Continuuity Reactor Developer Guide

Version 1.6.1 Beta

INTRODUCTION

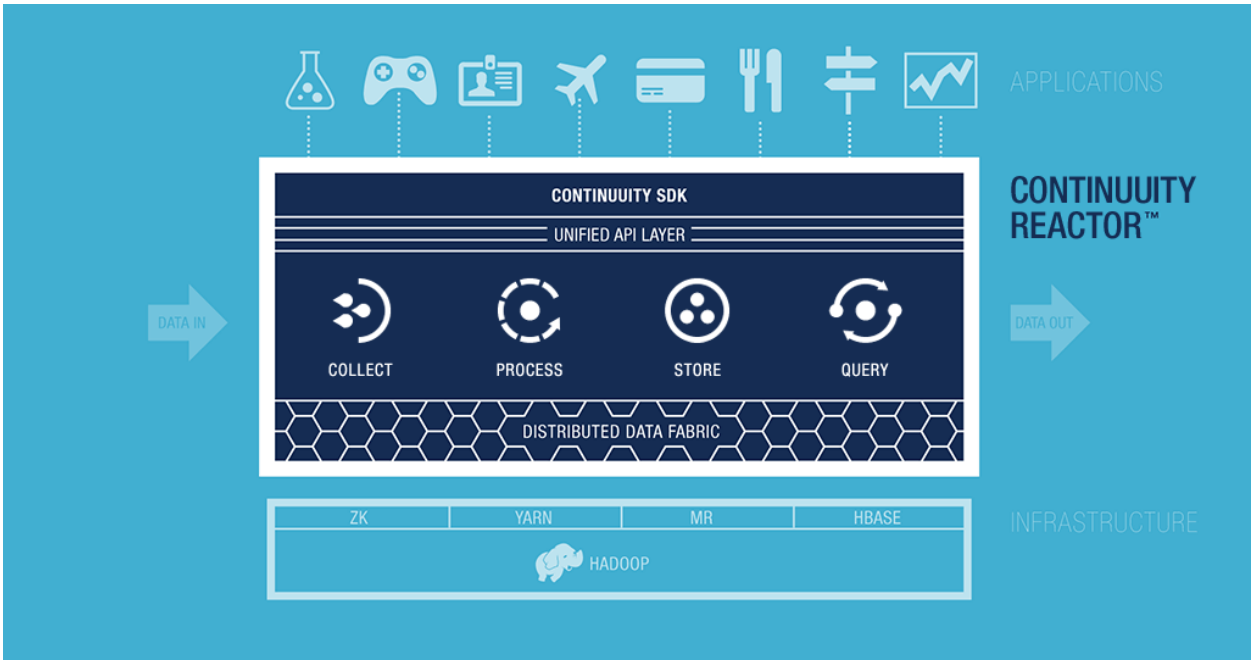
Developing, testing, running and scaling Big Data applications can be a difficult and complicated process requiring specialized expertise and large teams of engineers and operators. It is also a massive undertaking to educate yourself about the different projects that support Big Data applications within the Hadoop ecosystem, understand how they fit together, and effectively decide which technologies best suit your specific use cases.

Continuuity empowers developers by abstracting away unnecessary complexity and exposing the power of Big Data and Hadoop through higher-level abstractions, simple REST interfaces, powerful developer tools, and the Continuuity Reactor™, a scalable and integrated runtime environment and data platform with a rich visual user interface. Using Continuuity you can easily and quickly build, run, and scale Big Data applications from prototype to production.

This guide is intended for developers and explains the major concepts and key capabilities supported by the Continuuity Reactor, including an overview of the core APIs, libraries, and the Reactor Dashboard. The Getting Started Guide will have you running your own instance of the Reactor and deploying a sample Reactor application in minutes. To get you developing your own applications, the programming guide will deep-dive into the Core APIs and walk you through the implementation of an entire application, giving you an understanding of how Continuuity Reactor's capabilities enable you to quickly and easily build your own custom applications.

1.1. WHAT IS THE CONTINUUITY REACTOR™?

The Continuuity Reactor is a Java-based, integrated data and application framework that layers on top of Apache Hadoop, Apache HBase, and other Hadoop ecosystem components. It surfaces the capabilities of the underlying infrastructure through simple Java and REST APIs and shields you from unnecessary complexity. Rather than piecing together different open source frameworks and runtimes to assemble your own Big Data infrastructure stack, Continuuity provides an integrated platform – the Reactor – that makes it easy to compose the different elements of your Big Data application: collecting, processing, storing, and querying data.



The production Continuuity Reactors are available as a Hosted Reactor in the Continuuity cloud or as an Enterprise Reactor running behind your firewall. For development, you'll typically run your app on the local version of the Reactor on your own machine, which makes testing and debugging easy, and then you'll push your app to a free Sandbox Reactor in the Continuuity cloud to experience "Push-to-Cloud" functionality. Regardless of what version you use, your application code and your interactions with the Reactor remain the same.

1.2. What is the Continuuity Reactor Development Kit?

The Continuuity Reactor Development Kit gives you everything you need to develop, test, debug and run your own Big Data applications: a complete set of APIs, libraries, and documentation, an IDE plugin, sample applications, and the local version of the Reactor. The Reactor Development Kit is your on-ramp to the Continuuity Enterprise Reactor, enabling you to develop locally and then push to an Enterprise Reactor with a single click. Your interactions with the Enterprise Reactor are the same as with the Local Reactor, but you can now control the scale of your application to meet its demands, in real-time, with no application downtime.

What's in the Box?

The Continuuity Reactor Development Kit includes the Reactor Development Kit and the local version of the Continuuity Reactor.

Reactor Development Kit

The Reactor Development Kit includes the Continuuity Reactor Getting Started Guide, this Continuuity Reactor Developer Guide, all of the Continuuity APIs and libraries, Javadocs, command-line tools, Eclipse IDE plugin, and sample applications. See the Getting Started Guide for more details.

Local Reactor

The local variant of the Reactor is a fully functional but scaled-down runtime environment that emulates the typically distributed and large-scale Hadoop and HBase infrastructure in a lightweight way on your local machine. You run the Local Reactor on your own development machine, deploy your applications to it, and use the local dashboard to control and monitor it. You have direct access to your running application, making it easy to attach a debugger or profiler.

1.3. How to build Applications using Continuuity

You build the core of your application in your own IDE using the Continuuity Core Java APIs and libraries included in the Reactor SDK. We help to get you started with an Eclipse plugin and sample projects, as well as a set of example applications that utilize the various features of the Reactor. See the Getting Started Guide for more information.

Once the first version of your application has been built, you can deploy it to your Local Reactor using the Local Reactor Dashboard or the Eclipse Plugin. Then you can begin the process of testing, debugging, and iterating on your application.

Getting data in and out of your application can be done programmatically using REST APIs or via the dashboard and the command line tools.

Deploy your tested app to the Sandbox Reactor to test it in a cloud environment.

When it's ready for production you can easily deploy your app from your local machine to your Hosted Reactor or your Enterprise Reactor with no need for code changes or manual configuration. The production environment is highly available and can scale to meet the dynamic demands of your application.

2. HELLO WORLD!

Before going into the details of what the Continuuity Reactor is and how it works, here is a simple code example for the curious developer, a "Hello World!" application. It produces friendly greetings, using one stream, one dataset, one flow and one procedure. The next section introduces these concepts more thoroughly, and the Reactor Programming Guide on page 13 explains all of the APIs used here. The application: receives names as real-time events on a stream, processes the stream with a flow that stores each name in a key/value table, and on request, reads the latest name from the key/value table and returns "Hello <name>!"

```
public class HelloWorld implements Application {
  @Override
  public ApplicationSpecification configure() {
    return ApplicationSpecification.Builder.with().
      setName("HelloWorld").
      setDescription("A Hello World program for the App Fabric").
      withStreams().add(new Stream("who")).
      withDataSets().add(new KeyValueTable("whom")).
      withFlows().add(new WhoFlow()).
      withProcedures().add(new Greeting()).
      build();
  }
  public static class WhoFlow implements Flow {
    @Override
    public FlowSpecification configure() {
      return FlowSpecification.Builder.with().
        setName("WhoFlow").
        setDescription("A flow that collects names").
        withFlowlets().add("saver", new NameSaver()).
        connect().fromStream("who").to("saver").
        build();
    }
  }
  public static class NameSaver extends AbstractFlowlet {
    static final byte[] NAME = { 'n', 'a', 'm', 'e' };
    @UseDataSet("whom") KeyValueTable whom;
    public void processInput(StreamEvent event) throws OperationException {
      byte[] name = Bytes.toBytes(event.getBody());
      if (name != null && name.length > 0) {
        whom.write(NAME, name);
      }
    }
  }
  public static class Greeting extends AbstractProcedure {
    @UseDataSet("whom") KeyValueTable whom;
    @Handle("greet")
    public void greet(ProcedureRequest req, ProcedureResponder responder) throws
Exception {
      byte[] name = whom.read(NameSaver.NAME);
      String toGreet = name != null ? new String(name) : "World";
      responder.sendJson(new ProcedureResponse(SUCCESS), "Hello " + toGreet + "!");
    }
  }
}
```

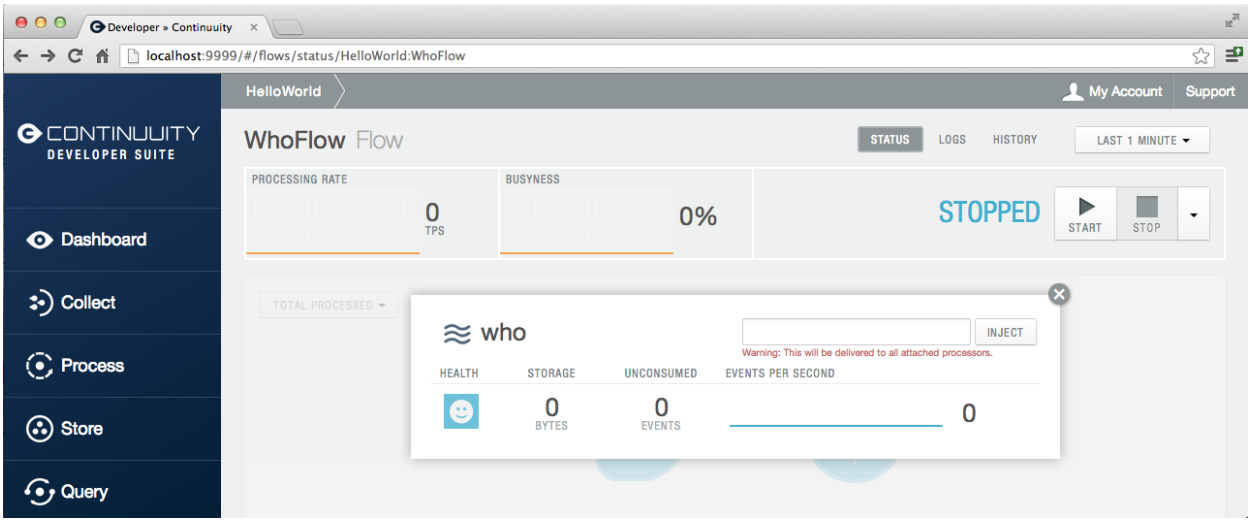
This code is included along with other examples in the Reactor Development Kit. To see this application working, first build it from the examples directory.

```
> cd continuuity-reactor-development-kit-1.6.1/examples/HelloWorld
> ant
```

