LUX ©

Plugin Development Kit (PDK)

Version 1 – July 2014

Version 2 – December 2016

#hgm^gml

Introduction

LUX Real-time Analytic Engine and Plugins Overview

Developing Plugins with the PDK

- Setting up a Development Environment
- PDK Project Layout
- Creating an Eclipse project for the PDK
- Compiling the Sample Plugins
- Deploying the Sample Plugins to LUX

Example Plugins

- WeatherIngest
- WeatherEnrichment
- WeatherAnalytic
- WeatherConsoleAlerter
- WeatherEventOutput
- Skeleton Plugins: MyIngester, MyEnrichment, MyAnalytic, MyAlerter, and MyEventOutput
- Configuration Files
  - conf/engine.properties
  - conf/ae.xml
- Running the Examples
- Running the Examples with EventOutput

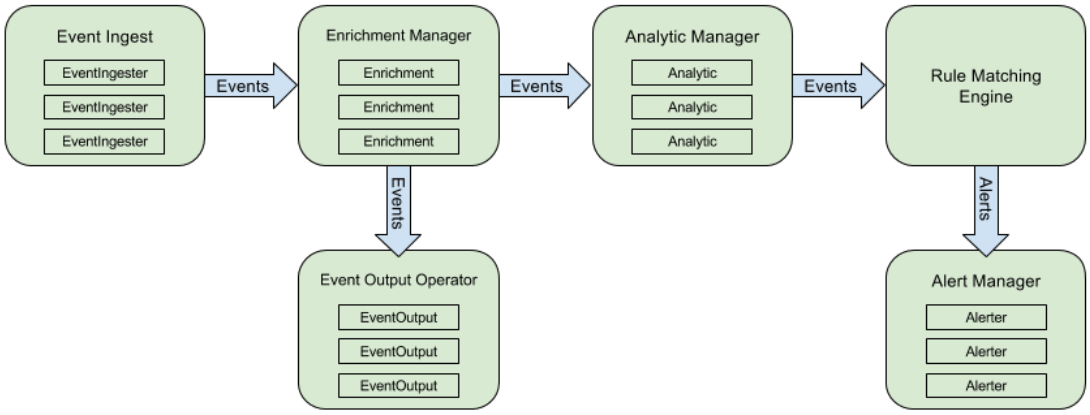
)gmkh]n\mbhg

The LUX Plugin Development Kit (PDK) enables a Java programmer to develop plugins that extend the functionality of the LUX Real-Time Analytics Engine (LUX). This guide walks a developer through the tasks of loading the PDK project into the eclipse workbench, writing lightweight extensions to the bare bones plugins provided in the PDK, and installing plugins into a an instance of LUX.

,58 Zg] Oen`bgl /o^kob^p

LUX ingests events from a wide variety of sources, enriches the events with data from other sources, and inspects, analyzes, correlates and filters the events to produce alerts.

Here is a conceptual overview of the LUX event pipeline, showing the points at which custom plugins can be added to modify the system's behavior (this is a simplification compared to some other diagrams of the lux pipeline; several components which do not have PDK plugins have been folded into the Rule Matching Engine):



Events are created in the EventIngest by EventIngester plugins, which usually fetch input data from some external source (e.g. JMS, data files, REST services, etc.) but could also generate events algorithmically. Events are created as strings by the EventIngester plugins and inserted into event streams by the EventIngest operator.

Once created, Events then go to the Enrichment Manager, which passes the events to a sequence of enrichment plugins. Enrichment plugins can attach additional information in the form of properties to events but can't otherwise modify them. Some enrichments might just analyze an events' content and attach computed results, others look up and attach data from an external source. Enrichments operate on individual events and should be conceptually stateless, so that they produce the same results whether events all go through the same instance of an enrichment in sequence or are distributed among multiple instances of the same enrichment in parallel.

After enrichment, events are passed to the Analytic Manager, and Matching Engine for further processing. They enriched events are also concurrently passed to the Event Output, if that operator is enabled.

The Analytic Manager runs a set of plugins called analytics. Unlike enrichments, analytics can't modify an Event at all. Instead, they create new Analytic Events, which are usually bundles of input events with additional properties added. Analytics are usually stateful, retaining information from the stream of Events they observe over time to produce their output. Scaling up a single analytic by distributing it over multiple instances is thus usually more difficult, requiring the separate instances to communicate shared state with each other. A Distribution Manager utility is provided which can help with this.

