CDAP Programming Guide

Introduction to Programming Applications for the Cask Data Application Platform (CDAP)

Contents

Introduction **3**

Conventions 3

Writing a CDAP Application **4**

Using the CDAP Maven Archetype 4

Programming APIs: Applications **5**

Collecting Data: Streams **6**

Processing Data: Flows **7**

Processing Data: Flowlets **8**

Input Context 9

Type Projection 9

Stream Event 10

Flowlet Method and @Tick Annotation 11

Connection 11

Processing Data: MapReduce **12**

MapReduce and Datasets 14

Processing Data: Workflows **15**

Store Data: Datasets **16**

Query Data: Procedures **17**

Where to Go Next **19**

Introduction

This document covers in detail the Cask Data Application Platform (CDAP) core elements—Applications, Streams, Datasets, Flows, Procedures, MapReduce, and Workflows—and how you work with them in Java to build a Big Data application.

For a high-level view of the concepts of the Cask Data Application Platform, please see the Introduction to the Cask Data Application Platform.

For more information beyond this document, see the Javadocs and the code in the examples directory, both of which are on the Cask.co Developers website as well as in your CDAP installation directory.

Conventions

In this document, *Application* refers to a user Application that has been deployed into CDAP.

Text that are variables that you are to replace is indicated by a series of angle brackets (< >). For example:

```
PUT /v2/streams/<new-stream-id>
```

indicates that the text `<new-stream-id>` is a variable and that you are to replace it with your value, perhaps in this case *mystream*:

```
PUT /v2/streams/mystream
```

Writing a CDAP Application

Note that the CDAP API is written in a "fluent" interface style, and often relies on `Builder` methods for creating many parts of the Application.

In writing a CDAP Application, it's best to use an integrated development environment that understands the application interface to provide code-completion in writing interface methods.

Using the CDAP Maven Archetype

To help you get started, Cask has created a Maven archetype to generate a skeleton for your Java project.

Maven is the very popular Java build and dependencies management tool for creating and managing a Java application projects.

If you are running in an environment whose network access is mediated by a proxy server, look at the Maven guide to configuring a proxy for instructions on how to modify your `settings.xml` file (usually `${user.home}/.m2/settings.xml`) so that dependencies can be downloaded and resolved correctly.

This Maven archetype generates a CDAP application Java project with the proper dependencies and sample code as a base to start writing your own Big Data application. To generate a new project, execute the following command:

```
$ mvn archetype:generate \
  -DarchetypeCatalog=https://repository.cask.co/content/groups/releases/archetype-catalog.xml \
  -DarchetypeGroupId=co.cask.cdap \
  -DarchetypeArtifactId=cdap-app-archetype \
  -DarchetypeVersion=2.4.0
```

In the interactive shell that appears, specify basic properties for the new project. For example, to create a new project called *MyFirstBigDataApp*, setting appropriate properties, such as your domain and a version identifier:

```
Define value for property 'groupId': : com.example
Define value for property 'artifactId': : MyFirstBigDataApp
Define value for property 'version': 1.0-SNAPSHOT: :
Define value for property 'package': com.example: :
Confirm properties configuration:
groupId: com.example
artifactId: MyFirstBigDataApp
version: 1.0-SNAPSHOT
package: com.example
Y: : Y
```

After you confirm the settings, the directory `MyFirstBigDataApp` is created under the current directory. To build the project:

```
$ cd MyFirstBigDataApp
$ mvn clean package
```

This creates `MyFirstBigDataApp-1.0-SNAPSHOT.jar` in the target directory. This JAR file is a skeleton CDAP application that is ready to be edited with the contents of your Application. When finished and compiled, deploy it to the Cask DAP by just dragging and dropping it anywhere on the CDAP Console and it will be deployed.

The remainder of this document covers what to put in that JAR file.

Programming APIs: Applications

An **Application** is a collection of Streams, Datasets, Flows, Procedures, MapReduce jobs, and Workflows.

