App Templates provides Zero App Experience
Use ETL Template to create ETL Pipelines

Lets start with how users will create ETL Pipelines with the proposed ETL Templates.

Scenario I: A user wants to create a Pipeline that reads from MySQL Table periodically, do simple Projection transform and write the data to a TableSink. He can use the ETL Template to create Pipelines without writing a single line of code.

Since the MySQLBatchSource, Projection Transform, TableSink are available right out of the box in CDAP, he can simply use the CLI to create and start his pipeline. There will be option to use a configuration file to create a pipeline too instead of a very verbose command line. Manifest is an instance of an App-Template.

> create manifest syncPipe template etl-template --properties={“schedule”:”1hr”, “type”:”batch”} source MySQLBatchSource --tableName=”abc” transforms Projection --dropField=”age” sink TableSink --tableName=”datasetinCDAP”
“Manifest created successfully”

> start syncPipe
“SyncPipe started”

The above command creates a Pipeline ‘syncPipe’ which uses template etl-template to create a batch pipeline with MySQLBatchSource as the source stage, Projection as the Transform stage and TableSink as the sink stage.

Scenario II: Lets say now user wants to do the same with an Oracle Database Table instead of MySQL table. And lets assume we don’t have that source available out of the box. Or it could be that the user wants to implement a complex Filter Transform that only allows records to pass through if the field age > 50 and state != “CA” etc. In both these cases, the user can use the etl-api to create their own custom etl pipeline stages (sources, transforms, sinks).

Once the user creates these using the ETL API interfaces, he can then put the JAR in the plugins directory (explained later) of the ETL Template. And when he restarts CDAP, he now can build his pipeline similar to the one above (and also start it in the same command).

> create pipeline oraclePipe type batch source OracleBatchSource --tableName=”hr” transforms Filter --age=”50” --ignoreState=”CA” sink TableSink --tableName=”hrTableinCDAP” schedule 30min --start

In the case of Scenario I, the user didn’t have to write a single line of code. All he had to do was pick the source, sink, transform(s) that are the stages of the pipeline and property the

properties to configure each of them. In case, any of source, transform, sink that he wants doesn't exist, then he can create his own custom stage by using the ETL API which is domain specific (ETL specific) and thus makes it easy for the user to write one. And once he implements his custom stage(s), he can copy the JARs over to the plugins directory and he can use these stages to create his pipeline.

Some of the use cases that we want to support are:
Realtime Pipelines:
Kafka -> Identity Transform -> Stream
JMS -> Identity -> Stream
Kafka -> Projection -> Stream
RDBMS -> T -> Dataset

Batch Pipelines:
Stream -> Transform -> HBase/Fileset
Dataset -> T (rules based) -> RDBMS
RDBMS -> T -> Dataset

As mentioned earlier, ETL Template is an App-Template.

What is an App-Template?

App template is a CDAP application that can provide higher level abstraction than CDAP API for common design patterns aka use cases. For example, our first App-Template will be an ETL Template. If users want to develop a simple ETL Pipeline with custom stages, they will need to work with the higher level API (which is more in sync with the ETL Pipeline design instead of raw CDAP API). For each instance of the App-Template (ie, in case of ETL Template, each Pipeline created from the ETL Template is an instance), we launch a run of the Program as described in the MANIFEST of the App-Template (more on how that is achieved is explained below). The basic idea is that only one CDAP Application is deployed for each App-Template and all instances created are just multiple runs of a Program of that CDAP Application.

How are App-Templates "deployed" in CDAP?

App-Templates are deployed in a different way compared to the regular CDAP Applications. The JARs of the App-Template (CDAP application + MANIFEST) is placed in the /templates directory of CDAP. When CDAP starts up, it scans this directory to get an account of the App-Templates that are present in the platform.

What does the Manifest file of the App-Template JAR contain:
JSON blob that contains the following info.