Both the original enriched Events and those created by the Analytics are sent to the rule matching engine. This is the component that checks Events against rules configured in the LUX UI and produces alerts based on the matches. The LUX PDK does not currently provide any customization options for this stage of the pipeline.

Alerts bundle events with information about the rules that matched them. An event will produce a separate alert for each rule that it matched. These alerts are received by the AlertManager, the final component in the event pipeline, and each configured alerter plugin. The standard alerters send alert information back to the LUX UI for display, but alerters can be used to send this information somewhere else; for example, custom alerters can be used to forward the LUX Engine's output to another software system for additional processing.

The EventOutputOperator is an optional operator; it isn't active by default in the current version of the LUX Engine. It passes events to a set of EventOutput plugins, which operate in parallel to the analytics and matching engine, and is intended as an efficient way to copy event data to other recipients without requiring them to have been processed by the rest of the event pipeline. For example, an EventOutput might store the events, or forward them to another system.

Plugin Type	Description
EventIngester	Ingests data from an external data source, transforms the data into a series of events and provides the events to the LUX engine.
Enrichment	Enriches individual events by adding properties to the event. An enrichment plugin should be conceptually stateless.
Analytic	Observes a stream of events and produces its own stream of events, which can include bundles of events from the input.
Alerter	Outputs alerts from the LUX Engine to external recipients, such as the LUX UI
EventOutput	Copies events to an external receiver in parallel with the rest of the pipeline

$^o^ehibg` 0en`bgl pbma ma^ 0$+

3^mmbg` ni Z $^o^ehi f ^gm %gobkhg f ^gm

The requirements for building with the LUX PDK are mostly the same as running the LUX Engine, plus some additional build tools.

Software Requirement	Description
OS	The engine is developed and deployed on Centos and RHEL OS with x64 architecture. It has also been deployed to PowerPC. For best results, install the PDK on a Centos or RHEL system.
JDK	Install the Java Standard Edition Development Kit 7 latest update. (http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html) OpenJDK 7 can also be used (**`yum install openjdk-1.7.0-devel`**)
Apache Ant	Install Ant 1.8.4 from http://ant.apache.org
Eclipse IDE	Install the Eclipse platform from the Eclipse Standard download (http://www.eclipse.org/downloads/) version 4.3.2 at the time this document was written. You can use eclipse to edit the PDK source code and run the unit tests. Make sure that the eclipse workspace is configured with the 1.7 jre.

0$+ 0khc^\m ,Zrhnm

The LUXPDK directory contains the following items:

src/	Directory containing source code for the sample plugins
test/	Directory containing source code for the sample plugin unit tests
lib/	Directory containing library files needed for building plugins
test_data/	Directory containing data used for testing plugins. This directory is copied to EngineMain/data/test_data/ by the deploy.sh script
conf/	Directory containing configurations for plugins. This directory is copied to EngineMain/data/conf/ by the deploy.sh script
build.xml	Ant build file for compiling the source and creating plugin jars. Outputs files to bin/ and

