Example Dataflow Templates

Apache NiFi provides users the ability to build very large and complex DataFlows using NiFi. This is achieved by using the basic components: Processor, Funnel, Input/Output Port, Process Group, and Remote Process Group. These can be thought of as the most basic building blocks for constructing a DataFlow. At times, though, using these small building blocks can become tedious if the same logic needs to be repeated several times. To solve this issue, NiFi provides the concept of a Template. A Template is a way of combining these basic building blocks into larger building blocks. Once a DataFlow has been created, parts of it can be formed into a Template. This Template can then be dragged onto the canvas, or can be exported as an XML file and shared with others. Templates received from others can then be imported into an instance of NiFi and dragged onto the canvas.

For more information on Templates, including how to import, export, and work with them, please see the Template Section of the User Guide.

Here, we have a collection of useful templates for learning about how to build DataFlows with the existing Processors. Please feel free to add any useful templates below.

Template	Description	Minimum NiFi Version	Processors Used
ReverseGeoLookup_ScriptedLookupService.xml	NOTE: This template depends on features available in the next release of Apache NiFi (presumably 1.3.0) which is not released as of this writing. This example flow illustrates the use of a ScriptedLookupService in order to perform a latitude/longitude lookup to determine geographical location. The latitude and longitude values are in the incoming record, and dynamic properties in LookupRecord are provided to "find" the lat/long fields and pass them to the ScriptedLookupService. The ScriptedLookupService itself accepts a dynamic property 'googleApiKey' which is the user's Google API Key for performing lookups using the Google Reverse Lookup web API. The ScriptedLookupService contains a Groovy script which provides the lat/long values (and API Key) to the web service, and returns a String result corresponding to the geolocation. LookupRecord (and its CSVRecordSetWriter) are configured to append this field (called "location") to the incoming records.	1.3.0	LookupRecord
ProvenanceStreamRecordReaderWriter XML AVRO JSON CSV	This example flow takes advantage of NiFi's ability to stream its own provenance data through the flow which it can then read, write, route, and transform for some interesting cases. The provenance data it generates then becomes its own stream to experiment with. To do this we take advantage of the site to site reporting task for provenance, the new QueryRecord processor powered by Apache Calcite, various record readers and writers including a custom one built on the fly using Groovy all to read in the provenance stream while simultaneously writing it out in JSON, Avro, CSV, and XML. All this really helps highlight the power this makes available as we get to reuse the same components but plugin in separate concerns such as formats and schemas which we can reuse at various points in the flow. Another great benefit here is we don't have to split data into individual records before the processors can operate on them. Through the record abstraction it understands how to parse, demarcate, and reproduce the transformed output without this having to be a concern of the user or the processors themselves. This creates tremendous reuse, flexibility, and massive performance benefits all while still maintaining full fidelity of the provenance trail! • Enable site to site in nifi.properties by setting the 'nifi.remote.input.socket.port' property to 8085 for example. • Enable the new provenance repository implementation by changing the property 'nifi.provenance.repository.implementation' line to 'nifi.provenance.repository.implementation=org.apache.nifi.provenance.WriteAheadProvenanceRepository' • Import the template [here] into nifi. Drag that template onto the graph. • Select 'configure' on 'Provenance R/W' group. Enable the internal schema registry and all services. Then on the root of the flow start all components. • Now add the reporting task to send the provenance events. Top level -> Controller Settings. Select 'Reporting Tasks'. Add 'SiteToSiteProvenanceReportingTask'. Edit it to change scheduling to '5 secs'. Destination url to 'http://localhost:8080/nifi'. Input port name to 'ProvStream'. Hit apply and start it. You should start seeing data flowing within seconds. • Now go into the provenance r/w group and you can use provenance features to quickly look at the content at the various points in the flow and see how the processors are setup to get a feel for how it works.	1.2.0	
ProvenanceStreamRecordReaderWriter XML AVRO JSON CSV (1.5.0+)	This example performs the same as the template above, and it includes extra fields added to provenance events as well as an updated ScriptedRecordSetWriter to generate valid XML.	1.5.0	

NiFi_Status_Elasticsearch.xml	NiFi status history is a useful tool in tracking your throughput and queue metrics, but how can you store this data long term? This template works in tandem with the new SiteToSiteStatusReportingTask to automatically pipe status history metrics to an external Elasticsearch instance.
	
