LOGZILLA DOCUMENTATION

Lua Engine Walkthrough

End-to-end LogZilla App example that parses SaaS webhook JSON with a Lua rule and ships dashboards and triggers for the normalized fields

Data Transforms · Generated April 27, 2026 · logzilla.ai/docs/data-transforms/lua-app-walkthrough

Lua Engine Walkthrough

This use case shows how to package parsing and enrichment logic together with dashboards and triggers as a LogZilla App. The example remains vendor-neutral and focuses on events posted via the HTTP Event Receiver.

Scenario

- A SaaS system posts webhook events to `/incoming`.
- The organization needs normalized fields, searchable tags, and a dashboard and alerts relevant to these events.
- The solution should be packaged for repeatable deployment.

Goal

- Normalize incoming events and add durable, searchable tags.
- Provide dashboards and triggers aligned to the normalized fields.
- Package everything as a LogZilla App and manage it via the Applications UI.

Prerequisites

- Permission to manage applications in Settings → Applications.
- Ability to post test data through the HTTP Event Receiver (https://www.logzilla.ai/docs/receiving-data/http-event-receiver).
- Access to rule management commands under Command Line Tools —- Data Commands (https://www.logzilla.ai/docs/command-line-tools/data-commands) for testing.

Plan

Define inputs and outputs

- Inputs: representative sample payloads from the SaaS webhook.
- Outputs: normalized fields and tags the organization will search on and chart in dashboards.

Create the Lua rule that processes webhook JSON payloads.
Extract key fields and add searchable tags.
Package the rule with a dashboard and trigger as a LogZilla App.
Install and verify the app via the Applications UI.

Implementation

Step 1: Create the Lua Rule
Create the file `rules/500-webhook-parser.lua`:

```
-- Webhook Parser Rule
-- Processes JSON payloads from HTTP Event Receiver
-- Sample webhook: {"event_type": "user_login", "user_id": "john.doe", "ip_address": "192.168.1.100",
"status": "success"}

function process(event)
    -- Only process events from the HTTP receiver for our webhook app
    if event.program ~= "http_receiver" or event.extra_fields._url_app ~= "webhook_parser" then
        return
    end

    -- Skip if already processed by an upstream rule
    if event.extra_fields._source_type ~= nil then
        return
    end

    -- Set basic event identification
    event.program = "WebhookApp"

    -- Extract and normalize key fields from the JSON payload
    for k, v in pairs(event.extra_fields) do
        -- Skip internal LogZilla fields
        if not starts_with(k, "_") and not starts_with(k, "tags[") then
            -- Map specific fields to user tags
            if k == "event_type" then
                event.user_tags["EventType"] = v
            elseif k == "user_id" then
                event.user_tags["UserID"] = v
            elseif k == "ip_address" then
                event.user_tags["SourceIP"] = v
            elseif k == "status" then
                event.user_tags["Status"] = v

                -- Add severity based on status
                if v == "failed" or v == "error" then
                    event.user_tags["Severity"] = "High"
                elseif v == "warning" then
                    event.user_tags["Severity"] = "Medium"
                else
                    event.user_tags["Severity"] = "Low"
                end
            end
        end
    end

    -- Create a readable message from the extracted data
    local event_type = event.user_tags["EventType"] or "unknown"
    local user_id = event.user_tags["UserID"] or "unknown"
    local status = event.user_tags["Status"] or "unknown"

    event.message = string.format("Webhook Event: %s for user %s - Status: %s",
                                  event_type, user_id, status)
```

```
    -- Mark event as processed by this app to avoid re-processing
    event.extra_fields._source_type = "webhook"
end
```

Best practice: Guard rules to avoid re-processing. Check an internal marker (for example, `event.extra_fields._source_type`) at the start of the rule and set it after successful processing. This reduces unnecessary work when events are revisited or flow through multiple rules.

Step 2: Create App Configuration

Create `config/config.yaml`:

```yaml
---
# Webhook Parser Configuration
enabled: true
severity_mapping:
  failed: "High"
  error: "High"
  warning: "Medium"
  success: "Low"
```

Step 3: Create App Metadata

Create `meta.yaml`:

```yaml
---
name: Webhook Parser
description: Parses and enriches webhook events from external systems
version: '1.0'
author: Your Organization
```

Step 4: Test the Rule

Test the Lua rule with sample data:

```
# Validate the rule syntax
logzilla rules validate rules/500-webhook-parser.lua

# Test with sample webhook data
curl -X POST http://your-logzilla-server/incoming/raw/webhook_parser \
  -H "Authorization: token YOUR_GENERATED_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"event_type": "user_login", "user_id": "john.doe", "ip_address": "192.168.1.100", "status":
"success"}'

# Check if events are being processed correctly
logzilla events search --limit 10 --filter program=WebhookApp
```