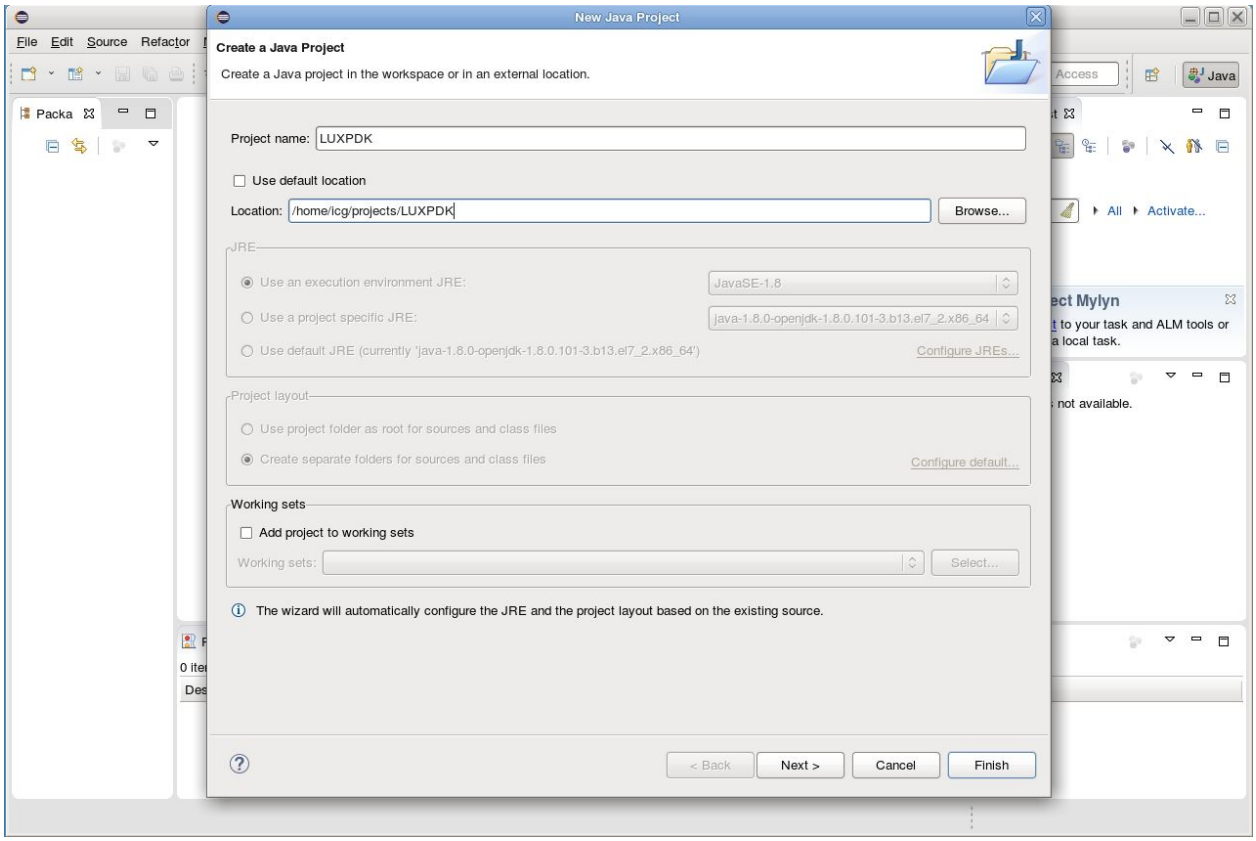
	plugins/
build.sh	Convenience script for cleaning and rebuilding plugin jars; calls ant on targets in build.xml
deploy.sh	Convenience script for installing plugins, configurations, and test data to an instance of the LUX Engine
bin/	Created during build; Directory containing compiled .class files from src
plugins/	Created during build; Directory containing compiled plugin jar files. This directory is copied to EngineMain/data/conf/ by the deploy.sh script

#k^Zmbg` Zg %\ebi l^ ikhc^\m_hk ma^ 0$+

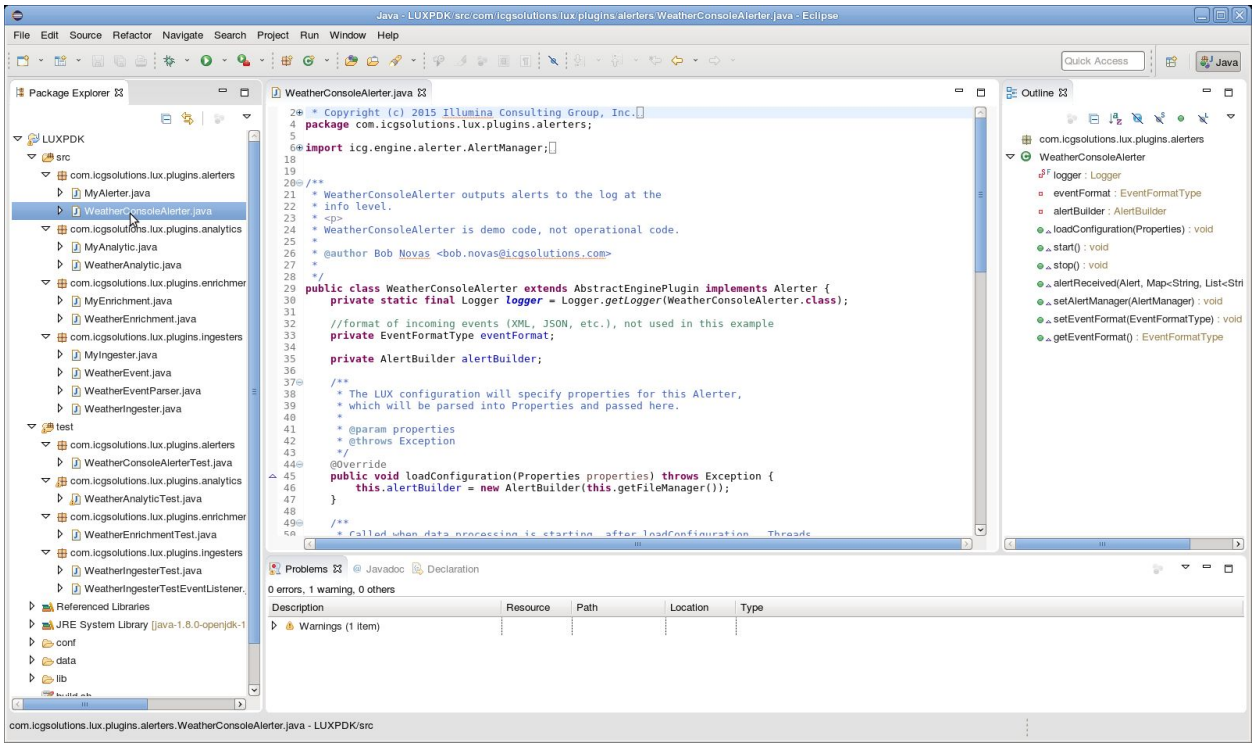
Load the “LUXPDK” folder into eclipse as a new project. In the eclipse menu, select

```
File -> New -> Java Project
```