	To use this template:
	
	1. Install Elasticsearch (this template uses Elasticsearch 2.x, but you can use PutElasticsearch5 for 5.x support).
	2. Set up a template for your index:
	
	`curl -XPUT 'localhost:9200/_template/nifi_status' -d '`
	`{`
	`  "template": "nifi-status*",`
	`  "order":    1,`
	`  "aliases": { "current-nifi-status": {}  },`
	`  "mappings": { "nifi": { "properties": {`
	`        "componentType": { "type": "string", "index": "not_analyzed" },`
	`        "componentName": { "type": "string", "index": "not_analyzed" },`
	`        "componentId": { "type": "string", "index": "not_analyzed" },`
	`        "hostname": { "type": "string", "index": "not_analyzed" },`
	`        "processorType": { "type": "string", "index": "not_analyzed" }`
	`  } } }`
	`}'`
	
	3. In NiFi, add a new SiteToSiteStatusReportingTask:
	
	a. **Destination URL** should be your nifi instance's URL
	b. **Input Port Name** should be NiFi Status Input
	c. **Batch Size** can be 1000
	d. **Platform Name** should be descriptive of your NiFi instance
	e. You can use the filters to limit the types and names of metrics that are reported
	f. Configure the **SSL Context Service** if applicable
	4. Drag the NiFi_Status_Elasticsearch template to the top level of your NiFi instance and edit the PutElasticsearchHttp URL to point to your Elasticsearch instance.
	5. Play all the relevant processors and the new input port, as well as the SiteToSiteStatusReportingTask.
	
	Tip: you can use Kibana to visualize the status over time.
	
	Tip: to automatically roll over the time-bound current-nifi-status Elasticsearch index on the first of the month, drag the Rollover_NiFi_Status_Elasticsearch_Index.xml template onto your flow, configure the host name in the processors, and enable all the processors.
		1.2.0	ConvertRecord, SplitJson, EvaluateJsonPath, UpdateAttribute, PutElasticsearchHttp

Pull_from_Twitter_Garden_Hose.xml	This flow pulls from Twitter using the garden hose setting; it pulls out some basic attributes from the Json and then routes only those items that are actually tweets.		
Retry_Count_Loop.xml	This process group can be used to maintain a count of how many times a flowfile goes through it. If it reaches some configured threshold it will route to a 'Limit Exceeded' relationship otherwise it will route to 'retry'. Great for processes which you only want to run X number of times before you give up.		
simple-httpget-route.template.xml	Pulls from a web service (example is nifi itself), extracts text from a specific section, makes a routing decision on that extracted value, prepares to write to disk using PutFile.		
InvokeHttp_And_Route_Original_On_Status.xml	This flow demonstrates how to call an HTTP service based on an incoming FlowFile, and route the original FlowFile based on the status code returned from the invocation. In this example, every 30 seconds a FlowFile is produced, an attribute is added to the FlowFile that sets q=nifi, the google.com is invoked for that FlowFile, and any response with a 200 is routed to a relationship called 200.		
Decompression_Circular_Flow.xml	This flow demonstrates taking an archive that is created with several levels of compression and then continuously decompressing it using a loop until the archived file is extracted out.		
SplitRouteMerge.xml sample-input.txt	This flow demonstrates splitting a file on line boundaries, routing the splits based on a regex in the content, merging the less important files together for storage somewhere, and sending the higher priority files down another path to take immediate action.		

TwitterSolr.xml	This flow shows how to index tweets with Solr using NiFi. Pre-requisites for this flow are NiFi 0.3.0 or later, the creation of a Twitter application, and a running instance of Solr 5.1 or later with a tweets collection:
	
	`./bin/solr start -c`
	`./bin/solr create_collection -c tweets -d data_driven_schema_configs -shards 1 -replicationFactor 1`		

CsvToJSON.xml	This flow shows how to convert a CSV entry to a JSON document using ExtractText and ReplaceText.		