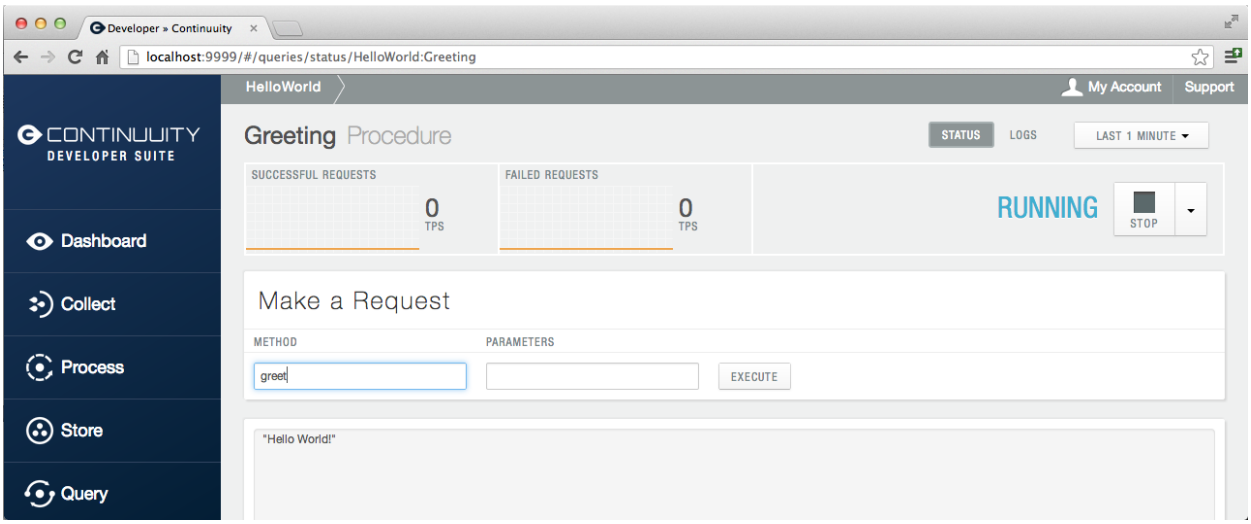
This creates an archive named `HelloWorld.jar` in the same directory. To deploy the application, start the Reactor:

```
> continuuity-reactor start
```

Go to the local user interface at http://localhost:9999/, click ADD AN APPLICATION on the right hand side, and drag the `HelloWorld.jar` onto the drop area. You will then see a dashboard showing one application named HelloWorld. Click it to see the stream, flow, dataset and procedure that belong to this application. To send a name to the stream, click the flow named WhoFlow and you will see a graphic rendering of the flow. Click the START button to start the flow, the click the stream item labeled "who" and enter a name:

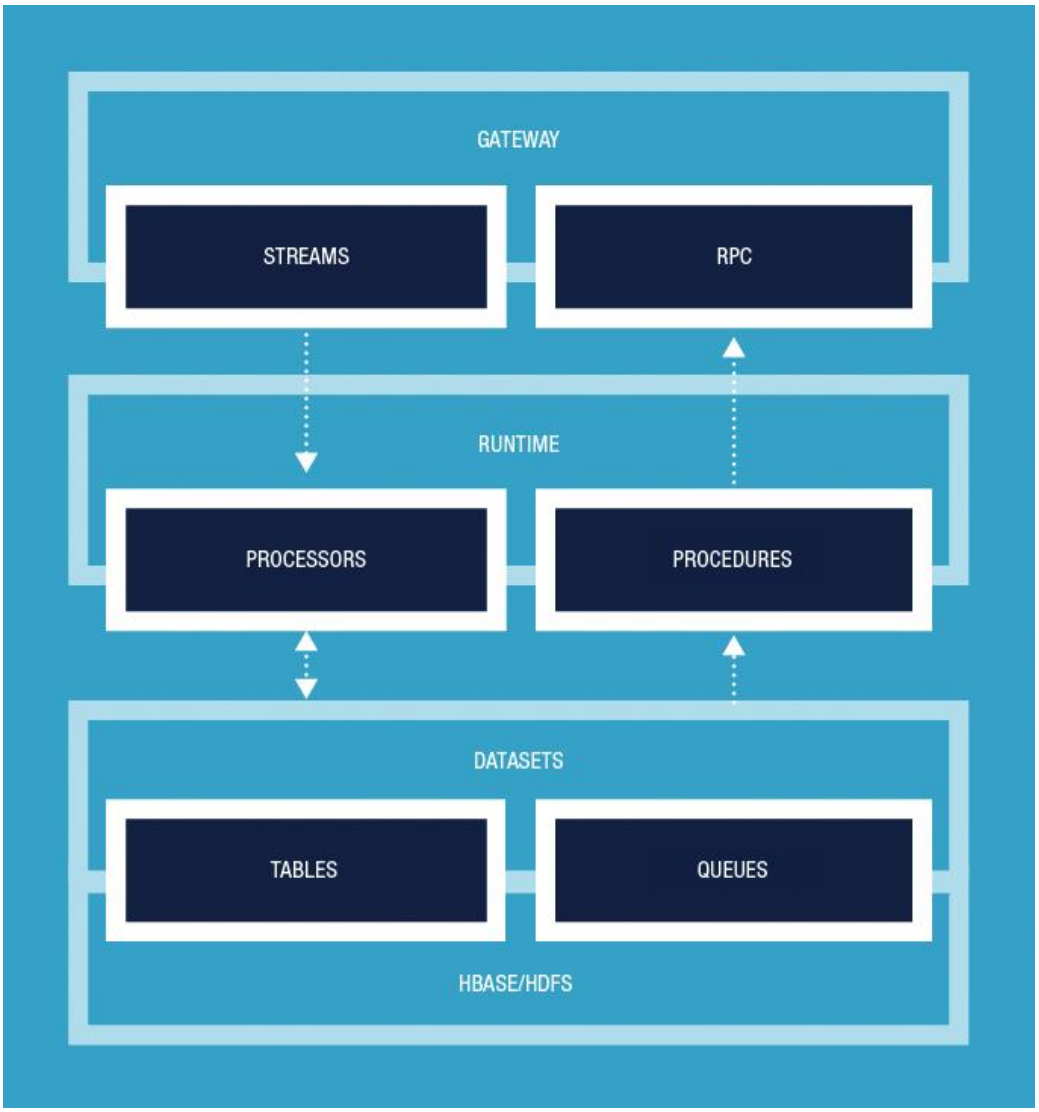


Click **Query** in the sidebar menu and you'll see the Greeting procedure. Click it to go to the Procedure screen, then click START to run the procedure. Now you can enter a query. Type `greet` into the METHOD box and click **EXECUTE to** see the response:



3. Understanding the Continuuity Reactor

The Continuuity Reactor is a unified Big Data application platform that brings various Big Data capabilities into a single environment and provides an elastic runtime for applications. Data can be stored in both structured and unstructured forms and ingestion, processing, and serving can be done in real-time.



The Reactor provides **Streams** for simple data ingestion from any external system, **Processors** for performing elastically scalable real-time stream processing, **Datasets** for storing data in a simple and scalable way without worrying about formats and schema, and **Procedures** for exposing data to external systems as simple or complex interactive queries. These are grouped into **Applications** for configuring and packaging into deployable Reactor artifacts.

You'll build applications in Java using the Continuuity Core APIs. Once your application is deployed and running, you can easily interact with it from virtually any external system by accessing the streams, datasets, and procedures using REST or other network protocols.

3.1. Collect with Streams

Streams are the primary means for sending data from external systems into the Reactor. You can write to streams easily using REST (see Section 5.2 on page 47) or command line tools (see Section 5.3 on page 57), either one operation at a time or in batches. Each individual signal sent to a stream is stored as an **Event**, which is comprised of a body (blob of arbitrary binary data) and headers (map of strings for metadata).

Streams are identified by a Unique Stream ID string and must be explicitly created before being used. They can be created using a command line tool (see Section 5.3 on page 57), the Management Dashboard, or programmatically within your application (see Section 4.1 on page 13). Data written to a stream can be consumed by flows and processed in real-time as described below.

3.2. Process with Flows

Flows are user-implemented real-time stream processors. They are comprised of one or more **Flowlets** that are wired together into a Directed Acyclic Graph (DAG). Flowlets pass **Data Objects** 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 (more about this in Sections 4.2 on page 23 and 4.3 on page 27).

Flows are deployed to the Reactor 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 get data into your flow, you can either connect the input of the flow to a stream, or you can implement a **Generator Flowlet,** which executes custom code in order to generate data or pull it from an external source.

To learn more about flows see Section 4.2 on page 23.

3.3. Store with Datasets

Datasets are your interface to the Reactor's storage engine, the DataFabric. Instead of requiring you to manipulate data with low-level DataFabric APIs, datasets provide higher level abstractions and generic, reusable Java implementations of common data patterns.

Types of Datasets

The **core** dataset of the DataFabric is a **Table**. Unlike relational database systems, these tables are not organized into rows with a fixed schema, and they are optimized for accessing and manipulating data at the column level. Tables allow for efficient storage of semi-structured data, data with unknown or variable schema, and sparse data.

All other datasets are built on top of the core datasets, that is, tables. For example, 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 tables into a more complex data pattern. For example, an indexed table can be implemented using one table for the data to index and a second table for the index.