to open a dialog titled “Create Java Project”. Uncheck the checkbox “Use default location”, click the “Browse ĺ ” button, and navigate the to the “LUX PDK” folder (or just type the full path into the Location field).



Click “Finish” on the dialog. This should create the “LUXPDK” project in the Package Explorer, as shown (expanded) on the left hand side of the image below. The right hand side shows the WeatherConsoleAlerter.java source open in the eclipse editor.



#h f ibebg` ma^ 3Z f ie^ Oen`bgl

An ant build file, build.xml, is provided with the following targets:

- **compile** – compiles the code
- **test** – runs the unit tests
- **export-[alerters|intesters|analytics|enrichments|event-outputs|all-plugins]** – exports the jar file(s)
- **clean** – cleans the LUX PDK folder

The build.xml ant script can be used to compile and test the sample plugins, as well as any plugins added to the same source tree, and export them to jar files that can be used by the LUX Engine. Note that exporting with this script involves special treatment of certain java packages:

com.icgsolutions.lux.plugins.alerters	Alerter plugin; exported to MyAlerter.jar
com.icgsolutions.lux.plugins.analytics	Analytic plugin; exported to MyAnalytic.jar
com.icgsolutions.lux.plugins.enrichments	Enrichment plugin; exported to MyEnrichment.jar
com.icgsolutions.lux.plugins.ingesters	EventIngester plugin; exported to MyIngesters.jar
com.icgsolutions.lux.plugins.outputs	EventOutput plugin; exported to MyEventOutputs.jar

Each of these types of plugin goes in a different directory in the LUX Engine data directory; the export tasks in build.xml create that directory structure under a 'plugins' directory in the LUXPDK directory.

For convenience, there is a build.sh script in the LUXPDK directory that will clean and re-compile and export all the plugins in the source tree.

$^iehrbg`ma^3Zfie^Oen`bglmh,58

The LUX PDK includes a convenience script, deploy.sh, that will copy compiled plugins, configuration information, and data to an instance of the LUX Engine on the same system. That is, it will copy the contents of the plugins/, conf/, and test_data/ directories into the appropriate subdirectories under the engine's EngineMain directory. Note that this will overwrite files with the same name already in that directory, so you may wish to make a backup copy of EngineMain before doing this.

When running the deploy.sh script, the location of the LUX Engine's EngineMain directory must be specified on the command line. Example:

deploy.sh /usr/local/lux/LUXEngine/EngineMain

Alternatively, you can put this value in an environment variable named LUX_ENGINEMAIN_PATH and run deploy.sh with no parameters. This is usually done by adding a line to .bashrc or .bash_profile like this:

export LUX_ENGINEMAIN_PATH=/usr/local/lux/LUXEngine/EngineMain

After deploying the files to EngineMain, the LUX Engine must be restarted if it is currently running for changes to take effect.

%qZfie^Oen`bgl

7^Zma^k)g`^lm

WeatherIngester is an EventIngester that periodically reads an input file scraped from the national weather alert RSS feed (http://alerts.weather.com) and converts the weather alerts found in the file to LUX Events.

