\neq$ expected value
<code>splunk.assertFieldContains(rows, field, substring, msg)</code>	Fails if the field on row[0] does not contain the substring
<code>splunk.assertNoErrors(index, earliest, msg)</code>	Fails if any ERROR or FATAL entries exist in the time window
<code>splunk.assertEventOccurred(spl, msg, opts)</code>	Combined search + <code>assertAtLeast(1)</code> in one call

Utility methods

Method	Description
<code>splunk.storeResult(rows, field, lockerKey)</code>	Store a field value from row[0] into TheLocker for downstream scripts via <code>retrieve(lockerKey)</code>

Usage Examples

Example 1 — Assert a login audit event was written

After executing a login API call, verify that Splunk captured the corresponding audit entry:

```
/**
 * @aiq.servicesworkbench
 */
include('{ds}/shared/splunk.js');
splunk.init({
  host: $SPLUNK_HOST,
  token: $SPLUNK_TOKEN,
  defaultIndex: 'audit',
});
var testUser = $TEST_USERNAME;
splunk.assertEventOccurred(
  'index=audit action=login userId="' + testUser + '" result=SUCCESS',
  'Login audit event must appear in Splunk after successful authentication',
  { earliest: '-5m' }
);
log('✓ Login audit event confirmed in Splunk');
```

Example 2 — Validate field values on a specific transaction

Retrieve a transaction log by its ID and assert that its status and currency fields are correct:

```
/**
 * @aiq.servicesworkbench
 */
include('{ds}/shared/splunk.js');
splunk.init({
  host: $SPLUNK_HOST,
  token: $SPLUNK_TOKEN,
});
var txnId = _retrieve('LAST_TXN_ID'); // value stored by an upstream test case
var rows = splunk.search(
  'index=payments txnId="' + txnId + '" | head 1',
  { earliest: '-10m' }
);
splunk.assertAtLeast(rows, 1, 'Transaction ' + txnId + ' must appear in Splunk');
```

```
splunk.assertFieldEquals(rows, 'status', 'APPROVED', 'Transaction must be APPROVED');
splunk.assertFieldEquals(rows, 'currency', 'USD', 'Currency must be USD');
log('✓ Transaction ' + txnId + ' validated – status=APPROVED, currency=USD');
```

Example 3 — No-error assertion after a critical flow

After running a checkout scenario, assert that no ERROR or FATAL entries were written to the application log index during the test window:

```
/**
 * @aiq.servicesworkbench
 */
include('{ds}/shared/splunk.js');
splunk.init({
  host: $SPLUNK_HOST,
  token: $SPLUNK_TOKEN,
  defaultIndex: 'app_logs',
});
splunk.assertNoErrors('app_logs', '-10m',
  'No ERROR or FATAL log entries expected after checkout flow'
);
log('✓ No errors found in Splunk after checkout flow');
```

Example 4 — Wait for an async event

Some flows write to Splunk with a delay (e.g. payment processors, message queues). Use `waitForEvent` to poll rather than failing immediately:

```
/**
 * @aiq.servicesworkbench
 */
include('{ds}/shared/splunk.js');
splunk.init({
  host: $SPLUNK_HOST,
  token: $SPLUNK_TOKEN,
});
var txnId = _retrieve('LAST_TXN_ID');
// Poll every 3 seconds, up to 10 times (30s total)
var rows = splunk.waitForEvent(
  'index=payments event=PAYMENT_CONFIRMED txnId="' + txnId + '"',
  {

```

```

    earliest: '-5m',
    pollMs: 3000,
    maxAttempts: 10
  }
);
splunk.assertFieldEquals(rows, 'processorCode', '00',
  'Processor must return approval code 00'
);
log('✓ Async payment confirmation found in Splunk');

```

Example 5 — Extract a value and store it in TheLocker

Retrieve a generated value from Splunk and make it available to downstream test cases:

```

/**
 * @aiq.servicesworkbench
 */
include('{ds}/shared/splunk.js');
splunk.init({
  host: $SPLUNK_HOST,
  token: $SPLUNK_TOKEN,
});
var testUser = $TEST_USERNAME;
var rows = splunk.search(
  'index=sessions userId="' + testUser + '" | sort -_time | head 1',
  { earliest: '-5m' }
);
splunk.storeResult(rows, 'sessionId', 'SPLUNK_SESSION_ID');
// Downstream scripts: var sid = _retrieve('SPLUNK_SESSION_ID');
log('✓ Session ID extracted from Splunk and stored in TheLocker');

```

Example 6 — SPL aggregation: validate counts across all hosts

Use a `| stats` pipe to check that no host served 5xx errors on a specific endpoint:

```

/**
 * @aiq.servicesworkbench
 */
include('{ds}/shared/splunk.js');

```

```
splunk.init({
  host: $SPLUNK_HOST,
  token: $SPLUNK_TOKEN,
});
var rows = splunk.search(
  'index=api endpoint="/checkout" | stats count(eval(status>=500)) as server_errors by host | where
server_errors=0',
  { earliest: '-30m' }
);
log('✓ ' + rows.length + ' host(s) had zero 5xx errors on /checkout in the last 30 minutes');
```

Example 7 — getAllValues: check for unexpected HTTP status codes

Retrieve all HTTP status codes from recent API calls and fail if any 5xx appears:

```
/**
 * @aiq.servicesworkbench
 */
include('{ds}/shared/splunk.js');
splunk.init({
  host: $SPLUNK_HOST,
  token: $SPLUNK_TOKEN,
});
var statuses = splunk.getAllValues(
  'index=api endpoint="/api/order" | head 200',
  'status',
  { earliest: '-5m' }
);
var badStatuses = statuses.filter(function(s) {
  return parseInt(s) >= 500;
});
assert(badStatuses.length === 0,
  'Found ' + badStatuses.length + ' 5xx response(s) on /api/order: ' + badStatuses.join(', '));
log('✓ All ' + statuses.length + ' /api/order responses were below 500');
```

Error handling

All library methods throw a descriptive error on failure. These surface directly in the AIQ test report with context including the HTTP status code, Splunk SID, field name, or assertion mismatch. No additional `try/catch` is required — let errors propagate so the scenario fails with a clear message.

Common issues

Symptom	Likely cause	Fix
HTTP 401 on job creation	Invalid or expired token	Regenerate the token in Splunk Settings → Tokens
Connection refused / timeout	TestNode cannot reach Splunk port 8089	Check EC2 security groups and firewall rules
<code>assertAtLeast</code> fails even though event exists	Log ingestion lag — event not yet indexed	Use <code>splunk.waitForEvent()</code> instead of <code>splunk.search()</code>
Search times out	Broad query or slow Splunk cluster	Narrow the SPL time window or add more filters; increase <code>maxPollAttempts</code>
Field not found error	Field name differs from what Splunk extracts	Run the same SPL in the Splunk UI and verify field names in the results table

Best practices

- **Always use DPL variables** for `SPLUNK_HOST` and `SPLUNK_TOKEN`. Never hardcode connection details in scripts.
 - **Keep time windows short** (e.g. `-5m` or `-10m`) to keep searches fast and assertions specific to the current test run.
 - **Use** `| head N` in SPL to cap result sizes when you only need a few rows.
 - **Prefer** `waitForEvent` over a fixed sleep for async flows — it stops as soon as the event appears.
 - **Use** `assertNoErrors` as a blanket post-scenario health check in addition to specific field assertions.
 - **Use** `storeResult` to pass Splunk values into TheLocker rather than re-querying Splunk in every downstream script.
 - **Call** `splunk.init()` **once per script** at the top, before any other `splunk.*` call.
-