You will also learn how to implement your own data patterns as **custom** datasets on top of tables. Because a number of useful datasets, including key/value tables, indexed tables and time series are already included with the Reactor, we call them **system** datasets.

To learn more about datasets see Section 4.4 on page 28.

3.4. Query with Procedures

Procedures allow you to make synchronous calls into the Reactor from external systems and perform server-side processing on-demand, similar to a stored procedure in a traditional database. A procedure implements and exposes a very simple API: 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.

Procedures are typically used to post-process data at query time. This post-processing can include filtering, aggregations, 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.

We will learn more about procedures in Section 4.1 on page 13.

3.5. Process in Batch with MapReduce

Batch programs allow you to process data with MapReduce. You can write your MapReduce jobs in the same way as you would with a conventional Hadoop system. In addition, you can access data sets from your MapReduce jobs, and use data sets as both input and output with MapReduce jobs. While a flow processes data as it arrives, with batch programs you can wait for a large amount of data to be collected and subsequently process that data in bulk. While batch processing does not happen in real-time as do flows do, it can achieve higher throughput.

3.6. Package with Applications

Applications are the highest-level concept and serve to specify and package all the elements and configurations of your Big Data application. Within the application, you can explicitly indicate (and if necessary, create) your streams and datasets and declare all of the flows, procedures, and MapReduce batch programs that make up the application.

3.7. Reactor Runtime Editions

The Continuuity Reactor can be run in different modes: in-memory mode for unit testing, local mode for local testing, and Hosted or Enterprise mode for staging and production. In addition, you have the option to get a free Sandbox Reactor in the cloud. Regardless of the runtime edition, the Reactor is fully functional and the code you develop never changes, however performance and scale are limited when using in-memory or local mode or a Sandbox Reactor.

In-Memory Reactor

The in-memory Reactor allows you to easily run the Reactor for use in JUnit tests. In this mode, the underlying Big Data infrastructure is emulated using in-memory data structures and there is no persistence. The dashboard is not available in this mode.

Local Reactor

The Local Reactor allows you to run the entire Reactor stack in a single JVM on your local machine and also includes a local version of the Reactor Dashboard. The underlying Big Data infrastructure is emulated on top of your local file system and a relational database. All data is persisted.

See the Continuuity Reactor Getting Started Guide for more information on how to start and manage your Local Reactor.

Sandbox Reactor

The Sandbox Reactor is a free version of the Reactor that is hosted and operated in the cloud. However, it does not provide the same scalability and performance as the Hosted Reactor the Enterprise Reactor. The Sandbox Reactor is a good way to experience all of the features of the "push-to-cloud" functionality of a Hosted Reactor without paying for a fully Hosted Reactor.

Hosted Reactor and Enterprise Reactor

The Hosted Reactor and the Enterprise Reactor run in fully distributed mode. This includes distributed and highly available deployments of the underlying Hadoop infrastructure in addition to the other system components of the Reactor. Production applications should always be run on a Hosted Reactor or an Enterprise Reactor.

To self-provision your free Sandbox Reactor, go to your Account Home page: https://accounts.continuuity.com.

To learn more about getting your own Hosted Reactor or Enterprise Reactor, see: http://www.continuuity.com/products.

4. Reactor Programming Guide

This section dives into more detail around each of the different Reactor core elements - Streams, Flows, Datasets, and Procedures - and how you work with them in Java to build your Big Data Application.

First there is an overview of all of the high-level concepts and core Java APIs. Then a deep-dive into the Flow System, Procedure System, Datasets, and Transactions will give an understanding of how these systems function. Finally an example application will be implemented to help illustrate these concepts and describe how an entire application is built.

4.1. Reactor Core APIs

This section briefly discusses the Reactor core APIs.

Application

An application is a collection of streams, flows, datasets, and procedures. To create an application implement the `Application` interface and its `configure()` method. This allows you to specify the application metadata, declare and configure the streams and datasets, and add associated flows and procedures.

```
public class MyApp implements Application {
  @Override
  public ApplicationSpecification configure() {
    return ApplicationSpecification.Builder.with()
      .setName("myApp")
      .setDescription("my sample app”)
      .withStreams().add(...) ...
      .withDataSets().add(...) ...
      .withFlows().add(...) ...
      .withProcedures().add(...) ...
      .withBatch().add(...)
      .build();
```

You can also specify that an application does not use a stream:

```
      .setDescription("my sample app”)
      .noStream().
      .withDataSets().add(...) ...
```

and so forth for all of the other constructs.

Stream

Streams are the primary means for pushing data to the Reactor. You can specify a stream in your application as follows:

```
      .withStreams().add(new Stream(“myStream”)) ...
```

Flow

Flows are a collection of connected flowlets wired into a DAG. To create a flow, implement the `Flow` interface and its `configure()` method. This allows you to specify the flow's metadata, flowlets, flowlet connections, stream to flowlet connections, and any datasets used in the flow via a `FlowSpecification`:

```
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();
  }
```

Flowlet

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 also 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, then emits the results. It has a very simple configuration method and does nothing for initialization and destruction (see additional sample code below for a simpler way to declare these methods):

```
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 FlowletException {
  }

  @Override
  public void destroy() {
  }
```

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:

```
OutputEmitter<Long> output;

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

Note that the flowlet declares the `OutputEmitter` but does not initialize it. The flow system injects its implementation at runtime. Also note that the method is annotated with `@ProcessInput` – this tells the flow system that this method can process input data. Another way to define this method is to have its name start with `process` – in which case no annotation is needed:

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

You can also 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 objects 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 can 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 by annotating the method with the origin name:

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

A process method can have an additional parameter, the input context. The input context provides information about the input object, such as its origin and the number of times the object has been retried (see Section 4.2 on page 23 for the retry logic of a flowlet). For example,

the following flowlet tokenizes text in a smart way and it uses the input context to decide what 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 compatibilities include:

- 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 A is compatible with 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 {
  ...
  public void processEvent(StreamEvent event) {
    ...
  }
```

GENERATOR

A special case of a flowlet is a generator flowlet. The difference from a standard flowlet is that a generator has no inputs. Instead of a process method, it defines a `generate()` method to emit data. This can be used, for instance, to generate test data, or to connect to an external data source and pull data from there.

For example, the following generator flowlet emits random numbers. It extends the `AbstractFlowlet` class that provides simple default implementations of the configure, initialize and destroy methods:

```
class RandomGenerator extends AbstractFlowlet implements GeneratorFlowlet {

  private OutputEmitter<Double> output;
  private Random random = new Random();

  @Override
  public void generate() {
    this.output.emit(random.nextDouble());
  }
}
```

Because this generator flowlet has an output of type Double, it can be connected to a `DoubleRoundingFlowlet`, which has a process method that accepts Double.

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, you can do the following when building the flow specification:

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

If you have two 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()
        .fromStream("random").to("rounding")
```

PROCEDURE

Procedures are a way to make calls to the Reactor from external systems and perform arbitrary server-side processing on demand.

To create a procedure you implement the `Procedure` interface, or more conveniently, you extend the `AbstractProcedure` class. It is configured and initialized similarly to a flowlet but instead of a process method, you define a handler method that, given a method name and map of string arguments, translated from an external request, sends a response. The most generic way to send a response is to obtain a `Writer` and stream out the response as bytes. You should make sure to close the writer when you are done:

```
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 in JSON. Procedures have a convenience methods for this:

```
    // 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(new ProcedureResponse(Code.SUCCESS), results);
```

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

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

DATASET

Datasets are the way you store and retrieve data. If your application uses a dataset, then you must declare it in the application specification. For example, to specify that your application uses a `KeyValueTable` dataset named `"myCounters"`, write:

```
  public ApplicationSpecification configure() {
    return ApplicationSpecification.Builder.with()
      ...
      .withDataSets().add(new KeyValueTable("myCounters"))
      ...
```

In order to use the dataset inside a flowlet or a procedure, you rely on the runtime system to inject an instance of the dataset. You do that with an annotation:

```
Class myFowlet extends AbstractFlowlet {

  @UseDataSet("myCounters")
  private KeyValueTable counters;
  ...
  void process(String key) throws OperationException {
    counters.increment(key.getBytes());
  }
```

The runtime system reads the dataset specification for the key/value table named `"myCounters"` from the metadata store and injects a functional instance of the dataset class into your code.

You can also implement your own datasets by extending the `DataSet` base class or extending any other existing type of dataset. More about this in Section 4.4 on page 28 on page and in the programming example in Section 4.5 on page 34.

BATCH

To batch process data using MapReduce, specify `withBatch()` in your application:

```
    public ApplicationSpecification configure() {
    return ApplicationSpecification.Builder.with()
      ...
      .withBatch().add(new WordCountJob())
```

You must implement the `MapReduce` interface, which requires the three methods `configure`, `beforeSubmit`, and `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 `Procedure`. It defines the name and description of the MapReduce job. Here you can also specify datasets to be used as input or output for the job (see Section 4.5 on page 34 for details).

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 `org.apache.hadoop.mapreduce.Job` to perform necessary job configuration, in the same way as if you were running the MapReduce 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, for example, you could perform cleanup or send a notification of job completion. 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
  }
```

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));
    }
  }
}
```

Read more about MapReduce in Section 4.5 on page 34.

Logging

The Reactor supports logging from flows and procedures using standard SLF4J APIs. For instance, in a flowlet you can write:

```
  private static Logger LOG = LoggerFactory.getLogger(WordCounter.class);
  ...
  public void process(String line) {
    LOG.debug(this.getContext().getName() + ": Received line " + line);
    ... // processing
    LOG.debug(this.getContext().getName() + ": Emitting count " + wordCount);
    output.emit(wordCount);
  }
```

The log messages emitted by your application code can be viewed in two different ways:

- All log messages of an application can be viewed in the dashboard, by clicking on the "Logs" button in the Flow and Procedure screens.
- In the Local Reactor, application log messages are also written to the system log files along with the messages emitted by the Reactor itself. These files are not available for viewing in the Sandbox Reactor.

4.2. The Flow System

Flows are user-implemented real-time stream processors. They are comprised of one or more flowlets that are wired together into a Directed Acyclic Graph (DAG). Flowlets pass data between one another; each flowlet is able to perform custom logic and execute data operations for each individual data object it processes.

A flowlet processes the data objects from its input one by one. If a flowlet has multiple inputs, they are consumed in a round-robin fashion. 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 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. See Section 4.3 on page 27 to learn more about transactions.

