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

ETL App Template enables users to build ETL pipelines out of the box on CDAP. 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, ByteBuffer-to-CSV String 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

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.

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) and this implies that the lifecycle of the pipelines will need to be maintained outside of the driver application and in custom App Template logic (basically needs to maintain a map of Pipeline-Name to RunId map) which will be used by CDAP.

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 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. The following are the proposed APIs for the same.

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)

```
public interface RealtimeSource<O> extends StageLifecycle {

 /**
  * Configure the Source.
  * @param configurer {@link SourceConfigurer}
  */
 void configure(SourceConfigurer configurer);

 /**
  * Initialize the Source.
  * @param context
  */
 void initialize(SourceContext context);

 /**
  * Invoked when source is suspended
  */
 void onSuspend();

 /**
  * Resume/reconfigure from the state of suspension.
```

```
   * @param oldInstance old instance count
   * @param newInstance new instance count
   */
  void onResume(int oldInstance, int newInstance);

  /**
   * Poll for new data.
   *
   * @param writer {@link Emitter}
   * @param currentState {@link SourceState} current state of the Source
   * @return {@link SourceState} state of the source after poll, will be persisted when all data
from poll are processed
   */
  @Nullable
  SourceState poll(Emitter<O> writer, @Nullable SourceState currentState);
}
```

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)

```
public interface RealtimeSink<I> extends StageLifecycle {

  /**
   * Configure the Sink.
   * @param configurer configurer
   */
  void configure(StageConfigurer configurer);

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

  /**
```

```
   * Write the given object.
   *
   * @param object Object to written
   */
  void write(I object);
}
```

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)

```
public interface Transform<I, O> extends StageLifecycle {

  void configure(StageConfigurer configurer);

  void initialize(TransformContext context);

  void transform(I input, Emitter<O> output);
}
```

Moving on to Batch Source:

Batch Adapters, 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:

```
public interface BatchSource<KEY, VALUE, O> extends StageLifecycle {
  void configure(StageConfigurer configurer);

  void prepareJob(MapReduceContext context);

  void initialize(MapReduceContext context);

  void read(KEY key, VALUE value, Emitter<O> data);

  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.

```
public interface BatchSink<I> extends StageLifecycle {

  void configure(StageConfigurer configurer);

  void prepareJob(MapReduceContext context);

  void initialize(MapReduceContext context);

  void write(Mapper.Context context, I input) throws IOException, InterruptedException;

  void onFinish(boolean succeeded, MapReduceContext context);
}
```

Examples of Batch Sink:

- CDAP Datasets - TableSink (accpets Put)

- RDBMS

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 adapter endpoint will expect a JSON body which contains the following properties:

**
Adapter Name
Properties for Adapter (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 will be provided:

- Creating an etl-app-template instance (aka etl pipeline)
- Deleting an etl pipeline
- Increase/Decrease/Get pipeline Instances
- Explicit Start/Stop?
- Available Stages - Sources, Sinks, Transforms

Similar commands will be provided in CLI. The tricky part is in creating a pipeline through CLI where the various configuration properties must be specified! One way to solve this is by allowing users to load a configuration file that contains all the properties required to construct an ETL pipeline.

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 Jar generated from cdap-etl-core is deployed 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. The library JAR (generated from cdap-etl-lib) is also placed in the same directory. More on this later in the docs.

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 adapter 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)]

End to End Use:

Users can choose to either:

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

  This will involve the user making a REST API call to the App-Templates endpoint requesting the CDAP platform to create an instance of the ETL App 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 App Template:

  When users want to create their own sources, sinks, they can create a maven project and add dependency on cdap-etl-api (this JAR will be published). 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 (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).

Contract provided by Etl-App-Template to Platform:

App Templates should provide the following non-trivial services to the CDAP platform. Currently this is focused specifically on etl-app-template but this can be generalized once we have more app-templates and common patterns.

- Provide a way to query what are the sources, sinks, transforms available to be used and the configuration parameters that each stage expects
- Ability to validate a pipeline configuration before starting a worker run
- Perform lifecycle management of Instances of the app-template by maintaining a table of currently configured, running pipelines
- Way to provide info or deploy etl-driver-app during startup

****