To create an Application, implement the `Application` interface or subclass from `AbstractApplication` class, specifying the Application metadata and declaring and configuring each of the Application elements:

```
public class MyApp extends AbstractApplication {
  @Override
  public void configure() {
    setName("myApp");
    setDescription("My Sample Application");
    addStream(new Stream("myAppStream"));
    addFlow(new MyAppFlow());
    addProcedure(new MyAppQuery());
    addMapReduce(new MyMapReduceJob());
    addWorkflow(new MyAppWorkflow());
  }
}
```

Notice that *Streams* are defined using provided `Stream` class, and are referenced by names, while other components are defined using user-written classes that implement correspondent interfaces and are referenced by passing an object, in addition to being assigned a unique name.

Names used for *Streams* and *Datasets* need to be unique across the DAP instance, while names used for *Flows*, *Flowlets* and *Procedures* need to be unique only to the application.

Collecting Data: Streams

Streams are the primary means for bringing data from external systems into the CDAP in realtime. You specify a Stream in your Application metadata:

```
addStream(new Stream("myStream"));
```

specifies a new Stream named *myStream*. Names used for Streams need to be unique across the DAP instance.

You can write to Streams either one operation at a time or in batches, using either the Cask Data Application Platform HTTP RESTful API or command line tools.

Each individual signal sent to a Stream is stored as an `StreamEvent`, which is comprised of a header (a map of strings for metadata) and a body (a blob of arbitrary binary data).

Streams are uniquely identified by an ID string (a "name") and are explicitly created before being used. They can be created programmatically within your application, through the CDAP Console, or by or using a command line tool. Data written to a Stream can be consumed by Flows and processed in real-time. Streams are shared between applications, so they require a unique name.

Processing Data: Flows

Flows are developer-implemented, real-time Stream processors. They are comprised of one or more Flowlets that are wired together into a directed acyclic graph or DAG.

Flowlets pass DataObjects between one another. Each Flowlet is able to perform custom logic and execute data operations for each individual data object processed. All data operations happen in a consistent and durable way.

When processing a single input object, all operations, including the removal of the object from the input, and emission of data to the outputs, are executed in a transaction. This provides us with Atomicity, Consistency, Isolation, and Durability (ACID) properties, and helps assure a unique and core property of the Flow system: it guarantees atomic and "exactly-once" processing of each input object by each Flowlet in the DAG.

Flows are deployed to the DAP instance and hosted within containers. Each Flowlet instance runs in its own container. Each Flowlet in the DAG can have multiple concurrent instances, each consuming a partition of the Flowlet's inputs.

To put data into your Flow, you can either connect the input of the Flow to a Stream, or you can implement a Flowlet to generate or pull the data from an external source.

The `Flow` interface allows you to specify the Flow's metadata, Flowlets, Flowlet connections, Stream to Flowlet connections, and any Datasets used in the Flow.

To create a Flow, implement `Flow` via a `configure` method that returns a `FlowSpecification` using `FlowSpecification.Builder()`:

```
class MyExampleFlow implements Flow {
  @Override
  public FlowSpecification configure() {
    return FlowSpecification.Builder.with()
      .setName("mySampleFlow")
      .setDescription("Flow for showing examples")
      .withFlowlets()
        .add("flowlet1", new MyExampleFlowlet())
        .add("flowlet2", new MyExampleFlowlet2())
      .connect()
        .fromStream("myStream").to("flowlet1")
        .from("flowlet1").to("flowlet2")
      .build();
}
```

In this example, the *name*, *description*, *with* (or *without*) Flowlets, and *connections* are specified before building the Flow.

Processing Data: Flowlets

Flowlets, the basic building blocks of a Flow, represent each individual processing node within a Flow. Flowlets consume data objects from their inputs and execute custom logic on each data object, allowing you to perform data operations as well as emit data objects to the Flowlet's outputs. Flowlets specify an `initialize()` method, which is executed at the startup of each instance of a Flowlet before it receives any data.