WeatherIngester is a Java class that extends AbstractEventIngester. The primary methods defined in AbstractEventIngester that an ingest plugins needs to implement are loadConfiguration, start, and stop.

loadConfiguration is called shortly after construction; it receives a java.util.Properties object, which is read from a .properties file specified in the individual plugin's configuration in conf/engine.properties. In WeatherIngester, two properties are read:

Property Name	Property Value
weather.alert.ingest.file.path	Path of the input .xml file, relative to EngineMain/data
weather.alert.ingest.cycle.time.seconds	Delay between repeated readings of the input file, in seconds (parsed as a Long)

Additional initialization that the plugin requires can be done in loadConfiguration. In WeatherIngester, an instance of WeatherEventParser is created and initialized. WeatherEventParser is what does the work of converting the rss feed's xml into LUX Events (in this example plugin, the use of WeatherEventParser is hard-coded, but many of the more general-purpose EventIngesters will read the class name of an EventParser as part of their config properties and create an instance of that EventParser). If an exception is thrown out of loadConfiguration, the engine will fail on startup.

start is called after all event ingesters have been initialized. In an EventIngester, start usually starts worker threads that will do the actual work of reading data, constructing events, and submitting them to the eventListener. The start method must not itself block, as the engine cannot proceed until it returns. If start throws an exception, the EventIngester that threw it will be marked as inactive, but the engine will continue without it.

eventListener is a field defined in AbstractEventIngester and populated during initialization. It defines a number of methods named eventReceived. In most cases, the EventIngester will simply pass the event string to this method, but there are variants that will accept other information such as a different eventTime or streamName to use, or a map additional EventProperties to add (EventProperties are usually only added by an Enrichment or Analytic).

In WeatherIngester, a single workerThread is created. This thread simply runs in a loop where it reads and parses the input file (using WeatherEventParser), then sleeps for the configured time interval, until stop is called.

stop is called when the engine shuts down, or when a command is received to stop a particular ingester. stop should terminate any worker threads the plugin created and do any other cleanup that needs to be done.

7 ^Zma^k%gkb\a f ^gm

WeatherEnrichment is an Enrichment that compares a weather alert event's 'expires' attribute to a specified fixed expiration date. If the 'expires' date is after the expiration date, the difference is calculated and attached to the event as a property named 'expired'.

WeatherEnrichment is a Java class that extends AbstractEnrichment. The primary methods defined in AbstractEnrichment that an enrichment needs to implement are loadConfiguration and processEvent.

loadConfiguration in an enrichment works the same way as in an event ingester. The properties passed to an enrichment's loadConfiguration function are read from the enrichment's definition in conf/ae.xml. In WeatherEnrichment, two properties are read:

Property Name	Property Value
expires_path	The XPath to select the 'expires' date string within the event string. The date string must be formatted like this: 2014-06-12T00:00:00-0000
expire_on_date	The date to compare the 'expires' date to, in the same format

processEvent is where an enrichment does its work; the function receives an EventDocument. An EventDocument wraps an event string and attaches additional information, such as properties that have been added by other enrichments. Information is accessed from an EventDocument with an EventQuery, which can wrap an XPath query to apply to an xml string within an EventDocument. Additionally, EventQueries can be used to access properties that have been attached to events using the following syntax: EventQuery myPropertyNameQuery = new EventQuery("${propertyName}").

processEvent should return a guava Multimap containing any new EventProperties it wishes to add to the event. The multimap maps property names to property values; the values are EventProperty objects, which can wrap strings or numbers, and also attach a display name for use by the LUX UI.

As with other plugins, Enrichments also have start and stop methods, but since enrichments are generally stateless and process events synchronously they don't often need to implement these methods.

7 ^Zma^k!gZermb\

WeatherAnalytic is an analytic that keeps track of the weather events it's seen and outputs an analytic event if it sees the same event more than a certain number of times in a certain time interval. What attribute is used to determine whether two events are 'the same' is configurable.

WeatherAnalytic is a Java class that extends AbstractAnalytic. The primary methods defined in AbstractAnalytic that an analytic needs to implement are loadConfiguration, processEvent, start, and stop.

The methods loadConfiguation, start, and stop work the same in analytics as they do in enrichments. Unlike enrichments, analytics are generally stateful and operate on their own threads, so they usually need to implement start and stop.