Sequential and Asynchronous Flowlet Execution

A flowlet processes the data from its inputs one object at a time, by repeating the following steps:

1. An object is dequeued from one of the inputs. This does not completely remove it from the input, but marks it as in-progress.
2. The matching `process()` method is selected and invoked, in a new transaction. The process method can perform dataset operations and emit data to its outputs.
3. Once the `process()` method returns, the transaction is committed, and:
    a. If the transaction **fails,** all the dataset operations are rolled back, and all emitted output data objects is discarded. The `onFailure()` callback of the flowlet is invoked, which allows you to retry or ignore
        i. If you retry, `process()` will be invoked with the same input object again
        ii. If you ignore, the failure will be ignored and the input object is permanently removed from the input.
    b. If the transaction **succeeds**, all dataset operations are persistently committed, all emitted output data is sent downstream, and the input object is permanently removed from the input. Then the `onSuccess()` callback of the flowlet is invoked

By default, steps 1, 2 and 3 above happen in sequence, one input object at a time. You can increase the throughput of the flowlet by declaring it as asynchronous:

```
@Async
class MyFlowlet implements Flowlet {
  ...
```

Then the three steps happen concurrently: While the transaction of one data object is committing, the flowlet already processes the next object from the input, and yet another object is already being read from the input at the same time. However, you should be aware that processing now happens in overlapping transactions, and the probability of write conflicts increases – especially if the duration of the transactions is long (The transaction system has special ways to avoid conflicts on the flowlet's inputs). Read more about Transactions in Section 4.3 on page 27.

Batch Execution in Flowlets

By default, a flowlet processes a single data object at a time within a single transaction. To increase throughput, you can also process a batch of data objects within the same transaction:

```
@Batch(100)
public void process(Iterator<String> words) {
  ...
```

In this example, 100 data objects are dequeued at one time and processed within a single transaction. Note that the signature of the process method is now different: Instead of a single object, it now expects an iterator over the input type.

As is the case with asynchronous processing, if you use batch processing, your transactions can take longer and the probability of a conflict increases. With batch execution, both asynchronous and synchronous modes can be used.

Flows and Instances

You can have one or *more* instances of any given flowlet, each consuming a disjoint partition of each input. You can control the number of instances programmatically, using the UI, or using the command line interface. This enables you to shape your application to meet capacity at runtime the same way you will do in production. In the local version provided with the Reactor Development Kit, multiples instances of a flowlet are run in threads, so in some cases actual performance may not be affected. In production, each instance is a separate JVM with independent compute resources.

Partitioning Strategies in Flowlets

If you have multiple instances of a flowlet, the input queue is partitioned among the instances. The partitioning can occur in different ways, and each flowlet can specify one of the following partitioning strategies:

- *First in first out (FIFO)*: **Default**. In this mode, every flowlet instance receives the next available data object in the queue. However, since multiple consumers may compete for the same data object, access to the queue must be synchronized. This may not always be the most efficient strategy.
- *Round robin*: This is the most efficient partitioning. Each instance receives every n^{th} item. Though this is more efficient than FIFO, it is not always optimal: some applications may need to group objects into buckets according to business logic. In such cases, use hash-based partitioning instead.
- *Hash-based partitioning*: If the emitting flowlet annotates each data object with a hash key, this partitioning ensures that all objects of a given key are received by the same consumer instance. This can be useful for aggregating by key, and can help reduce write conflicts (for more information on transaction conflicts, see section 4.3 on page 27).

Suppose we have a flowlet that counts words:

```
public class Counter extends AbstractFlowlet {

  @UseDataSet("wordCounts")
  private KeyValueTable wordCountsTable;

  @ProcessInput("wordOut")
  public void process(String word) throws OperationException {
    this.wordCountsTable.increment(Bytes.toBytes(word), 1L);
  }
}
```

This flowlet uses the default strategy of *FIFO*. To increase the throughput when this flowlet has many instances, we can specify round-robin partitioning:

```
@RoundRobin
@ProcessInput("wordOut")
public void process(String word) throws OperationException {
  this.wordCountsTable.increment(Bytes.toBytes(word), 1L);
}
```

Now, if we have 3 instances of this flowlet, every instance will receive every third word. For example, for the sequence of words in the sentence, "*I scream, you scream, we all scream for ice cream*":

- The first instance receives the words: *I scream scream cream*
- The second instance receives the words: *scream we for*

The potential problem with this is that both instances might attempt to increment the counter for the word *scream* at the same time, and that may lead to a write conflict. To avoid conflicts we can use hash partitioning:

```
@HashPartition("wordHash")
@ProcessInput("wordOut")
public void process(String word) throws OperationException {
  this.wordCountsTable.increment(Bytes.toBytes(word), 1L);
}
```

Now only one of the flowlet instances will receive the word *scream*, and there can be no more write conflicts. Note that in order to use hash partitioning, the emitting flowlet must annotate each data object with a hash key:

```
@Output("wordOut")
private OutputEmitter<String> wordOutput;
...
public void process(StreamEvent event) throws OperationException {
  ...
  // emit the word with a hash key name "wordHash"
  wordOutput.emit(word, "wordHash", word.hashCode());
}
```

Note that the emitter must use the same name (`"wordHash"`) for the key that the consuming flowlet specifies as the partitioning key. If the output is connected to more than one flowlet, you can also annotate a data object with multiple hash keys – each consuming flowlet can then use a different partitioning. This is useful if you want to aggregate by multiple keys, for example, if you want to count purchases by product ID as well as by customer.

Partitioning can be combined with batch execution:

```
@Batch(100)
@HashPartition("wordHash")
@ProcessInput("wordOut")
public void process(Iterator<String> words) throws OperationException {
  ...
```

GETTING DATA IN

Input data can be pushed to a flow using streams or pulled from within a flow using a generator flowlet**.**

- A **Generator Flowlet** actively retrieves or generates data, and the logic to do so is coded into the flowlet's generate() method. For instance, a generator flowlet can connect to the Twitter fire hose or another external data source.
- A **Stream** is passively receiving events from outside (remember that streams exist outside the scope of a flow). To consume a stream, connect the stream to a flowlet that implements a process method for `StreamEvent`. This is useful when your events come from an external system that can push data using REST calls. It is also useful when you're developing and testing your application, because your test driver can send mock data to the stream that covers all your test cases.

4.3. The Transaction System

A flowlet processes the data objects from its inputs one at a time. While processing a single input object, all operations, including the removal of the data from the input, and emission of data to the outputs, are executed in a transaction. This provides us with ACID properties:

- The process method runs under read **isolation** to ensure that it does not see dirty writes (uncommitted writes from concurrent processing) in any of its reads. It does see, however, its own writes.
- A failed attempt to process an input object leaves the DataFabric in a **consistent** state, that is, it does not leave partial writes behind.
- All writes and emission of data are committed **atomically**, that is, either all of them or none of them are persisted.
- After processing completes successfully, all its writes are persisted in a **durable** way.

In case of failure, the state of the DataFabric is unchanged and therefore, processing of the input object can be reattempted. This ensures exactly-once processing of each object.

The Reactor uses Optimistic Concurrency Control (OCC) to implement transactions. Unlike most relational databases that use locks to prevent conflicting operations between transactions, under OCC we allow these conflicting writes to happen. When the transaction is committed, we can detect whether it has any conflicts: namely if during the lifetime of this transaction, another transaction committed a write for one the same keys that this transaction has written. In that case, the transaction is aborted and all of its writes are rolled back.

In other words: If two overlapping transactions modify the same row, then the transaction that commits first will succeed, but the transaction that commits last is rolled back due to a write conflict.

Optimistic Concurrency Control is lockless and therefore avoids problems such as idle processes waiting for locks, or even worse, deadlocks. However, it comes at the cost of rollback in case of write conflicts. We can only achieve high throughput with OCC if the number of conflicts is small. It is therefore a good practice to reduce the probability of conflicts where possible:

- Keep transactions short. The Reactor attempts to delay the beginning of each transaction as long as possible. For instance, if your flowlet only performs write operations, but no read operations, then all writes are deferred until the process method returns. They are then performed and transacted, together with the removal of the processed object from the input, in a single batch execution. This minimizes the duration of the transaction.
- However, if your flowlet performs a read, then the transaction must begin at the time of the read. If your flowlet performs long-running computations after that read, then the transaction runs longer, too, and the risk of conflicts increases. It is therefore a good practice to perform reads as late in the process method as possible.
- There are two ways to perform an increment: As a write operation that returns nothing, or as a read-write operation that returns the incremented value. If you perform the read-write operation, then that forces the transaction to begin, and the chance of conflict increases. Unless you depend on that return value, you should always perform an increment as a write operation.
- Use hash partitioning for the inputs of highly concurrent flowlets that perform writes. This helps reduce concurrent writes to the same key from different instances of the flowlet.

Keeping these guidelines in mind will help you write more efficient code.

4.4. The Dataset System

Datasets are your interface to the DataFabric. Instead of having to manipulate data with low-level DataFabric APIs, datasets provide higher level abstractions and generic, reusable Java implementations of common data patterns. A dataset represents both the API and the actual data itself. In other words, a dataset class is a reusable, generic Java implementation of a common data pattern. A dataset Instance is a named collection of data with associated metadata, and it is manipulated through a Dataset class.

Types of Datasets

A dataset is a Java class that extends the abstract `DataSet` class with its own, custom methods. The implementation of a dataset typically relies on one or more underlying (embedded) datasets. For example, the `IndexedTable` dataset can be implemented by two underlying `Table` datasets, one holding the data itself, and one holding the index. We distinguish three categories of datasets: core, system, and custom datasets:

- The **core** dataset of the DataFabric is a **Table**. Its implementation is hidden from developers and it may use private DataFabric interfaces that are not available to you.
- A **custom** dataset is implemented by you and can have arbitrary code and methods. It is typically built around one or more tables (or other datasets) to implement a more specific data pattern. In fact, a custom dataset can only interact with the DataFabric through its underlying datasets.
- A **system** dataset is bundled with the Reactor but implemented in the same way as a custom dataset, relying on one or more underlying core or system datasets. The key difference between custom and system datasets is that system datasets are implemented (and tested) by Continuuity and as such they are reliable and trusted code.

Each dataset instance has exactly one dataset class to manipulate it - we think of the class as the type or the interface of the dataset. Every instance of a dataset has a unique name (unique within the account that it belongs to), and some metadata that defines its behavior. For example, every `IndexedTable` has a name and indexes a particular column of its primary table: The name of that column is a metadata property of each instance.