The example below shows a Flowlet that reads *Double* values, rounds them, and emits the results. It has a simple configuration method and doesn't do anything for initialization or destruction:

```
class RoundingFlowlet implements Flowlet {

  @Override
  public FlowletSpecification configure() {
    return FlowletSpecification.Builder.with().
      setName("round").
      setDescription("A rounding Flowlet").
      build();
  }

  @Override
    public void initialize(FlowletContext context) throws Exception {
  }

  @Override
  public void destroy() {
  }

  OutputEmitter<Long> output;
  @ProcessInput
  public void round(Double number) {
    output.emit(Math.round(number));
  }
```

The most interesting method of this Flowlet is `round()`, the method that does the actual processing. It uses an output emitter to send data to its output. This is the only way that a Flowlet can emit output to another connected Flowlet:

```
OutputEmitter<Long> output;
@ProcessInput
public void round(Double number) {
  output.emit(Math.round(number));
}
```

Note that the Flowlet declares the output emitter but does not initialize it. The Flow system initializes and injects its implementation at runtime.

The method is annotated with @``ProcessInput``—this tells the Flow system that this method can process input data.

You can overload the process method of a Flowlet by adding multiple methods with different input types. When an input object comes in, the Flowlet will call the method that matches the object's type:

```
OutputEmitter<Long> output;

@ProcessInput
public void round(Double number) {
  output.emit(Math.round(number));
}
```

```
@ProcessInput
public void round(Float number) {
  output.emit((long)Math.round(number));
}
```

If you define multiple process methods, a method will be selected based on the input object's origin; that is, the name of a Stream or the name of an output of a Flowlet.

A Flowlet that emits data can specify this name using an annotation on the output emitter. In the absence of this annotation, the name of the output defaults to "out":

```
@Output("code")
OutputEmitter<String> out;
```

Data objects emitted through this output can then be directed to a process method of a receiving Flowlet by annotating the method with the origin name:

```
@ProcessInput("code")
public void tokenizeCode(String text) {
  ... // perform fancy code tokenization
}
```

Input Context

A process method can have an additional parameter, the `InputContext`. The input context provides information about the input object, such as its origin and the number of times the object has been retried. For example, this Flowlet tokenizes text in a smart way and uses the input context to decide which tokenizer to use:

```
@ProcessInput
public void tokenize(String text, InputContext context) throws Exception {
  Tokenizer tokenizer;
  // If this failed before, fall back to simple white space
  if (context.getRetryCount() > 0) {
    tokenizer = new WhiteSpaceTokenizer();
  }
  // Is this code? If its origin is named "code", then assume yes
  else if ("code".equals(context.getOrigin())) {
    tokenizer = new CodeTokenizer();
  }
  else {
    // Use the smarter tokenizer
    tokenizer = new NaturalLanguageTokenizer();
  }
  for (String token : tokenizer.tokenize(text)) {
    output.emit(token);
  }
}
```

Type Projection

Flowlets perform an implicit projection on the input objects if they do not match exactly what the process method accepts as arguments. This allows you to write a single process method that can accept multiple **compatible** types. For example, if you have a process method:

```
@ProcessInput
count(String word) {
  ...
}
```

and you send data of type `Long` to this Flowlet, then that type does not exactly match what the process method expects. You could now write another process method for `Long` numbers:

```
@ProcessInput count(Long number) {
  count(number.toString());
}
```

and you could do that for every type that you might possibly want to count, but that would be rather tedious. Type projection does this for you automatically. If no process method is found that matches the type of an object exactly, it picks a method that is compatible with the object.

In this case, because Long can be converted into a String, it is compatible with the original process method. Other compatible conversions are:

- Every primitive type that can be converted to a `String` is compatible with `String`.
- Any numeric type is compatible with numeric types that can represent it. For example, `int` is compatible with `long`, `float` and `double`, and `long` is compatible with `float` and `double`, but `long` is not compatible with `int` because `int` cannot represent every `long` value.
- A byte array is compatible with a `ByteBuffer` and vice versa.
- A collection of type A is compatible with a collection of type B, if type A is compatible with type B. Here, a collection can be an array or any Java `Collection`. Hence, a `List<Integer>` is compatible with a `String[]` array.
- Two maps are compatible if their underlying types are compatible. For example, a `TreeMap<Integer, Boolean>` is compatible with a `HashMap<String, String>`.
- Other Java objects can be compatible if their fields are compatible. For example, in the following class `Point` is compatible with `Coordinate`, because all common fields between the two classes are compatible. When projecting from `Point` to `Coordinate`, the color field is dropped, whereas the projection from `Coordinate` to `Point` will leave the `color` field as `null`:

  ```
  class Point {
    private int x;
    private int y;
    private String color;
  }

  class Coordinates {
    int x;
    int y;
  }
  ```

Type projections help you keep your code generic and reusable. They also interact well with inheritance. If a Flowlet can process a specific object class, then it can also process any subclass of that class.