Name: ETLTemplate (unique)
InstanceName: Pipeline (unique)
Types: Realtime {ProgramName: ETLWorkerDriver ProgramType: Worker {properties: instances}}, Batch {ProgramName: ELTWorkflowDriver ProgramType: Workflow {properties: schedule}}
Plugins: {Yes {Dir: /etl}} (unique)

More info could be added to this MANIFEST as required during development. But the basic information is as follows:

Types - these are the instance types that could be created using this template. For example, in the case of ETL Template (above example), one could create an Realtime Pipeline, or a Batch Pipeline. If one is creating Realtime pipeline, the program to be started is ETLWorkerDriver which is a Worker. And also the property to look for during the configuration is “instances” - since this is a property that is used outside of the driver itself (that is to set the number of instances of the Pipeline (Worker)). Similarly for Batch pipeline, the program to be started is ETLWorkflowDriver and the property to be used while starting the program is “schedule”.

Plugins are an important part of the App-Templates. This contains different implementations of the API exposed as part of the App-Template. For example, these are JARs that contain the different sources, sinks, transforms implementations which could be used in creation of ETL Pipelines.

In case of ETL Template, we have a CDAP Application which acts as the Driver for the different Pipelines that are created by the user. Now the implementation of different sources, sinks, transforms (aka Pipeline stages) are part of the plugins for the ETL Template. The JARs can be placed in this directory before the start of CDAP (/templates/etl in the case of ETL Template) and these JARs will be made available to all the Programs when they are started (in the case of ETL, both the Worker and MapReduce).

REST APIs for App-Templates:

These are implemented in CDAP Platform.

List all App-Templates:
GET /v3/templates/

Creating an instance from App-Template:
POST /v3/templates/{template-id} -d ‘{ JSON blob which contains the configuration of the instance}’

For ex: In case of ETL template, PUT /v3/templates/etl/manifests/{manifest-id} -d ‘{“ JSON “}’

Starting a Manifest [please note I am using ‘manifest’ as a placeholder until we come up with a suitable name for app-templates and its instances]:

POST /v3/templates/{template-id}/manifests/{manifest-name}/start

For ex: /v3/templates/etl/manifests/KafkaToStream/start

Stop a Manifest:
POST /v3/templates/{template-id}/manifests/{manifest-name}/stop

Status of a Manifest:
GET /v3/templates/{template-id}/manifests/{manifest-name}/status

Configuration of a Manifest:
GET /v3/templates/{template-id}/manifests/{manifest-name}

Deleting a Manifest:
DELETE /v3/templates/{template-id}/manifests/{manifest-name}

Increasing the number of instances of a Manifest:
PUT /v3/templates/{template-id}/manifests/{manifest-name}/instances -d ‘{“instances”:”3”}’

List all Manifests of a App-Template:
GET /v3/templates/{template-id}/manifests

Plugin Info of an App-Template:
GET /v3/templates/{template-id}/info

This returns info about the plugins. In case of ETL Template, it returns the names of each stage along with the configuration parameters.

In order to provide the above lifecycle management for manifests of app-templates, the CDAP Platform maintains the following info (table):

Manifest Name	Template	Manifest Properties	Program Type	RunId
KafkaToStreamPipe	ETL Template	{“source”: ….}	Worker	XXXX

We have discussed about App-Templates. In the next section, we will now delve into ETL Template (which is the first template that we will shipping along with CDAP). Note that users can create their own App Templates if they need. But they can use ETL Template to create ETL Pipelines out of the box when the install CDAP.

ETL Template consists of -> Driver Application, ETL API (which allows users to create custom Sources, Sinks, Transforms (Stages) which may not be available out of the box - we will be shipping the common ones - KafkaSource, TwitterSource etc).

ETL Template enables users to build ETL pipelines out of the box on CDAP. Each instance/manifest of an ETL template is known as ETL Pipeline. Logically, ETL Pipeline is composed of the following stages:

- **Source (Realtime/Batch)**
- **Transform(s)**
- **Sink (Realtime/Batch)**

Source and Sink can be either Realtime or Batch. Transform stages are purely functional and stateless and thus can be used either in a Realtime or in a Batch pipeline.

The following are a sample of the ETL operations (Use Cases) that be could be performed.

Realtime Use Cases:

- KafkaSource, (No transform), Stream Sink
- JMS Message, Filter Transform, Stream Sink
- Database (stream new records), Projection Transform, TableSink

Batch Use Cases:

- StreamSource, Rules-based-Transform, HBaseSink or FileSetSink
- DatasetSource, Rules-based-Transform, HBaseSink or FileSetSink
- RDBMS Source, AvroRecord-to-Put Transform, TableSink
- KafkaSource, ByteBuffer-to-AvroRecord, ProjectionTransform, Avro-to-Put Transform, TableSink

Detailed Use Case:

Lets take one example from above and delve into it in detail.