Every application must declare all datasets that it uses in its application specification. The specification of the dataset must include its name and all its metadata, including the specifications of its underlying datasets. This allows creating the dataset - if it does not exist yet - and storing its metadata at the time of deployment of the application. Application code (for example, a flow or procedure) can then use a dataset by giving only its name and type - the runtime system can use the stored metadata to create an instance of the dataset class with all required metadata.

Core Datasets - Tables

Tables are the only core dataset, and all other datasets are built using one or more underlying tables. A table is similar to tables in a relational database, but with a few key differences:

- Tables have no fixed schema. Unlike relational tables where every row has the same schema, every row of a table can have a different set of columns.
- Because the set of columns is not known ahead of time, the columns of a row do not have a rich type. All column values are byte arrays and it is up to the application to convert them to and from rich types. The column names and the row key are also byte arrays.
- When reading from a table, one need not know the names of columns: The read operation returns a map from column name to column value. It is, however, possible to specify exactly which columns to read.
- Tables are organized in a way that the columns of a row can be read and written independently of other columns, and columns are ordered in byte-lexicographic order. They are therefore also called Ordered Columnar Tables.

Interface

The table interface provides methods to perform read and write operations, plus a special increment method:

```
public class Table extends DataSet {

  public Table(String name);

  public OperationResult<Map<byte[], byte[]>> read(Read read)
    throws OperationException;

  public void write(WriteOperation op)
    throws OperationException;

  public Map<byte[], Long> increment(Increment increment)
    throws OperationException;
}
```

All operations can throw an `OperationException`. In case of success, the read operation returns an `OperationResult`, which is a wrapper class around the actual return type. In addition to carrying the result value, it can indicate that no result was found and the reason why.

```
class OperationResult<ReturnType> {
  public boolean isEmpty();
  public String getMessage();
  public int getStatus();
  public ReturnType getValue();
}
```

Read

To read from a table, specify a row key and optionally the columns to read:

```
Table table;
// reads all columns
result = table.read(new Read(rowkey));
// reads only one column
result = table.read(new Read(rowkey, column1));
// reads specified set of columns
result = table.read(new Read(rowkey, new byte[][] { col1, col2, col3 }));
// reads all columns from colA up to but excluding colB
result = table.read(new Read(rowkey, colA, colB));
// reads upto 100 columns from colA up to but excluding colB
result = table.read(new Read(rowkey, colA, colB, 100));
// read all columns up to, but excluding colB
result = table.read(new Read(rowkey, null, colB));
// read upto 100 columns starting with colA
result = table.read(new Read(rowkey, colA, null, 100));
// read all columns in the row
result = table.read(new Read(rowkey, null, null));
```

Write

There are four types of write operations: Write, Delete, Swap and Increment. A write specifies a row key, a set of column keys and the same number of column values:

```
// write a new value to a single column
table.write(new Write(rowkey, column1, value1));
// write multiple columns
table.write(new Write(rowkey, new byte[][] { col1, col2 },
                              new byte[][] { val1, val2 }))
```

> Note that the write does not replace the entire row: It only overwrites the values of the specified columns whereas existing values of other columns remain unmodified. If you want remove columns from a row, you must use a Delete operation (see below).

INCREMENT

An Increment interprets each column as an 8-byte (long) integer and increments it by a given value. If the column does not exist, it is created with the value that was given as the increment. If the existing value of the column is not 8 bytes long, the operation will throw an exception.

```
// write a new value to a single column
table.write(new Increment(rowkey, countCol, 1L));
// write multiple columns
table.write(new Increment(rowkey, new byte[][] { col1, col2 },
                                  new long[]   { 5   , 10   }));
```

> Note that this does not return the incremented values. If you want to use the result of the increment operation in subsequent code, you can use the `incrementAndGet` method. For example, to emit the incremented value to an output:

```
Map<byte[], Long> res =
  table.incrementAndGet(new Increment(rowkey, col, 1L));
Long incrementedValue = res.get(col);
output.emit(incrementedValue);
```

DELETE

A delete removes the specified columns from a row. Note that this does not remove the entire row. If you want to delete an entire row, you need to know its columns and specify each one.

```
// delete a single column
table.write(new Delete(rowkey, column1));
// delete multiple columns
table.write(new Delete(rowkey, new byte[][] { col1, col2 }));
```

SWAP

A swap operation compares the existing value of a column with an expected value, and if it matches, replaces it with a new value. This is useful to verify that a value has not changed since it was read. If it does not match, the operation fails and throws an exception. Read more about Optimistic Concurrency Control in Section 4.3.

```
// read a user profile
result = table.read(new Read(userkey, profileCol));
oldProfile = result.getValue().get(profileCol);
...
newProfile = manipulate(oldProfile, ...);
// fails if somebody else has updated it in the mean time
table.swap(userkey, profileCol, oldProfile, newProfile);
```

System Datasets

The Reactor comes with several system-defined datasets, including key/value tables, indexed tables and time series. Each of them is defined with the help one or more embedded tables, but defines its own interface. For example:

- The `KeyValueTable` implements a key/value store as a table with a single column.
- The `IndexedTable` implements a table with a secondary key using two embedded tables, one for the data and one for the secondary index.
- The `TimeseriesTable` uses a table to store keyed data over time and allows querying that data over ranges of time.

See the Java documentation of these classes to learn more about these datasets.

Custom Datasets

You can define your own dataset classes to implement common data patterns specific to your code. For example, suppose we want to define a counter table that in addition to counting words also counts how many unique words it has seen. The dataset will be built on top two underlying datasets, a `KeyValueTable` to count all the words and a core table for the unique count:

```
public class UniqueCountTable extends DataSet {

  private Table uniqueCountTable;
  private KeyValueTable wordCountTable;
```

In the constructor we take a name and create the two underlying datasets. Note that we use different names for the two tables, both derived from the name of the unique count table.

```
  public UniqueCountTable(String name) {
    super(name);
    this.uniqueCountTable = new Table("unique_count_" + name);
    this.wordCountTable = new KeyValueTable("word_count_" + name);
  }
```

Like most other elements of an application, the dataset must implement a configure() method that returns a specification. In the specification we save metadata about the dataset (such as its name) and the specifications of the embedded datasets obtained by calling their respective configure methods.

```
  @Override
  public DataSetSpecification configure() {
    return new DataSetSpecification.Builder(this)
        .dataset(this.uniqueCountTable.configure())
        .dataset(this.wordCountTable.configure())
        .create();
  }
```

So far, we have written all code needed to use the dataset in an application specification, that is, at application deploy time. At that time the dataset specification returned by `configure()` is stored with the application's metadata. At runtime the dataset must be available in all places that use the dataset, that is, in all instances of flowlets, procedures, or MapReduce jobs. To accomplish this the dataset is instantiated by reading its specification from the metadata store and calling the dataset's constructor that takes a dataset specification as argument. Every dataset class must implement such a constructor; otherwise, it will not be functional at runtime:

```
public UniqueCountTable(DataSetSpecification spec) {
    super(spec);
    this.uniqueCountTable = new Table(
        spec.getSpecificationFor("unique_count_" + this.getName()));
    this.wordCountTable = new KeyValueTable(
        spec.getSpecificationFor("word_count_" + this.getName()));
  }
```

Now we can begin with the implementation of the dataset logic. We start with a constant.

```
  // Row and column name used for storing the unique count
  private static final byte [] UNIQUE_COUNT = Bytes.toBytes("unique");
```

The dataset stores a counter for each word in its own row of the word count table, and for every word it increments its counter. If the resulting value is 1, then this was the first time we encountered the word, hence we have new unique word and we increment the unique counter.

```
  public void updateUniqueCount(String word)
      throws OperationException {
    // increment the counter for this word
    long newCount = wordCountTable.incrementAndGet(Bytes.toBytes(word), 1L);
    if (newCount == 1L) { // first time? Increment unique count
      uniqueCountTable.write(new Increment(UNIQUE_COUNT, UNIQUE_COUNT, 1L));
    }
  }
```

Note how this method first uses the `incrementAndGet()` method to increase the count for the word, because it needs to know the result to decide whether this is a new unique word. But the second increment is done with the `write()` method of the table, which does not return a result. Unless you really need to get back the result, you should always use that method, because it can be optimized for higher performance by the transaction engine (more details on that in Section 4.3 on page 27).

Finally, we write a method to retrieve the number of unique words seen. This method is extra cautious as it verifies that the value of the unique count column is actually eight bytes long.

```
public Long readUniqueCount() throws OperationException {
    OperationResult<Map<byte[], byte[]>> result =
        this.uniqueCountTable.read(new Read(UNIQUE_COUNT, UNIQUE_COUNT));
    if (result.isEmpty()) {
      return 0L;
    }
    byte [] countBytes = result.getValue().get(UNIQUE_COUNT);
    if (countBytes == null || countBytes.length != 8) {
      return 0L;
    }
    return Bytes.toLong(countBytes);
  }
```

You may have noticed that we only use one single cell of the `uniqueCountTable`, and we could certainly write this code more efficiently (yet the purpose of this example is to illustrate how to implement a custom dataset on top of multiple underlying datasets).

This concludes the implementation of this dataset.