WeatherAnalytic's loadConfiguration parses several properties from its configuration (specified in the analytic's element in conf/ae.xml):

Property Name	Property Value
value_path	XPath to select the attribute from an event that is compared to other events to determine whether they're 'the same'
event_threshold	The (integer) number of events that the analytic must see with the same value from from the value_path query in order to output an analytic event
event_window_seconds	The (float or double) number of seconds that limits the time span within which the matching values must be found. Events outside this time window are discarded.
purge_interval_seconds	The (float or double) number of seconds that specifies how often the analytic checks to purge old events from its store. Defaults to 30.

WeatherAnalytic keeps track of the values and events it's seen in a ConcurrentHashMap. The key in the map is the String result returned by value_path. The value in the map's entries is a list of event times (as longs) that can be checked to determine whether a previously seen event is still within the analytic's configured time window.

WeatherAnalytic's processEvent method checks the event's eventTime, queries its value_path value, and adds it to its map. It then checks all the old entries in the map with the same key, removes eventTimes that are older than the time window, and, if there's still enough events with the same key, outputs an AnalyticEvent containing the most recent event along with 'description' and 'eventCount' properties.

Note: While an analytic can keep a reference to an EventDocument in order to add it to an AnalyticEvent later (and often analytics will keep around a large number of them), any EventQueries on the event must be done within the processEvent method. This is so that the engine can efficiently discard extra data (e.g. a parsed DOM structure) associated with the event while keeping around only the unparsed event string and event properties.

WeatherAnalytic's start and stop methods are simple; they simply start and stop a timer with a TimerTask to check all the events in the map and evict events outside the time window (while processEvent also evicts events, it only checks events with the same value_path value as the event it's currently looking at).

7 ^Zma^k#hglhe^!e^km^k

WeatherConsoleAlerter is a simple Alerter that logs alerts to a Log4j logger.

WeatherConsoleAlerter is a Java class that extends AbstractAlerter. The primary methods defined in AbstractAlerter that an alerter needs to implement are loadConfiguration, start, stop, and alertReceived.

loadConfiguration, start, and stop work the same in an Alerter as in EventIngester. Since WeatherConsoleAlerter doesn't do much, it only performs some minimal initialization in loadConfiguration and does nothing in start or stop.

WeatherConsoleAlerter's loadConfiguration method initializes an AlertBuilder object. This is a utility class used to format alerts for submission to the LUX UI. While WeatherConsoleAlerter doesn't talk to the LUX UI, it wants to be able to log alert titles in the same format, and uses the alertBuilder to do this.

It logs the alert title as INFO in the alertReceived method.

7 ^Zma^k%o^gm/nminm

WeatherEventOutput is an EventOutput that copies weather events to the local filesystem.

WeatherEventOutput is a Java class that extends AbstractEventOutput. The primary methods that an event output needs to implement are loadConfiguation and processEvent.

loadConfiguration works the same in an event output as it does in an enrichment. In WeatherEventOutput, it parses an output directory name and an event id path from the config:

Property Name	**Property Value**
id_path	XPath to an id field in an event, which is used in forming the output file name (after being stripped of special characters)
output_dir	Output directory to write files to (default value is "event_output")

WeatherEventOutput's processEvent function extracts the event id value from the event using the configured id_value path, strips it of punctuation and whitespace, and appends '.txt' to form an output file name. It then creates the file (including the parent directory) if it doesn't already exist, and writes the event string. If the file already exists, a debug message is logged but nothing is written.

As with enrichments, an event output can implement start and stop, but usually doesn't need to.

3d^e^mhg Oen`bgl Ç - r)g`^lm^kÅ - r%gkb\a f ^gmÅ - r ! gZermb\Å - r ! e^km^kÅ Zg] - r%o^gm / nm i nm

MyIngester, MyEnrichment, MyAnalytic, MyAlerter, and MyEventOutput are skeleton plugins already defined in the appropriate packages to be built and deployed by the scripts in the PDK. The easiest way to begin building a plugin is to copy or rename one of these classes.

#hg_b`nkZmbhg &be^l

The engine configuration files in LUXPDK/conf can be deployed to a local instance of the LUX Engine with the included deploy.sh script. These configuration files specify an engine with one event ingester (WeatherIngest), one enrichment (WeatherEnrichment), one analytic (WeatherAnalytic), and one alerter (WeatherAlerter). Additionally, the rule matching engine will read its rules from the file test_date/rest_rules_weather_analytic.json (specified in conf/lux.properties), which contains a single rule that will alert on every analytic event that the WeatherAnalytic outputs.