MySQL Source -> Projection Transform -> AvroRecord-to-Put -> Table Sink

MySQL Source emits AvroRecord. Projection Transform accepts AvroRecord and emits AvroRecord. Projection Transform can drop fields or rename fields in a record. It will be configured through runtime arguments this will be a map (have to figure out finer details when implementing the exact Transform logic). It emits AvroRecord and this is wired to the AvroRecordToPutTransform which takes in AvroRecords and emits HBase Puts. And finally TableSink accepts Put objects which it receives from the Avro-to-Put transform.

For now, they the custom user sources/sinks/transform JARs (including the pre-built library of commonly used sources/transforms/sinks) will be placed in plugins directory to be scanned during the starting of CDAP (similar to current Adapters) but this will be enhanced to have the ability to deploy user's custom Source, Sink, Transforms through UI.

List of Transforms that we are targeting (some of these can be rolled into one if we have a generic way of configuring the transform):

- Projection Transform (Input : AvroRecord, Output : AvroRecord, Configuration : JSON Map which gives info about field renaming and dropping fields)
- Filter Transform (Input : AvroRecord, Output : AvroRecord, Configuration : JSON info of greater, lesser, equality filter info which is used to drop records that don't match the criteria)
- Lookup Transform (Input : AvroRecord, Output : AvroRecord, Configuration : JSON info which maps a specific field (Int)1->(String)Male (Int)2->(String)Female)
- Split Transform (Input : AvroRecord, Output : AvroRecord, Configuration : JSON that will enable splitting of one record to multiple ones - need more clarity)

Transforms that will NOT be possible with our ETL Template:

- Aggregation (transforms are meant to purely functional and and not hold any state)
- Joins (supporting multiple sources to one transform is not planned for this release)

Drivers:

There will be one ETLTemplateDriver CDAP Application that contains one Worker, which acts as a driver for Realtime pipelines, and one MapReduce Program, which acts as a driver for Batch pipelines. It might also have a wrapper Workflow to enable scheduling of the Batch pipelines.

Realtime Pipelines are executed in a Worker driver. Each pipeline gets its own Worker run (Twill Application) whose instances can be increased/decreased on the fly [this needs Worker instance change callback implementation]. Creation of a new pipeline will result in the spawning of another Worker run (a new Twill application).

Worker Driver:

```
for ( ; ; ) {
        list<data> <= poll(source)
        list<tdata> <= transform(list<data>)
        In Tx {
                write(sink) <= list<tdata>
                persist(sourceState)
        }
}
```

Batch Pipelines are executed in a MapReduce Program. Each pipeline is scheduled at the frequency specified in the Pipeline configuration.

In addition to driver logic, the Worker/MapReduce should also validate if the source, transform(s) and sinks can be wired together. This is done by checking if the generic types match (source out == transform in && transform out == transform in && …. && transform out == sink in).

Programmatic APIs:

In addition to providing Out of the Box ETL pipeline stages, we also want to provide simple, intuitive APIs to users to help them build their own custom sources, sinks and transforms. So if the user who wants to built an ETL Pipeline through CLI/REST API, does not find a stage that he wants, he can build his own using the ETL APIs as described below.

Realtime Source:

Things to note:

Sources implement a poll method which is invoked to ‘poll’ for data by the Worker driver. This data is then passed on to the subsequent transform (or sink, if there are no transform stages) stage. All the data transmission between the stages are accomplished through in-memory data structures.

Sources can limit the max number of instances of the Pipeline. This is conveyed to the driver through SourceConfigurer. During the increasing/decreasing of the Pipeline instances, the source might have to reconfigure itself (for ex, Kafka source might want to adjust the partitions that it is subscribing). onSuspend is called to pause the workers from fetching from the source and once the new instances have been created (or additional ones destroyed), the onResume method is invoked and the source can re-configure and start fetching data again). When we fetch data from the source using the poll method, we also supply it with its previously persisted state which can be used to decide what data to fetch. Optionally it can return back a

state which is then persisted when all the data that it emitted in the current poll cycle has been processed. This helps with failure recovery by preventing data loss. None of the source methods are executed within a transaction. Only the fetching and persistence of states are executed within a transaction (in order to mitigate long latencies that sources might face while fetching data).