Stream Event

A Stream event is a special type of object that comes in via Streams. It consists of a set of headers represented by a map from String to String, and a byte array as the body of the event. To consume a Stream with a Flow, define a Flowlet that processes data of type `StreamEvent`:

```
class StreamReader extends AbstractFlowlet {
  ...
  @ProcessInput
  public void processEvent(StreamEvent event) {
    ...
  }
```

Flowlet Method and @Tick Annotation

A Flowlet's method can be annotated with `@Tick`. Instead of processing data objects from a Flowlet input, this method is invoked periodically, without arguments. This can be used, for example, to generate data, or pull data from an external data source periodically on a fixed cadence.

In this code snippet from the *CountRandom* example, the `@Tick` method in the Flowlet emits random numbers:

```
public class RandomSource extends AbstractFlowlet {

  private OutputEmitter<Integer> randomOutput;

  private final Random random = new Random();

  @Tick(delay = 1L, unit = TimeUnit.MILLISECONDS)
  public void generate() throws InterruptedException {
    randomOutput.emit(random.nextInt(10000));
  }
}
```

Connection

There are multiple ways to connect the Flowlets of a Flow. The most common form is to use the Flowlet name. Because the name of each Flowlet defaults to its class name, when building the Flow specification you can simply write:

```
.withFlowlets()
  .add(new RandomGenerator())
  .add(new RoundingFlowlet())
.connect()
  .fromStream("RandomGenerator").to("RoundingFlowlet")
```

If you have multiple Flowlets of the same class, you can give them explicit names:

```
.withFlowlets()
  .add("random", new RandomGenerator())
  .add("generator", new RandomGenerator())
  .add("rounding", new RoundingFlowlet())
.connect()
  .from("random").to("rounding")
```

Processing Data: MapReduce

MapReduce is used to process data in batch. MapReduce jobs can be written as in a conventional Hadoop system. Additionally, CDAP **Datasets** can be accessed from MapReduce jobs as both input and output.

To process data using MapReduce, specify `withMapReduce()` in your Application specification:

```
    public void configure() {
      ...
addMapReduce(new WordCountJob());
```

You must implement the `MapReduce` interface, which requires the implementation of three methods:

- `configure()`
- `beforeSubmit()`
- `onFinish()`

```
public class WordCountJob implements MapReduce {
  @Override
  public MapReduceSpecification configure() {
    return MapReduceSpecification.Builder.with()
      .setName("WordCountJob")
      .setDescription("Calculates word frequency")
      .useInputDataSet("messages")
      .useOutputDataSet("wordFrequency")
      .build();
  }
```

The configure method is similar to the one found in Flow and Application. It defines the name and description of the MapReduce job. You can also specify Datasets to be used as input or output for the job.

The `beforeSubmit()` method is invoked at runtime, before the MapReduce job is executed. Through a passed instance of the `MapReduceContext` you have access to the actual Hadoop job configuration, as though you were running the MapReduce job directly on Hadoop. For example, you can specify the Mapper and Reducer classes as well as the intermediate data format:

```
@Override
public void beforeSubmit(MapReduceContext context) throws Exception {
  Job job = context.getHadoopJob();
  job.setMapperClass(TokenizerMapper.class);
  job.setReducerClass(IntSumReducer.class);
  job.setMapOutputKeyClass(Text.class);
  job.setMapOutputValueClass(IntWritable.class);
}
```

The `onFinish()` method is invoked after the MapReduce job has finished. You could perform cleanup or send a notification of job completion, if that was required. Because many MapReduce jobs do not need this method, the `AbstractMapReduce` class provides a default implementation that does nothing:

```
@Override
public void onFinish(boolean succeeded, MapReduceContext context) {
  // do nothing
}
```