4.5. MapReduce Programs and Datasets

A MapReduce job can interact with a dataset by using it as input or output, or it can directly read or write a dataset similar to a flowlet or procedure. For a procedure, you must:

- Declare the dataset in the MapReduce job's `configure()` method. For example, to have access to a dataset named `catalog`, you write:

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

- Inject the dataset into the mapper or reducer implementation where you need to use 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());
      ...
    }
```

BatchReadable and BatchWritable Datasets

When you run a MapReduce job, you can configure it to read its input from a dataset instead of the file system. That dataset must implement the `BatchReadable` interface, which requires two methods:

```
public interface BatchReadable<KEY, VALUE> {
  List<Split> getSplits() throws OperationException;
  SplitReader<KEY, VALUE> createSplitReader(Split split);
}
```

These two methods complement each other: `getSplits()` must return all splits of the dataset that the MapReduce job will read; `createSplitReader()` is then called in every mapper to read one of the splits. Note that the `KEY` and `VALUE` type parameters of the split reader must match the input key and value type parameters of the mapper.

Because `getSplits()` has no arguments, it will typically create splits that cover the entire dataset. If you want to use a custom selection of the input data, you can define another method in your dataset that takes additional parameters, and explicitly set the input in the `beforeSubmit()` method. For example, the system dataset `KeyValueTable` implements `BatchReadable<byte[], byte[]>` with an extra method that allows to specify the number of splits and a range of keys:

```
public class KeyValueTable extends DataSet
                           implements BatchReadable<byte[], byte[]> {
  ...
  public List<Split> getSplits(int numSplits, byte[] start, byte[] stop);
}
```

To read only a range of keys and give a hint that you want to get 16 splits, write:

```
@Override
@UseDataSet("myTable")
KeyValueTable kvTable;
...
public void beforeSubmit(MapReduceContext context) throws Exception {
  ...
  context.setInput(kvTable, kvTable.getSplits(16, startKey, stopKey));
}
```

Similarly to reading input from a dataset, you have the option to write to a dataset as the output destination of a MapReduce job – if that dataset implements the `BatchWritable` interface:

```
public interface BatchWritable<KEY, VALUE> {
  void write(KEY key, VALUE value) throws OperationException;
}
```

The `write()` method is used to redirect all writes performed by a reducer to the dataset. Again, the `KEY` and `VALUE` type parameters must match the output key and value type parameters of the reducer.

Transactions

When you run a MapReduce job that interacts with datasets, the system creates a long-running transaction for the job. Similar to the transaction of a flowlet or a procedure:

- Reads can only see the writes of other transactions that were committed at the time the long-running transactions was started.
- All writes of the long-running transaction are committed atomically, and only become visible to others after they are committed.
- The long-running transaction can read its own writes.

However, there is a key difference: Long-running transactions do not participate in conflict detection. If another transaction overlaps with the long-running transaction and writes to the same row, it will not cause a conflict but simply overwrite it. The reason for this is that if conflict detection were performed, the long-running transaction would almost always be aborted (because it is likely that the long-running transaction commits after the other transaction). Because of this, it is not recommended to write to the same datasets from both real-time and batch programs. It is better to use different datasets, or at least ensure that the real-time processing writes to a disjoint set of columns.

Also, Hadoop will reattempt a task (mapper or reducer) if it fails. If the task is writing to a dataset – be it via `@UseDataSet` or `@UseOutputDataSet` – the reattempt of the task will most likely repeat the writes that were already performed in the failed attempt. Therefore it is highly advisable that all writes performed by MapReduce programs be idempotent.

4.6. End-to-End Programming Example

To illustrate how the capabilities of the Reactor can be used to build an application, we can implement a simple end-to-end example using a slightly modified version of the classic Word Count[1]. This application, named **WordCount**, consists of the following:

1. A stream named `wordStream` that receives strings of words to be counted
2. A flow named `WordCounter` that processes the strings from the stream to calculate the word counts and other word statistics using four flowlets:
    - The `splitter` splits the input string into words and aggregates and persists global statistics.
    - The `counter` takes words as inputs and calculates and persists per-word statistics.
    - The `unique` flowlet is used to calculate the unique number of words seen.
    - The `associator` stores word associations between all the words in each input string.
3. A procedure named `RetrieveCounts` to serve read requests for the calculated word counts, statistics, and associations. It supports two methods:
    - `getCount` for accessing the word count of a specified word and it's word associations.
    - `getStats` for accessing the global word statistics.
4. Four datasets used by the flow and query in order to model, store, and serve the necessary data
    - A core `Table` named `wordStats` to track global word statistics.
    - A system `KeyValueTable` dataset named `wordCounts` to count the occurrences of each word.
    - A custom `UniqueCountTable` dataset named `uniqueCount` to determine and count the number of unique words seen.
    - A custom `AssociationTable` dataset named `wordAssocs` to track associations between words.

[1] http://wiki.apache.org/hadoop/WordCount

Defining The Application

The definition of the application is straightforward and simply wires together all of the different elements described:

```
public class WordCount implements Application {
  @Override
  public ApplicationSpecification configure() {
    return ApplicationSpecification.Builder.with()
      .setName("WordCount")
      .setDescription("Example Word Count App")
      .withStreams()
        .add(new Stream("wordStream"))
      .withDataSets()
        .add(new Table("wordStats"))
        .add(new KeyValueTable("wordCounts"))
        .add(new UniqueCountTable("uniqueCount"))
        .add(new AssociationTable("wordAssocs"))
      .withFlows()
        .add(new WordCounter())
      .withProcedures()
        .add(new RetrieveCounts())
      .noBatch()
      .build();
  }
}
```

Defining The Flow

The flow must define a configure methods that wires up the flowlets with their connections. Note that here we need not declare the streams and datasets used:

```
public class WordCounter implements Flow {
  @Override
  public FlowSpecification configure() {
    return FlowSpecification.Builder.with()
        .setName("WordCounter")
        .setDescription("Example Word Count Flow")
        .withFlowlets()
            .add("splitter", new WordSplitter())
            .add("counter", new Counter())
            .add("associator", new WordAssociator())
            .add("unique", new UniqueCounter())
        .connect()
            .fromStream("wordStream").to("splitter")
            .from("splitter").to("counter")
            .from("splitter").to("associator")
            .from("counter").to("unique")
        .build();
  }
```

The `splitter` is directly connected to the `wordStream`. It splits each input into words and sends them to the `counter` and the `associator`, the first of which forwards them to the `unique` counter flowlet.

Implementing Flowlets

With the application and flow defined, it's now time to dig into the actual logic of our application. The processing logic and data operations occur within flowlets. For each flowlet you extend the `AbstractFlowlet` base class.

The `WordCounter` contains four different flowlets: `splitter`, `counter`, `unique`, and `associator`. The implementation of each of these is shown below with an overview of each.

SPLITTER FLOWLET

The first flowlet in the flow is `splitter`. For each event from the `wordStream`, it interprets the body as a string and splits it into individual words, removing any non-alphabet characters from the words. It counts the number of words and computes the aggregate length of all words, and it writes both these values as increments to the `wordStats` table that keeps track of the total word and character count. It then emits each word separately to one of its outputs, for consumption by the `counter`, and the list of all words to its other output, for the `associator`.

```
public class WordSplitter extends AbstractFlowlet {
  @UseDataSet("wordStats")
  private Table wordStatsTable;

  private static final byte[] TOTALS_ROW = Bytes.toBytes("totals");
  private static final byte[] TOTAL_LENGTH = Bytes.toBytes("total_length");
  private static final byte[] TOTAL_WORDS = Bytes.toBytes("total_words");

  @Output("wordOut")
  private OutputEmitter<String> wordOutput;
  @Output("wordArrayOut")
  private OutputEmitter<List<String>> wordListOutput;

  public void process(StreamEvent event) throws OperationException {
    // Input is a String, need to split it by whitespace
    String inputString = Charset.forName("UTF-8")
                                .decode(event.getBody()).toString();

    String [] words = inputString.split("\\s+");
    List<String> wordList = new ArrayList<String>(words.length);

    long sumOfLengths = 0, wordCount = 0;

    // We have an array of words, now remove all non-alpha characters
    for (String word : words) {
      word = word.replaceAll("[^A-Za-z]", "");
      if (!word.isEmpty()) {
        // emit every word that remains
        wordOutput.emit(word);
        wordList.add(word);
        sumOfLengths += word.length();
        wordCount++;
      }
    }
    // Count other word statistics (word length, total words seen)
    this.wordStatsTable.write(
      new Increment(TOTALS_ROW,
                    new byte[][] { TOTAL_LENGTH,  TOTAL_WORDS },
                    new long[]   { sumOfLengths,  wordCount}));

    // Send the list of words to the associater
    wordListOutput.emit(wordList);
  }
}
```

Counter Flowlet

The counter flowlet receives single words as inputs. It counts the number of occurrences of each word in the key/value table wordCounts:

```
public class Counter extends AbstractFlowlet {
  @UseDataSet("wordCounts")
  private KeyValueTable wordCountsTable;

  private OutputEmitter<String> wordOutput;

  @ProcessInput("wordOut")
  public void process(String word) throws OperationException {

    // Count number of times we have seen this word
    this.wordCountsTable.increment(Bytes.toBytes(word), 1L);
    // Forward the word to the unique counter flowlet to do the unique count
    wordOutput.emit(word);
  }
}
```

Unique Counter Flowlet

The unique flowlet receives each word from the counter flowlet. Its data logic is coded into the UniqueCountTable dataset – we already know it from Section 4.4 on page 28. This dataset uses tables to determine the number of unique words seen.

```
public class UniqueCounter extends AbstractFlowlet {

  @UseDataSet("uniqueCount")
  private UniqueCountTable uniqueCountTable;

  public void process(String word) throws OperationException {
    this.uniqueCountTable.updateUniqueCount(word);
  }
}
```

Associator Flowlet

The associator flowlet receives arrays of words and stores associations between them using the AssociationTable custom dataset:

```
public class WordAssociator extends AbstractFlowlet {

  @UseDataSet("wordAssocs")
  private AssociationTable associationTable;

  public void process(Set<String> words) throws OperationException {
    // Store word associations
    this.associationTable.writeWordAssocs(words);
  }
}
```

Note that even though process expects a set of strings, it can consume lists of strings from the splitter flowlet – that is the magic of type projection!

IMPLEMENTING CUSTOM DATASETS

This application uses four datasets: A core table, a system key/value table, and two custom datasets built using core tables. The first custom dataset is the `UniqueCountTable` that we already discussed in Section 4.4 on page 28.

The other custom dataset is the `AssociationTable` and it tracks associations between words by counting the number of times they occur together. Rather than requiring that this pattern be implemented within our flowlets and procedures, it is implemented as a custom dataset, exposing a simple and specific API rather than a complex and generic one. Its interface has two public methods, `writeWordAssocs()` and `readWordAssocs()` that both use a core table. First of all, as all datasets, it must define two constructors and a `configure()` method (see Section 4.4 on page 28):

```
public class AssociationTable extends DataSet {

  private Table table;

  public AssociationTable(String name) {
    super(name);
    this.table = new Table("word_assoc_" + name);
  }

  public AssociationTable(DataSetSpecification spec) {
    super(spec);
    this.table = new Table(
      spec.getSpecificationFor("word_assoc_" + this.getName()));
  }

  @Override
  public DataSetSpecification configure() {
    return new DataSetSpecification.Builder(this)
      .dataset(this.table.configure())
      .create();
  }
```

The dataset operates on bags of words to compute for each word the set of other words that most frequently occur in the same bag. It uses a columnar table to count the number of times that two words occur together. That counter is in a cell of the table with the first word as the row key and the second word as the column key.