Example of Realtime Sources that we are targeting in the order of priority:

- Kafka (emits ByteBuffer)
- Twitter (emits ?)
- JMS (emits String)
- CDAP Streams (emits StreamEventData)
- Amazon Simple Queue Service (emits ?)
- DB Sync (Appends) (emits AvroRecord)

```
/**
 * Realtime Sink.
 * @param <I> Object sink operates on
 */
public abstract class AbstractRealtimeSink<I> implements StageLifecycle {

  private SinkContext context;

  /**
   * Configure the Sink.
   * @param configurer {@link StageConfigurer}
   */
  public void configure(StageConfigurer configurer) {
    configurer.setName(this.getClass().getSimpleName());
  }

  /**
   * Initialize the Sink.
   * @param context {@link SinkContext}
   */
  public void initialize(SinkContext context) {
    this.context = context;
  }

  /**
   * Write the given object.
   * @param object object to be written
```

```
   */
  public abstract void write(I object);

  @Override
  public void destroy() {
    //no-op
  }

  protected SinkContext getContext() {
    return context;
  }
}
```

Realtime Sink:

The write method is invoked for each data object from the previous stage.

Examples of Realtime Sinks:

- Stream Sink (Accepts StreamEventData)
- KeyValueTableSink (?)
- TableSink (accepts Put)

```
/**
 * Realtime Sink.
 * @param <I> Object sink operates on
 */
public abstract class AbstractRealtimeSink<I> implements StageLifecycle {

  private SinkContext context;

  /**
   * Configure the Sink.
   * @param configurer {@link StageConfigurer}
   */
  public void configure(StageConfigurer configurer) {
    configurer.setName(this.getClass().getSimpleName());
  }

  /**
   * Initialize the Sink.
   * @param context {@link SinkContext}
```

```
   */
  public void initialize(SinkContext context) {
    this.context = context;
  }

  /**
   * Write the given object.
   * @param object object to be written
   */
  public abstract void write(I object);

  @Override
  public void destroy() {
    //no-op
  }

  protected SinkContext getContext() {
    return context;
  }
}
```

Transform API :

Transforms are purely functional and no state is saved. They operate on each record by record.

Examples of Transforms:

- Projection Transform (AvroRecord to AvroRecord)

```
/**
 * Transform Stage
 * @param <I> input
 * @param <O> output
 */
public abstract class AbstractTransform<I, O> implements StageLifecycle {

  private TransformContext context;

  /**
   * Configure the Transform stage.
   * @param configurer {@link StageConfigurer}
```

```
   */
  public void configure(StageConfigurer configurer) {
    configurer.setName(this.getClass().getSimpleName());
  }

  /**
   * Initialize the Transform Stage.
   * @param context {@link TransformContext}
   */
  public void initialize(TransformContext context) {
    this.context = context;
  }

  /**
   * Process I input and emit O output using {@link Emitter}
   * @param input input data
   * @param output {@link Emitter} emit output
   */
  public abstract void transform(I input, Emitter<O> output);

  @Override
  public void destroy() {
    //no-op
  }

  protected TransformContext getContext() {
    return context;
  }
}
```

Moving on to Batch Source:

Batch Pipes, as mentioned earlier, are executed in MapReduce programs (with only mappers). So, batch sources have the following requirements:

- They need to set the InputFormat class. This is set by StreamBatchReadble in case of Streams, MapReduceContext.setInput(Dataset s) for datasets. For RDBMS, it is set to DBInputFormat etc. Sources might also have to set some Hadoop configuration settings and all these are done during the preparation of the MR job.
- read method : which takes in the KEY, VALUE from the map function and emits a Object that it wants to propagate to the next stage (transform or sink).