NetworkActvityExample.xml	This flow grabs network activity using tcpdump, then performs geo-enrichment if possible, before delivering the tcpdump entries to Kafka and HDFS.		
SyslogExample.xml	This flow shows how to send and receive messages from Syslog. It requires a Syslog server to be accepting incoming connections using the protocol and port specified in PutSyslog, and forwarding connections using the protocol and port specified in ListenSyslog. **NOTE: This template can be used with the latest code from master, or when 0.4.0 is released**	0.4.0	PutSyslog, ListenSyslog
Working_With_CSV.xml	This flow uses http://randomuser.me to generate random data about people in CSV format. It then manipulates the data and writes it to a directory. A second flow then uses ListFile / FetchFile processors to pull that data into the flow, strip off the CSV header line, and groups the data into separate FlowFiles based on the first column of each row in the CSV file (the "gender" column) and finally puts all of the data to Apache Kafka, using the gender as part of the name of the topic.	0.4.0	ListFile, FetchFile, PutKafka, RouteText, PutFile, ReplaceText, InvokeHTTP
Working_with_Logs.xml	Tails the nifi-app and nifi-user log files, and then uses Site-to-Site to push out any changes to those logs to remote instance of NiFi (this template pushes them to localhost so that it is reusable). A second flow then exposes Input Ports to receive the log data via Site-to-Site. Then data is then aggregated until the data for a single log is in the range of 64-128 MB or 5 minutes passes, which occurs first. The aggregated log data is then pushed to a directory in HDFS, based on the current timestamp and the type of log file (e.g., pushed to /data/logs/nifi-app-logs/2015/12/03 or /data/logs/nifi-user-logs/2015/12/03, depending on the type of data). **NOTE:** In order to use this template Site-to-Site must be enabled on the node. To do this, open the $NIFI_HOME /conf/nifi.properties file and set the "nifi.remote.input.socket.port" property to some open port number and set "nifi.remote.input.secure" to "false" (unless, of course, you are running in a secure environment). For more information on Site-to-Site, see the Site-to-Site Section of the User Guide.	0.4.0	TailFile, MergeContent, PutHDFS, UpdateAttribute, Site-to-Site, Remote Process Group, Input Ports
Fun_with_HBase.xml	Downloads randomly generated user data from http://randomuser.me and then pushes the data into HBase. The data is pulled in 1,000 records at a time and then split into individual records. The incoming data is in JSON Format. The entire JSON document is pushed into a table cell named "user_full", keyed by the Row Identifier that is the user's Social Security Number, which is extracted from the JSON. Next, the user's first and last names and e-mail address are extract from the JSON into FlowFile Attributes and the content is modified to become a new JSON document consisting of only 4 fields: ssn, firstName, lastName, email. Finally, this smaller JSON is then pushed to HBase as a single row, each value being a separate column in that row. At the same time, a GetHBase Processor is used to listen for changes to the Users table. Each time that a row in the Users table is changed, the row is pushed to Kafka as JSON. **NOTE:** In order to use this template, there are a few pre-requisites. First, you need a table created in HBase with column family 'cf' and table name 'Users' (This can be done in HBase Shell with the command: *create 'Users', 'cf'*). After adding the template to your graph, you will need to configure the controller services used to interact with HBase so that they point to your HBase cluster appropriately. You will also need to create a Distributed Map Cache Server controller Service (all of the default values should be fine). Finally, each of the Controller Services needs to be enabled.	0.4.0	InvokeHTTP, SplitJson, EvaluateJsonPath, AttributesToJson, PutHBaseCell, PutHBaseJSON, GetHBase, PutKafka
Hello_NiFi_Web_Service.xml	An Ad Hoc web service that "enriches" an HTTP request to port 8011 with a NiFi greeting utilizing HandleHttpRequest, HandleHttpResponse and the StandardHttpContextMap controller service		HandleHttpRequest, HandleHttpResponse, ReplaceText StandardHttpContextMap
Syslog_HBase.xml	Inserts Syslog messages to HBase. Requires creating a table in HBase: create 'syslog', {NAME => 'msg'}	0.4.0	ListenSyslog, AttributesToJson, PutHbaseJSON
GroovyJsonToJsonExample.xml	Illustrates the ExecuteScript processor using Groovy to perform JSON-to-JSON transformations	0.5.0	ExecuteScript
WebCrawler.xml	A template that takes in an initial seed URL scraps the contents of the site for more URLs. For each URL it will extract lines which match specific phrases ("nifi" in the example) email the FlowFile to an address. Also it bundles together websites then compresses and puts them to a local folder. Note: This template requires a DistributedMapCacheServer with default values to run. It is not included because at the time of creation there was no way to explicitly include a controller service with no processors referring to it.	0.5.0	CompressContent, DetectDuplicate, ExtractText, GetHTTP, InvokeHTTP, LogAttribute, MergeContent, PutEmail, PutFile, RouteOnAttribute, RouteText, SplitText, UpdateAttribute
DateConversion.xml	This flow demonstrates how to extract a date string from a FlowFile and then replace that date in the flow file with the same date in a new format.	0.6.1	ExtractText, ReplaceText