CDAP `Mapper` and `Reducer` implement the standard Hadoop APIs:

```
public static class TokenizerMapper
    extends Mapper<byte[], byte[], Text, IntWritable> {

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  public void map(byte[] key, byte[] value, Context context)
      throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(Bytes.toString(value));
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}

public static class IntSumReducer
    extends Reducer<Text, IntWritable, byte[], byte[]> {

  public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    context.write(key.copyBytes(), Bytes.toBytes(sum));
  }
}
```

MapReduce and Datasets

Both CDAP `Mapper` and `Reducer` can directly read from a Dataset or write to a Dataset similar to the way a Flowlet or Procedure can.

To access a Dataset directly in Mapper or Reducer, you need (1) a declaration and (2) an injection:

1. Declare the Dataset in the MapReduce job's configure() method. For example, to have access to a Dataset named *catalog*:

```
public class MyMapReduceJob implements MapReduce {
  @Override
  public MapReduceSpecification configure() {
    return MapReduceSpecification.Builder.with()
      ...
      .useDataSet("catalog")
      ...
```

2. Inject the Dataset into the mapper or reducer that uses it:

```
public static class CatalogJoinMapper extends Mapper<byte[], Purchase, ...> {
  @UseDataSet("catalog")
  private ProductCatalog catalog;

  @Override
  public void map(byte[] key, Purchase purchase, Context context)
      throws IOException, InterruptedException {
    // join with catalog by product ID
    Product product = catalog.read(purchase.getProductId());
    ...
  }
```

Processing Data: Workflows

Workflows are used to execute a series of MapReduce jobs. A Workflow is given a sequence of jobs that follow each other, with an optional schedule to run the Workflow periodically. On successful execution of a job, the control is transferred to the next job in sequence until the last job in the sequence is executed. On failure, the execution is stopped at the failed job and no subsequent jobs in the sequence are executed.

To process one or more MapReduce jobs in sequence, specify `withWorkflows()` in your application:

```
    public void configure() {
...
addWorkflow(new PurchaseHistoryWorkflow());
```

You'll then implement the `Workflow` interface, which requires the `configure()` method. From within `configure`, call the `addSchedule()` method to run a WorkFlow job periodically:

```
public static class PurchaseHistoryWorkflow implements Workflow {

  @Override
  public WorkflowSpecification configure() {
    return WorkflowSpecification.Builder.with()
      .setName("PurchaseHistoryWorkflow")
      .setDescription("PurchaseHistoryWorkflow description")
      .startWith(new PurchaseHistoryBuilder())
      .last(new PurchaseTrendBuilder())
      .addSchedule(new DefaultSchedule("FiveMinuteSchedule", "Run every 5 minutes",
                   "0/5 * * * *", Schedule.Action.START))
      .build();
  }
}
```

If there is only one MapReduce job to be run as a part of a WorkFlow, use the `onlyWith()` method after `setDescription()` when building the Workflow:

```
public static class PurchaseHistoryWorkflow implements Workflow {

  @Override
  public WorkflowSpecification configure() {
    return WorkflowSpecification.Builder.with() .setName("PurchaseHistoryWorkflow")
      .setDescription("PurchaseHistoryWorkflow description")
      .onlyWith(new PurchaseHistoryBuilder())
      .addSchedule(new DefaultSchedule("FiveMinuteSchedule", "Run every 5 minutes",
                   "0/5 * * * *", Schedule.Action.START))
      .build();
  }
}
```

Store Data: Datasets

Datasets store and retrieve data. Datasets are your means of reading from and writing data to the CDAP's storage capabilities. Instead of requiring you to manipulate data with low-level APIs, Datasets provide higher-level abstractions and generic, reusable Java implementations of common data patterns.

The core Dataset of the CDAP is a Table. Unlike relational database systems, these tables are not organized into rows with a fixed schema. They are optimized for efficient storage of semi-structured data, data with unknown or variable schema, or sparse data.