- Finish any task at the end of MapReduce task in onFinish method

This gives rise to the following API for Batch Source:

```
/**
 * Batch Source forms the first stage of a Batch ETL Pipeline.
 * @param <KEY> Mapper Key class
 * @param <VALUE> Mapper Value class
 * @param <O> Object that BatchSource emits
 */
public abstract class AbstractBatchSource<KEY, VALUE, O> implements StageLifecycle {

  private MapReduce context;

  /**
   * Configure the Batch Source stage.
   * @param configurer {@link StageConfigurer}
   */
  public void configure(StageConfigurer configurer) {
    configurer.setName(this.getClass().getSimpleName());
  }

  /**
   * Setup MapReduce configuration related to the Batch Source.
   * @param context {@link MapReduceContext}
   */
  public abstract void prepareJob(MapReduce context);

  /**
   * Initialize the Batch Source.
   * @param context {@link MapReduceContext}
   */
  public void initialize(MapReduce context) {
    this.context = context;
  }

  /**
   * Process data.
   * @param key Key from Mapper
   * @param value Value from Mapper
   * @param data Emit data
   */
  public abstract void read(KEY key, VALUE value, Emitter<O> data);
```

```
  @Override
  public void destroy() {
    // no-op
  }

  /**
   * Operation to be performed at the end of the Batch job.
   * @param succeeded true if Batch operation succeeded, false otherwise
   * @param context {@link MapReduceContext}
   * @throws Exception
   */
  public abstract void onFinish(boolean succeeded, MapReduceContext context) throws
Exception;
}
```

Examples of Batch Sources:

- CDAP Streams - (Key=LongWritable, Value=StreamEventData, Emiting StreamEventData)
- CDAP Datasets - KeyValueTable (Key=byte[], Value=byte[], Emitting = ?)
- RDBMS - MySQL, Oracle etc. (Key=LongWritable, Value=POJO, Emitting=AvroRecord)
- Kafka (Key=EtlKey, Value=CamusWrapper, Emitting=ByteBuffer)

Batch Sink:

Similar to Batch Sources, sinks have similar requirements. Instead of read method, they have a write method that takes in the Object that needs to be written and also the Mapper context that will be used to write to the target destination as set by the OutputFormatClass.

Examples of Batch Sink:

- CDAP Datasets - TableSink, FileSet Sink (accpets Put)
- RDBMS

```
/**
 * Batch Sink forms the last stage of a Batch ETL Pipeline.
 * @param <I> Object sink operates on
 */
```

```
public abstract class AbstractBatchSink<I> implements StageLifecycle {

  private MapReduceContext context;

  /**
   * Configure the Sink.
   * @param configurer {@link StageConfigurer}
   */
  public void configure(StageConfigurer configurer) {
    configurer.setName(this.getClass().getSimpleName());
  }

  /**
   * Prepare the Batch Job
   * @param context {@link MapReduceContext}
   */
  public abstract void prepareJob(MapReduceContext context);

  /**
   * Initialize the Sink.
   * @param context {@link MapReduceContext}
   */
  public void initialize(MapReduceContext context) {
    this.context = context;
  }

  /**
   * Write the given object
   * @param context {@link Mapper.Context}
   * @param object object to be written
   * @throws IOException
   * @throws InterruptedException
   */
  public abstract void write(Mapper.Context context, I object) throws IOException,
InterruptedException;

  @Override
  public void destroy() {
    // no-op
  }

  /**
   * Invoked after Batch Job is completed
```

```
   * @param succeeded true if batch job completed successfully
   * @param context {@link MapReduceContext}
   */
  public abstract void onFinish(boolean succeeded, MapReduceContext context);

  protected MapReduceContext getContext() {
    return context;
  }
}
```

Configuring Stages and Creating an Pipeline:

Each Pipeline stage exposes a Set<String> reqdProperties which returns the set of properties that is required by that stage to function. The creation of pipeline endpoint (ie, manifest will expect a JSON body which contains the following properties - one source with properties, one sink with properties and zero or more transforms in a list:

**
Name :
Type :
Properties for Pipeline (for example, in case of Batch - the frequency schedule)
Source Type : Properties for the Source
[List of < Transform Type : Properties for the Transform >]
Sink Type : Properties for the Sink
**

The following REST endpoints and possible CLI design for the lifecycle management of manifests were described in the beginning of the document.