\hg_Ø^g`bg^È i kh i ^kmb^l

The WeatherIngester is configured in engine.properties with the following lines:

```
event.ingest.1.stream.name=default_stream
event.ingest.1.classpath=com.icgsolutions.lux.plugins.ingesters.WeatherIngester
event.ingest.1.name=WeatherIngester
event.ingest.1.confpath=engine.properties
weather.alert.ingest.file.path=test_data/weatherAlerts.xml
weather.alert.ingest.cycle.time.seconds=1
```

The WeatherAlerter is configured in engine.properties with the following lines:

```
alerter.1.classpath=com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter
alerter.1.name=Console
alerter.1.maxaps=25
alerter.1.alert.format=XML
```

\hg_ØZ^Èq f e

The WeatherEnrichment is configured with the following lines:

```
<java_enrichment name="com.icgsolutions.lux.plugins.enrichments.WeatherEnrichment"
display_name="Weather Enrichment" classification="UNCLASSIFIED">
        <stream_in>default_stream</stream_in>
        <description>A PDK WeatherAlert Java Enrichment</description>
```

```
        <property name="expires_path" value="/rss/channel/item/lux:attribute[@name='cap:expires']" />
        <property name="expire_on_date" value="2014-06-12T00:00:00-0000" />
</java_enrichment>
```

The WeatherAnalytic is configured with the following lines:

```
<java_analytic name="com.icgsolutions.lux.plugins.analytics.WeatherAnalytic"
classification="UNCLASSIFIED" stream_out="WeatherAnalytic" display_name="WeatherAnalytic" >
        <stream_in>default_stream</stream_in>
        <description>Alerts when 10 of the same WeatherAlerts come in</description>
        <deadlock_check_time_millis>10000</deadlock_check_time_millis>
        <property name="value_path" value="/rss/channel/item/lux:attribute[@name='id']"/>
        <property name="event_threshold" value="10"/>
        <property name="event_window_seconds" value="11.0"/>
        <property name="purge_interval_seconds" value="30.0"/>
</java_analytic>
```

The WeatherEventOutput is configured with the following lines:

```
<event_output name="com.icgsolutions.lux.plugins.outputs.WeatherEventOutput"
display_name="Weather Event Output" classification="UNCLASSIFIED">

        <stream_in>default_stream</stream_in>
        <description>Copies weather events to weather_events/</description>
        <property name="id_path" value="/rss/channel/item/lux:attribute[@name='id']"/>
        <property name="output_dir" value="weather_events"/>
</event_output>
```

2nggbg` ma^ %qZ f ie^l

After building the sample plugins with build.sh (or ant export-all-plugins) and deploying them to the engine using deploy.sh, go to LUXEngine's bin directory and run the engine on the command line using run-lite.sh