Other Datasets are built on top of Tables. A Dataset can implement specific semantics around a Table, such as a key/value Table or a counter Table. A Dataset can also combine multiple Datasets to create a complex data pattern. For example, an indexed Table can be implemented by using one Table for the data to index and a second Table for the index itself.

You can implement your own data patterns as custom Datasets on top of Tables. A number of useful Datasets—we refer to them as system Datasets—are included with CDAP, including key/value tables, indexed tables and time series.

You can create a Dataset in CDAP using either Cask Data Application Platform HTTP RESTful API or command line tools.

You can also specify to create a Dataset by Application components if one doesn't exist. For that you must declare its details in the Application specification. For example, to create a DataSet named *myCounters* of type *KeyValueTable*, write:

```
public void configure() {
    createDataset("myCounters", "KeyValueTable");
    ...
```

To use the Dataset in a Flowlet or a Procedure, instruct the runtime system to inject an instance of the Dataset with the `@UseDataSet` annotation:

```
class MyFowlet extends AbstractFlowlet {
  @UseDataSet("myCounters")
  private KeyValueTable counters;
  ...
  void process(String key) {
    counters.increment(key.getBytes());
  }
```

The runtime system reads the Dataset specification for the key/value table *myCounters* from the metadata store and injects a functional instance of the Dataset class into the Application.

You can also implement custom Datasets by implementing the `Dataset` interface or by extending existing Dataset types. See the PageViewAnalytics example for an implementation of a Custom Dataset. For more details, refer to Advanced Cask Data Application Platform Features.

Query Data: Procedures

To query CDAP and its Datasets and retrieve results, you use Procedures.

Procedures allow you to make synchronous calls into CDAP from an external system and perform server-side processing on-demand, similar to a stored procedure in a traditional database.

Procedures are typically used to post-process data at query time. This post-processing can include filtering, aggregating, or joins over multiple Datasets—in fact, a Procedure can perform all the same operations as a Flowlet with the same consistency and durability guarantees. They are deployed into the same pool of application containers as Flows, and you can run multiple instances to increase the throughput of requests.

A Procedure implements and exposes a very simple API: a method name (String) and arguments (map of Strings). This implementation is then bound to a REST endpoint and can be called from any external system.

To create a Procedure you implement the `Procedure` interface, or more conveniently, extend the `AbstractProcedure` class.

A Procedure is configured and initialized similarly to a Flowlet, but instead of a process method you'll define a handler method. Upon external call, the handler method receives the request and sends a response.

The initialize method is called when the Procedure handler is created. It is not created until the first request is received for it.

The most generic way to send a response is to obtain a `Writer` and stream out the response as bytes. Make sure to close the `Writer` when you are done:

```
import static co.cask.cdap.api.procedure.ProcedureResponse.Code.SUCCESS;
...
class HelloWorld extends AbstractProcedure {

  @Handle("hello")
  public void wave(ProcedureRequest request,
                   ProcedureResponder responder) throws IOException {
    String hello = "Hello " + request.getArgument("who");
    ProcedureResponse.Writer writer =
      responder.stream(new ProcedureResponse(SUCCESS));
    writer.write(ByteBuffer.wrap(hello.getBytes())).close();
  }
}
```

This uses the most generic way to create the response, which allows you to send arbitrary byte content as the response body. In many cases, you will actually respond with JSON. A CDAP `ProcedureResponder` has convenience methods for returning JSON maps:

```
// Return a JSON map
Map<String, Object> results = new TreeMap<String, Object>();
results.put("totalWords", totalWords);
results.put("uniqueWords", uniqueWords);
results.put("averageLength", averageLength);
responder.sendJson(results);
```

There is also a convenience method to respond with an error message:

```
@Handle("getCount")
public void getCount(ProcedureRequest request, ProcedureResponder responder)
                     throws IOException, InterruptedException {
```

```
  String word = request.getArgument("word");
  if (word == null) {
    responder.error(Code.CLIENT_ERROR,
                    "Method 'getCount' requires argument 'word'");
    return;
  }
```

Where to Go Next

Now that you've had an introduction to programming applications for CDAP, take a look at:

- Advanced Cask Data Application Platform Features, with details of the Custom Services, Flow, Dataset, and Transaction systems, and best practices for developing applications.