Step 5: Package and Deploy the App

Create the complete app directory structure:

```
webhook_parser/
├── meta.yaml
├── config/
│   └── config.yaml
└── rules/
    └── 500-webhook-parser.lua
```

Install the app via the Applications UI or CLI:

```
# Create the app and note the printed app_dir
logzilla apps create webhook_parser

# Validate without installing
logzilla apps test webhook_parser

# Install by code
logzilla apps install webhook_parser

# Verify installation
logzilla apps list
```

Test the complete workflow:

```
# Send test webhook
curl -X POST http://your-logzilla-server/incoming/raw/webhook_parser \
  -H "Authorization: token YOUR_GENERATED_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"event_type": "user_logout", "user_id": "jane.smith", "status": "failed"}'

# Verify tags were created
logzilla events values --scope tags --limit 20
```

Verify App

After installation, verify the app is working by checking:

Events are being parsed: Search for `program=WebhookApp`

Tags are being created: Look for `EventType`, `UserID`, `SourceIP`, `Status`, `Severity` tags

Messages are readable: Verify the formatted message content

Sample Test Data

Use this sample webhook payload for testing:

```
curl -H 'Content-Type: application/json' \
     -H 'Authorization: token YOUR_GENERATED_TOKEN' \
     -X POST -d '{
       "events": [ {
         "host": "webhook.source.local",
         "program": "http_receiver",
         "message": "saas webhook",
         "extra_fields": {
           "source": "saas",
           "tenant": "acme",
           "action": "user_login",
           "status": "success"
         }
       } ] }' \
     'http://lzserver.company.com/incoming'
```

Design normalized fields and tags

- Define a concise set of fields and tags used by dashboards and triggers (for example, `tenant`, `action`, `resource`, `status`, `severity`, and `site`).
- Avoid high-cardinality values where possible.

Implement transformation logic (high level)

- Lua rules: extract multiple values from the payload, apply conditional branches for message variants, normalize severity, and build a human-readable message when necessary.
- Rewrite rules (optional): apply simple match-and-modify normalization such as standardizing `program` or adding a static `device_role`.

Prepare dashboards and triggers

- Dashboards: present trends and breakdowns across the normalized fields (for example, actions over time, top tenants, failures by resource).
- Triggers: alert on important conditions, such as repeated failures or high-severity events.

Install and manage the app

- Use Settings → Applications to install the packaged app.
- The App Store shows available and installed apps, details, and actions.
- Installing enables the app's rules, dashboards, and triggers.

Verification

- Post representative events via the HTTP Event Receiver and confirm:
  - Normalized fields and tags are present in search results.

- Dashboards populate as expected.
- Triggers evaluate correctly and produce the intended actions.

```
# Review recent rule errors
logzilla rules errors

# Parser performance and health
logzilla events parser-stats

# Inspect values observed for fields and tags
logzilla events values --scope fields --limit 50
logzilla events values --scope tags --limit 50
```

Troubleshooting

- Verify payload ingestion and parser health:

```
logzilla events parser-stats
```

- Check for rule errors:

```
logzilla rules errors
```

- Inspect fields and tags used by dashboards and triggers:

```
logzilla events values --scope fields --limit 50
logzilla events values --scope tags --limit 50
```

- Reload rules after changes:

```
logzilla rules reload
```

Rollout and maintenance

- Version the app and record notable changes and deprecations.
- Keep normalized fields stable to protect dashboards and alerts.
- Review rule errors and parser metrics immediately after updates. {{ ... }}

Notes

- Apps are the preferred vehicle for distributing transformation rules with dashboards and triggers for consistent outcomes.
- Use Lua rules for complex extraction and enrichment; supplement with rewrite rules for simple normalization where appropriate.
- Avoid exposing sensitive values in tags or messages.

Related reading

- Creating custom apps (https://www.logzilla.ai/docs/data-transforms/creating-custom-apps)
- Lua rules (https://www.logzilla.ai/docs/data-transforms/lua-rules)
- Rewrite rules (https://www.logzilla.ai/docs/data-transforms/rewrite-rules)
- Testing and verification (https://www.logzilla.ai/docs/data-transforms/testing-and-verification)
- LogZilla Apps (https://www.logzilla.ai/docs/administration/logzilla-apps)
- Command Line Tools —- Data Commands (https://www.logzilla.ai/docs/command-line-tools/data-commands)