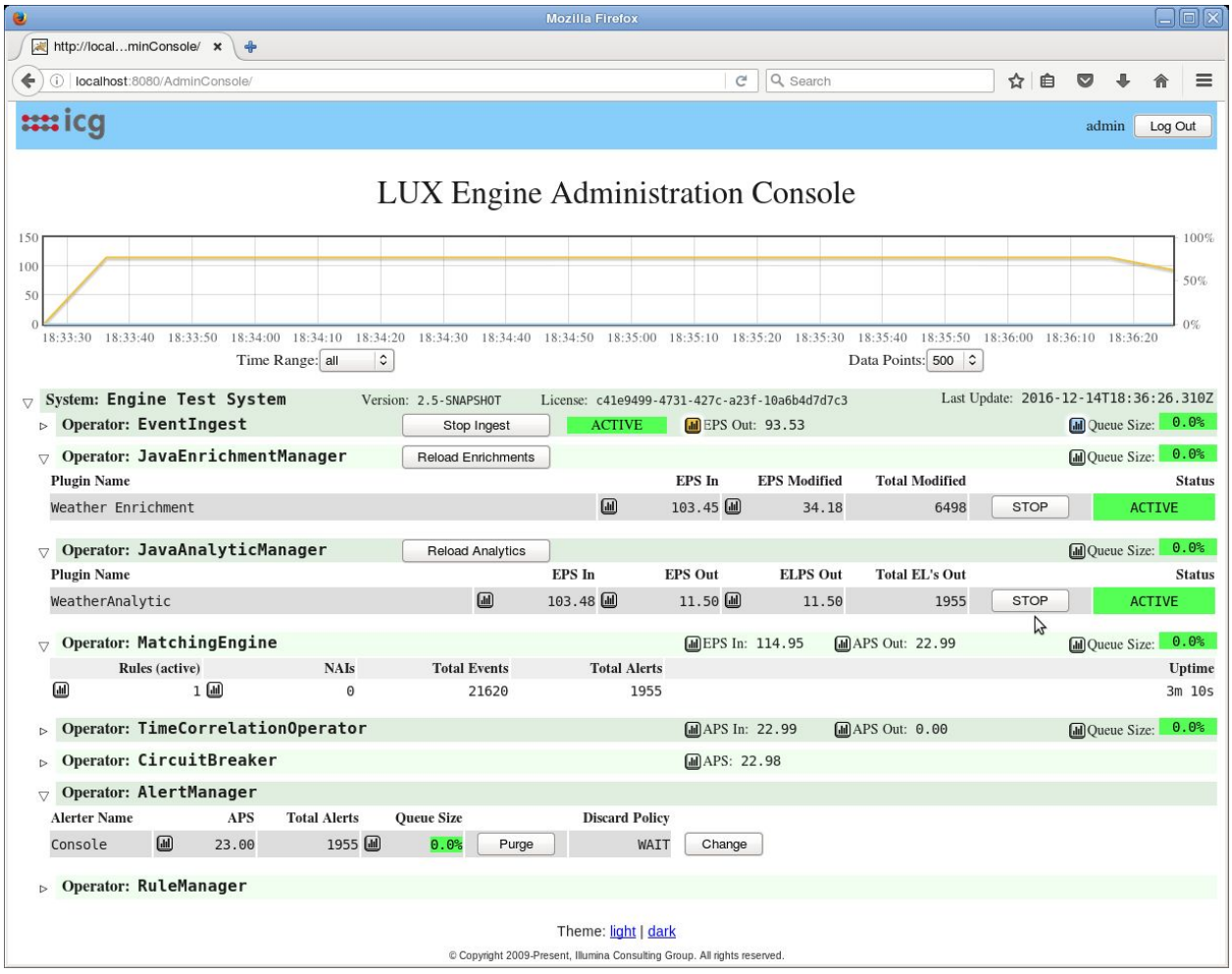
Since the WeatherIngester is configured to repeat the contents of the weatherAlerts.xml file once every second, and the WeatherAnalytic is configured to alert when it sees 10 of the same event within an 11-second time window, it should produce an analytic event for every event in the input file once every 10 seconds. The engine's single test rule should match on this analytic event and produce an alert:

```
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:30] [com.icgsolutions.lux.plugins.alerters.WeatherConsoleAlerter] - Weather Alert Repeats
INFO  [11 Dec 2016 20:03:34] [icg.matching.engine.MatchingEngine] - Engine: 1 Rules; 114.91 EPS in; 22.98 APS out; 26910 events total.
INFO  [11 Dec 2016 20:03:39] [icg.matching.engine.EnginePartition] - Updating nais...  Adding 0 nais
INFO  [11 Dec 2016 20:03:39] [icg.matching.engine.EnginePartition] - Updating rules...  Adding 1 rules
INFO  [11 Dec 2016 20:03:39] [icg.matching.engine.MatchingEngine] - Engine: 1 Rules; 114.89 EPS in; 0.00 APS out; 27485 events total.
INFO  [11 Dec 2016 20:03:40] [icg.engine.lux.rest.jersey.JerseyAdminConsoleRESTClient] - Sent console state V2, response code: 200
```

If the engine's AdminConsole webapp is running in tomcat, you can enable it by looking for the following config block in engine.properties and changing admin.console.web.enabled to true and supplying the correct URL for the webapp:

```
########## ADMIN CONSOLE ##########
#turn the admin console webpage updates on or off
admin.console.web.enabled=false
#Update frequency for REST calls, updates
admin.console.update.frequency.ms=10000
#Base URL for admin console REST services
admin.console.rest.url=http://localhost:8080/AdminConsole/rest/
```

After logging into the webapp, you should see something like this:



Notice the total number of events and alerts (in the example image, 21620 and 1955). Since there are 115 events in the input file, we expect both these numbers to be divisible by 115. We also expect the number of alerts to be a little less than 1/10th the number of events, since every 10 repetitions of each of the 115 events should produce an alert (minus some timing delays).