<table>
<tr><td>ConvertCSVtoCQL.xml</td><td>This template describes a flow where a CSV file (whose filename and content) contributes to the fields in a Cassandra table is processed, then CQL statements are constructed and executed.</td><td>1.0.0</td><td>PutCassandraQL, InvokeScriptedProcessor, UpdateAttribute, SplitText, ExtractText, ReplaceText, ExecuteScript</td></tr>
<tr><td>ScriptedMapCacheExample.xml</td><td>This template shows how to use ExecuteScript to populate and fetch from a DistributedMapCacheServer using a DistributedMapCacheClientService. Prior to NiFi 1.0.0 this had to be done manually using the map cache protocol, now the DistributedMapCacheClient can be used directly.</td><td>1.0.0</td><td>ExecuteScript</td></tr>
<tr><td>publish-to-confluent-kafka.xml</td><td>This template generates canned CSV data via GenerateFlowFile and publishes the records to Kafka in Avro format using PublishKafkaRecord_0_10 with the Confluent Schema Registry.

Instructions for using this template:

1. Create Avro Schema for Schema Registry and write to file product-schema.json

<pre>{
  "schema": "{
    \"name\": \"products\",
    \"namespace\": \"nifi\",
    \"type\": \"record\",
    \"fields\": [
        { \"name\": \"id\", \"type\": \"string\" },
        { \"name\": \"product_name\", \"type\": \"string\" },
        { \"name\": \"description\", \"type\": \"string\" }
    ]
  }"
}</pre>

2. Uplaod schema to Schema Registry:
curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" --data @product-schema.json http://localhost:8081/subjects/products/versions

>>> {"id":2}

3. Create dataflow using template (publish-to-confluent-kafka.xml)

4. Publish data via NiFi to Kafka topic by starting flow

5. Consume data from Kafka. Consumer can be created with following pom:

<pre><project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.test</groupId>
    <artifactId>kafka-stream-test</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>0.11.0.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.avro</groupId>
            <artifactId>avro</artifactId>
            <version>1.8.2</version>
        </dependency>
        <dependency>
            <groupId>io.confluent</groupId>
            <artifactId>kafka-avro-serializer</artifactId>
            <version>3.3.0</version>
        </dependency>
    </dependencies>
</project></pre></td><td></td><td></td></tr>
</table>

	And the following consumer:
	
	`package org.test;`
	
	`import java.util.Collections;`
	`import java.util.Properties;`
	`import java.util.UUID;`
	`import org.apache.kafka.clients.consumer.Consumer;`
	`import org.apache.kafka.clients.consumer.ConsumerRecords;`
	`import org.apache.kafka.clients.consumer.KafkaConsumer;`
	`import org.apache.kafka.common.serialization.StringDeserializer;`
	`import io.confluent.kafka.schemaregistry.client.`
	`CachedSchemaRegistryClient;`
	`import io.confluent.kafka.schemaregistry.client.SchemaRegistryClient;`
	`import io.confluent.kafka.serializers.KafkaAvroDeserializer;`
	
	`public class Consume {`
	
	`    public static void main(final String[] args) {`
	`        Properties props = new Properties();`
	`        props.put("bootstrap.servers", "localhost:9092");`
	
	`        final String registryUrl = "http://localhost:8081";`
	`        final SchemaRegistryClient schemaReg = new`
	`CachedSchemaRegistryClient(registryUrl, 1024);`
	`        final KafkaAvroDeserializer deser = new KafkaAvroDeserializer`
	`(schemaReg);`
	
	`        props.setProperty("group.id", UUID.randomUUID().toString());`
	
	`        Consumer<String, Object> consumer = new KafkaConsumer<>(props,`
	`new StringDeserializer(), deser);`
	`        consumer.subscribe(Collections.singleton("products"));`
	
	`        while (true) {`
	`            final ConsumerRecords<String, Object> records = consumer.poll`
	`(1000);`
	`            if (records == null) {`
	`                continue;`
	`            }`
	`            records.forEach(rec -> System.out.println(rec.key() + " : " +`
	`rec.value()));`
	`        }`
	`    }`
	
	`}`		

JoinCSVRecords.xml	This example illustrates how to use LookupRecord processor to join multiple columns from another CSV file. The key configuration is to define a result schema at 'RecordWriter' having the original columns AND the columns those are enriched by the lookuped values. Also, use 'Insert Record Fields' as LookupRecord 'Record Result Content', and '/' as 'Result Record Path' to join the lookuped values into the original record. Detailed explanation and sample data can be found in this Gist page.		LookupRecord

http://nifi.apache.org/docs/nifi-docs/html/user-guide.html#site-to-site