Organization of Modules:

cdap -> cdap-api
-> cdap-app-templates
-> cdap-etl
-> cdap-etl-api [API Jar]
-> cdap-etl-core [Generates App Jar and template services]
-> cdap-etl-lib [Generates Jar that contains sources, etc]
-> cdap-etl-test [API Jar]

When CDAP starts up, the app-templates are deployed (etl-template being one of them) and acts as the Driver. All the pre-built (cdap-etl-lib) sources, transforms, sinks are packaged into a single JAR and it is placed in the ‘plugins’ directory. This is the directory which will be

scanned during the startup to get all sources, transforms, sinks that can be used in the driver (and this directory could also contain custom user sources, transforms, sinks). The library JAR (generated from cdap-etl-lib) is also placed in the same directory.

Metrics:

Ability to query and fetch metrics using RunId in addition to the Program Id (since each pipeline is tied to a RunId of a Worker/MapReduce Program).

Logging:

Ability to fetch logs at RunId level (similar to Metrics)

Test Framework (cdap-etl-test): We will provide a test framework that makes it easy for users to test their sources, sinks and transforms.

CDAP Feature Request List:

- Worker instance change should in call backs - before and after change [this will allow us to invoke onSuspend, onResume methods of Realtime Source (alternative: fixed number of worker instances, configured in the state - can't be changed during runtime; on the plus side, simplifies API)]
- Ability to create Datasets in the program [in Batch/Realtime Sink, if the dataset does not exist, we should be able to create it (alternative: require datasets to exist ahead of time or create them during pipe creation stage)]
- Generic Stream Consumer - allowing ability to consume Streams in Worker [will allow us to have a Realtime Stream Source (alternative: timestamp based fetch or no stream realtime source)]
- Not requiring useDatasets in Workers/MapReduce [since we don't know in advance what are the datasets being used in the Worker/MR (alternative: add it to configurer) - this is a very simple change in Workers and it is already complete in MR]
- FormatSpecification in StreamBatchReadble should support StreamEventData as a Map's input value class (alternative: support string output)]

Summary of End to End view for Users wrt ETL Template:

Users can choose to either:

- Build ETL Pipelines out of the available Sources, Sinks, Transforms that comes as part of the ETL Template:

This will involve the user making a REST API call to the App-Templates endpoint requesting the CDAP platform to create a manifest of the ETL Template aka ETL Pipeline which consists of: <Pipeline Properties, [Source A {with properties}, Transform X {with properties}, Transform Y {with properties}, Sink B {with properties}]>. This will create an ETL Pipeline whose lifecycle (start, stop, increase/decrease instances) will be managed by the CDAP platform.

- Expand the set of sources, sinks, transforms that are available in the ETL Template:

  When users want to create their own sources, sinks, they can create a maven project and add dependency on cdap-etl-api. The cdap-etl-api JAR will contain Abstract Source/Sink/Transform classes from which users can extend and build their own. They can optionally also add dependency on cdap-etl-test which is a simple extension of the CDAP's TestBase and deploys the Driver App. It provides other simple helper TestClasses to test Transforms, Sinks etc. Once they have created and tested their sources, sinks, transforms, they can copy the JAR to the plugins directory (which contains user's source/transform/sink classes, cdap-etl-api jar and libraries used by users' classes).

  Currently the users' need to restart CDAP to redo the scan of the JARs, but we will find out if there is a way to mitigate this and do it while CDAP is running. Also the users need to specify what are the sources, sinks, transform classes that can be used from the User's JAR. We will figure out a way if this process can be made less cumbersome (ie automated).

UI support for ETL Template:

Users will be able to create ETL Pipelines and monitor their status, start, stop and increase/decrease instances. User Metrics and Logs could possibly be supported but System Metrics for ETL Pipelines might be addressed in subsequent release.

Things to be discussed:

- How to localize/make plugin JARs available for Programs to load pipeline stages?
- How to make plugin info of ETL template available to the user?
- How to validate a configuration before starting an instance of a template (if possible)
- How to validate an increase in number of instances of Worker instances (since Source stage might have a limit on number of instances)

Shortcomings:

- User needs to restart CDAP for new App-Templates and their plugins to take effect.

- Custom Dataset sinks are not possible.

****