```
public void writeWordAssocs(Set<String> words) throws OperationException {
    for (String rootWord : words) {
      for (String assocWord : words) {
        if (!rootWord.equals(assocWord)) {
          this.table.write(new Increment(Bytes.toBytes(rootWord),
                                         Bytes.toBytes(assocWord), 1L));
        }
      }
    }
  }
```

Note that this table is very sparse – most words never occur together and will never be counted. In a table with a fixed schema, such as a relational table, this would be a very space-consuming representation. In a columnar table, however, non-existent cells occupy (almost) no space. Furthermore, we can use the second word, which is only known at runtime, as the column key. That would also be impossible in a traditional relational table.

Even though this method is very intuitive to understand, it is quite inefficient, because for a set of *N* words it performs $(N-1)^2$ increment operations, each for a single column. The table dataset allows incrementing multiple columns in a single operation, and we can make use of that to optimize this method:

```
public void writeWordAssocs(Set<String> words) throws OperationException {

    // for sets of less than 2 words, there are no associations
    int n = words.size();
    if (n < 2) return;

    // every word will get (n-1) increments (one for each of the other words)
    long[] values = new long[n - 1];
    Arrays.fill(values, 1);

    // convert all words to bytes, do this one time only
    byte[][] wordBytes = new byte[n][];
    int i = 0;
    for (String word : words) {
      wordBytes[i++] = Bytes.toBytes(word);
    }

    // generate an increment for each word, with all other words as columns
    for (int j = 0; j < n; j++) {
      byte[] row =  wordBytes[j];
      byte[][] columns = new byte[n - 1][];
      System.arraycopy(wordBytes, 0, columns, 0, j);
      System.arraycopy(wordBytes, j + 1, columns, j, n - j - 1);
      this.table.write(new Increment(row, columns, values));
    }
  }
```

To read the most frequent associations for a word, the dataset method simply reads the row for the word, iterates over all columns and passes them to a top-K collector (let's assume that is already implemented).

```
  public Map<String,Long> readWordAssocs(String word, int limit)
    throws OperationException {

    // Retrieve all columns of the word's row
    OperationResult<Map<byte[], byte[]>> result =
      this.table.read(new Read(Bytes.toBytes(word), null, null));
    TopKCollector collector = new TopKCollector(limit);
    if (!result.isEmpty()) {
      // iterate over all columns
      for (Map.Entry<byte[],byte[]> entry : result.getValue().entrySet()) {
        collector.add(Bytes.toLong(entry.getValue()),
                      Bytes.toString(entry.getKey()));
      }
    }
    return collector.getTopK();
  }
```

Implementing a Procedure

To read the data written by the flow, we implement a procedure, which will bind handlers to REST endpoints to get external access. The procedure has two methods, one to get the overall statistics and one to get the statistics for a single word. It begins with declaring the four datasets used by the application.

```
public class RetrieveCounts extends AbstractProcedure {

  @UseDataSet("wordStats")
  private Table wordStatsTable;
  @UseDataSet("wordCounts")
  private KeyValueTable wordCountsTable;
  @UseDataSet("uniqueCount")
  private UniqueCountTable uniqueCountTable;
  @UseDataSet("wordAssocs")
  private AssociationTable associationTable;
```

Now we can implement a handler for the first method, `getStats`. It returns general statistics across all words seen:

```
  @Handle("getStats")
  public void getStats(ProcedureRequest request,
                       ProcedureResponder responder) throws Exception {

    long totalWords = 0L, uniqueWords = 0L;
    double averageLength = 0.0;

    // Read the total length and total count to calculate average length
    OperationResult<Map<byte[],byte[]>> result =
      this.wordStatsTable.read(
        new Read(TOTALS_ROW, new byte[][] { TOTAL_LENGTH, TOTAL_WORDS }));
    if (!result.isEmpty()) {
      // extract the total sum of lengths
      byte[] lengthBytes = result.getValue().get(TOTAL_LENGTH);
      Long totalLength = lengthBytes == null ? 0L : Bytes.toLong(lengthBytes);
      // extract the total count of words
      byte[] wordsBytes = result.getValue().get(TOTAL_WORDS);
      totalWords = wordsBytes == null ? 0L : Bytes.toLong(wordsBytes);
      // compute the average length
      if (totalLength != 0 && totalWords != 0) {
        averageLength = (double)totalLength/(double)totalWords;
        // Read the unique word count
        uniqueWords = this.uniqueCountTable.readUniqueCount();
      }
    }

    // return a map as JSON
    Map<String, Object> results = new TreeMap<String, Object>();
    results.put("totalWords", totalWords);
    results.put("uniqueWords", uniqueWords);
    results.put("averageLength", averageLength);
    responder.sendJson(new ProcedureResponse(Code.SUCCESS), results);
  }
```

The second handler is for the method `getCount`. Given a word, it returns the count of the word together with the top words associated with that word, up to a specified limit, or up to 10 if no limit is given:

```
@Handle("getCount")
public void getCount(ProcedureRequest request,
                     ProcedureResponder responder) throws Exception {
  String word = request.getArgument("word");
  if (word == null) {
    responder.error(Code.CLIENT_ERROR,
                    "Method 'getCount' requires argument 'word'");
    return;
  }

  String limitArg = request.getArgument("limit");
  int limit = limitArg == null ? 10 : Integer.valueOf(limitArg);

  // Read the word count
  byte[] countBytes = this.wordCountsTable.read(Bytes.toBytes(word));
  Long wordCount = countBytes == null ? 0L : Bytes.toLong(countBytes);

  // Read the top associated words
  Map<String,Long> wordsAssocs =
    this.associationTable.readWordAssocs(word, limit);

  // return a map as JSON
  Map<String, Object> results = new TreeMap<String, Object>();
  results.put("word", word);
  results.put("count", wordCount);
  results.put("assocs", wordsAssocs);
  responder.sendJson(new ProcedureResponse(Code.SUCCESS), results);
}
```

This concludes the `WordCount` example. You can find it in the examples directory of the Reactor Development Kit.

4.7. Testing Your Applications

The Reactor comes with a convenient way to unit test your applications. The base for these tests is `AppFabricTestBase`, which is packaged separately from the API in its own artifact because it depends on the Reactor's runtime classes. You can include it in your test dependencies in two ways:

- Include all JAR files in the `lib` directory of the Reactor Development Kit installation
- Include the `continuuity-test` artifact in your Maven test dependencies (see the `pom.xml` of `WordCount` for an example).

Note that for building an application, you only need to include the Reactor API in your dependencies; for testing, however, you need the Reactor run-time. To build your test case, extend the `AppFabricTestBase` class. Let's write a test case for the `WordCount` example:

```
public class WordCountTest extends AppFabricTestBase {

  @Test
  public void testWordCount() throws Exception {
```

The first thing we do in this test is deploy the application, then we'llstart the flow and the procedure:

```
    // deploy the application
    ApplicationManager appManager = deployApplication(WordCount.class);

    // start the flow and the procedure
    FlowManager flowManager = appManager.startFlow("WordCounter");
    ProcedureManager procManager = appManager.startProcedure("RetrieveCount");
```

Now that the flow is running, we can send some events to the stream:

```
    // send a few events to the stream
    StreamWriter writer = appManager.getStreamWriter("wordStream");
    writer.send("hello world");
    writer.send("a wonderful world");
    writer.send("the world says hello");
```

To wait for all events to be processed, we can get a metrics observer for the last flowlet in the pipeline, the word associator, and wait for its processed count to reach 3, or time out after 5 seconds:

```
    // wait for the events to be processed, or at most 5 seconds
    RuntimeMetrics metrics = RuntimeStats.
      getFlowletMetrics("WordCount", "WordCounter", "associator");
    metrics.waitForProcessed(3, 5, TimeUnit.SECONDS);
```

Now we can start verifying that the processing was correct. Start by obtaining a client for the procedure, and submitting a query for the global statistics:

```
    // now call the procedure
    ProcedureClient client = procManager.getClient();
    // query global statistics
    String response = client.query("getStats", Collections.EMPTY_MAP);
```

If the query fails for any reason this method would throw an exception. In case of success, the response is a JSON string. We must deserialize the JSON in order to verify the results:

```
    Map<String, String> map = new Gson().fromJson(response, stringMapType);
    Assert.assertEquals("9", map.get("totalWords"));
    Assert.assertEquals("6", map.get("uniqueWords"));
    Assert.assertEquals(((double)42)/9,
      (double)Double.valueOf(map.get("averageLength")), 0.001);
```

Then we ask for the statistics of one of the words in the test events. The verification is a little more complex, because we now have a nested map as a response, and the value types in the top-level map are not uniform.

```
    // verify some statistics for one of the words
    response = client.query("getCount", ImmutableMap.of("word","world"));
    Map<String, Object> omap = new Gson().fromJson(response, objectMapType);
    Assert.assertEquals("world", omap.get("word"));
    Assert.assertEquals(3.0, omap.get("count"));
    // the associations are a map within the map
    Map<String, Double> assocs = (Map<String, Double>) omap.get("assocs");
    Assert.assertEquals(2.0, (double)assocs.get("hello"), 0.000001);
    Assert.assertTrue(assocs.containsKey("hello"));
  }
}
```

5. API and Tool Reference

5.1. Java APIs

The Javadocs for all Core Reactor Java APIs is included in the Reactor Development Kit:

```
./continuuity-reactor-development-kit-1.6.1/javadocs/index.html
```

Note that some APIs are annotated as `@Beta`. They represent experimental features that have not been fully tested or documented yet – they may or may not be functional. Also, these APIs may be removed in future versions of the Reactor SDK. Use them at your own discretion.

5.2. REST APIs

The Continuuity Reactor Platform exposes these REST interfaces:

1. **Stream**: To send data events to a stream or to inspect the contents of a stream.
2. **Data:** To interact with datasets (currently, only Tables).
3. **Procedure**: To send queries to a procedure.
4. **Monitor:** To monitor the status of running applications and to retrieve metrics.
5. **Reactor:** To deploy and manage applications.

Common return codes for all REST calls:

- *200 OK*: The request returned successfully
- *400 Bad Request*: The request had a combination of parameters that is not recognized/allowed
- *401 Unauthorized*: The request did not contain an authentication token
- *403 Not Allowed*: The request was authenticated but the client does not have permission
- *404 Not Found*: The request did not address any of the known URIs
- *405 Method Not Allowed*: A request with an unsupported method was received
- *500 Internal Server Error*: An internal error occurred while processing the request
- *501 Not Implemented*: A request contained a query, which is not supported by this API

Note: These may be omitted from the descriptions below but any request may return them.

REST Endpoint Port Configuration

By default, each of the three REST interfaces binds to a set of default ports on localhost. You can modify these ports by modifying the *continuuity-site.xml* in your Reactor Development Kit `conf/` directory. The properties you can modify are:

Description	Property	Default Value
Stream REST Port	`stream.rest.port`	10000
Data REST Port	`data.rest.port`	10002
Procedure REST Port	`procedure.rest.port`	10010
Monitor REST Port	`monitor.rest.port`	10005
Reactor Client REST Port	`app.rest.port`	10007

The descriptions below assume the default ports.

When you interact with a Sandbox Reactor, all REST APIs require that you use SSL for the connection and that you authenticate your request by sending your API key in an HTTP header:

X-Continuuity-ApiKey : <API key>

STREAM REST API

This interface supports creating streams, sending events to a stream and reading single events from a stream.

CREATING A STREAM

A stream can be created with an HTTP put request:

PUT http://<hostname>:10000/stream/<new-stream-id>

The request returns *200 OK* in case of success.

SENDING EVENTS

A request to send an event to a stream is an HTTP post. The URI is formed as

POST http://<hostname>:10000/stream/<stream-id>

where the *<stream-id>* identifies an existing stream. The body of the request must contain the event in binary form. You can pass headers for the event as HTTP headers, by prefixing them with the stream id, that is:

<stream-id>.<property > : <string value>

After receiving the request, the REST connector will transform it into a stream event as follows:

- The body of the event is an identical copy of the bytes in the body of the HTTP post request
- If the request contains any headers prefixed with the stream id, then stream id prefix is stripped from the header name and the header is added to the event.

Return codes for the request are:

- *200 OK*: Everything went well.
- *404 Not found*: The stream does not exist.

The response will always have an empty body.

Reading Events

Streams may have multiple consumers (for example multiple flows), each of which may be a group of different agents (for example multiple instances of a flowlet). In order to read, a client must first obtain a consumer (group) id, which needs to be passed to subsequent read requests.

Getting a consumer id is performed as an HTTP GET to the URL

GET http://<hostname>:10000/stream/<stream-id>?q=newConsumer

The new consumer id is returned in a response header and, for convenience, also in the body of the response.

X-Continuuity-ConsumerId: <consumer-id>

Return codes:

- *201 Created*: Everything went well, and the new consumer is returned.
- *404 Not found*: The stream does not exist.

Once this is completed single events can be read from the stream, in the same way that a flow reads events. That is, the read will always return the event from the queue that was inserted first and has not been read yet (FIFO semantics). In order to read the third event that was sent to a stream, two previous reads have to be performed. Note that you can always start reading from the first event by getting a new consumer id. A read is performed as an HTTP get to the URI:

GET http://<hostname>:10000/stream/<stream-id>?q=dequeue

and the request must pass the consumer id in a header of the form

X-Continuuity-ConsumerId: <consumer-id>

The response will contain the binary body of the event in its body and a header

<stream-id>.<property > : <value>

for each header of the stream event, analogous to how you send headers in the post request.

Return codes:

- *200 OK*: Everything went well.
- *204 No Content*: The stream exists but it is empty or the given consumer id has read all events in the stream.
- *404 Not found*: The stream does not exist.

Reading Multiple Events

Reading multiple events is not supported directly by the stream API, but the stream-client tool has a way to view all, the first N, or the last N events in the stream. For more information see the command-line guide in Section 5.3 on page 57.

DATA REST API

The data API allows you to interact with tables (the core datasets) through REST. You can create tables and read, write, modify, or delete data.

CREATE A TABLE

To create a new table, issue an HTTP PUT request:

PUT http://<hostname>:10002/data/Table/<table-name>

This will create a table with the given name. The table name should only contain ASCII letters, digits and hyphens. If a table with the same name already exists, no error is returned, and the existing table remains in place. However, if a dataset of a different type exists with the same name, for example a key/value table, this call will return an error. The expected return codes are:

- *200 OK*: Everything went well.
- *409 Conflict*: A dataset of a different type already exists with the given name.

WRITE DATA TO A TABLE

To write to a table, send an HTTP PUT request to the table's URL:

PUT http://<hostname>:10002/data/Table/<table-name>/<row-key>

In the body, you must specify the columns and values that you want to write as a JSON string map, for example:

```
{"x":"y","y":"a","z":"1"}
```

This writes three columns named `x`, `y`, and `z` with values `y`, `a`, and `1`, respectively. Note that we specify all column keys and values are strings, whereas in the Data Fabric, all keys and values are byte arrays. For now, let us just assume that these byte values can simply be converted to Strings, and we will soon learn how to use keys and values that are not ASCII strings.

The request will return:

- *200 OK*: Everything went well.
- *400 Bad Request:* The JSON string is not well-formed or cannot be parsed as a map from string to string.
- *404 Not found*: A table with the given name does not exist.

READ DATA FROM A TABLE

To read from a table, address the row that you want to read directly in an HTTP GET request:

GET http://<hostname>:10002/data/Table/<table-name>/<row-key>

The response will be a JSON string representing a map from column name to value. For example, reading back the row that we wrote in the previous, the response is:

```
{"x":"y","y":"a","z":"1"}
```

If you are only interested in some of the columns, you can specify a list of columns explicitly or give a range of columns, in all the same ways that you specify the columns for a `Read` operation (see Section 4.4 Core Datasets - Tables on page 29. For example:

GET ...<table-name>/<row-key>?columns=x,y	returns only columns `x` and `y`.
GET ...<table-name>/<row-key>?start=c5	returns all columns greater or equal to `c5`.
GET ...<table-name>/<row-key>?stop=c5	returns all columns less than (exclusive) `c5`.
GET ...<table-name>/<row-key>?start=c2&stop=c5	returns all columns between `c2` and (exclusive) `c5`.

The request will return:

- *200 OK*: Everything went well.
- *404 Not found*: A table with the given name does not exist.

Increment Data in a Table

You can also perform an atomic increment of cells of a table, and receive back the incremented values, by posting to the following URL (note that incrementing is not idempotent and hence cannot be an HTTP PUT).

POST http://<hostname>:10002/data/Table/<table-name>/<row-key>?op=increment

In the body, you must specify the columns and values that you want to write as a JSON map from strings to long numbers, for example:

```
{"x":1,"y":7}
```

This REST call has the same effect as the corresponding table `Increment` operation (see Section 4.4 Core Datasets - Tables on page 29. If successful, the response contains a JSON map from column key to the incremented values. For example, the existing value of column `x` was 4, and column `y` did not exist, then the response is (column `y` is newly created):

```
{"x":5,"y":7}
```

The expected HTTP return codes are:

- *200 OK*: Everything went well.
- *400 Bad Request:* The JSON string is not well-formed or cannot be parsed as a map from string to long, or one of the existing column values is not an 8-byte long value.
- *404 Not found*: A table with the given name does not exist.

Delete Data from a Table

To delete from a table, you submit an HTTP delete request:

DELETE http://<hostname>:10002/data/Table/<table-name>/<row-key>?columns=<column-key,…>

You must explicitly list the columns that you want to delete. The expected return codes are:

- *200 OK*: Everything went well.
- *400*: No columns were specified.
- *404 Not found*: A table with the given name does not exist.

Encoding of Keys of Values

The URLs and JSON bodies of your REST requests contain row keys, column keys and values, all of which are binary byte arrays in the Java API (see 4.4, Core Datasets - Tables on page 29. Therefore you need a way to encode binary keys and values as strings in the URL and the JSON body. The `encoding` parameter of the URL specifies this encoding. For example, if you append a parameter *encoding=hex* to the request URL, then all keys and values are interpreted as hexadecimal strings, and returned JSON from read and increment requests also has the keys and values encoded that way. But be aware that this applies to all keys and values involved in the request: Suppose you incremented a column in a new table by 42:

POST http://<hostname>:10002/data/Table/counters/a?op=increment

```
{"x":42}
```

Now the value of column `x` is the 8-byte number 42. If you query for the value of this with

GET http://<hostname>:10002/data/Table/counters/a?columns=x

Then the returned JSON will contain a non-printable string for the value of column x:

```
{"x":"\u0000\u0000\u0000\u0000\u0000\u0000\u0000*"}
```

Note the Unicode escapes in the string, and the asterisk at the end (which happens to be the character at code point 42). To make this more legible, you can request for hexadecimal notation, and that will require that you also encode the row key and the column key in your request as hexadecimal:

GET http://<hostname>:10002/data/Table/counters/61?columns=78

The response now contains both the column key and the value as hexadecimal strings.

```
{"78":"000000000000002a"}
```

The supported encodings are:

-	Default. Only ASCII characters are supported and mapped to bytes one-to-one.
encoding=hex	Hexadecimal strings. For example, the ASCII string `a:b` is represented as `613A62`.
encoding=url	URL encoding (also known as %-encoding). URL-safe characters use ASCII-encoding, other bytes values are escaped using a % sign. For example, the hexadecimal value `613A62` is represented as the string `a%3Ab`.
encoding=base64	URL-safe Base-64 encoding without padding. For more information, see Internet RFC 2045. For example, the hexadecimal value `613A62` is represented as the string `YTpi`.

If you specify an encoding that is not supported, or you specify keys or values that cannot be decoded using the that encoding, the request will return HTTP code *400 Bad Request*.

COUNTER VALUES

Your values may frequently be counters, whereas the row and column keys may not be numbers. In such cases it is more convenient to represent values as numeric literals, by specifying *counter=true*, for example:

GET http://<hostname>:10002/data/Table/counters/a?columns=x&counter=true

The response now contains the column key as text and the value as a number literal:

```
{"x":"42"}
```

Note that you can also specify the *counter* parameter in a PUT.

PROCEDURE REST API

This interface supports sending queries to the procedures of your application.

EXECUTING PROCEDURES

Remember that a procedure accepts a method name and a map of string arguments as parameters. To send a query to a procedure, you send the method name as part of the request URI and the arguments as a JSON string in the body of the request. The request is an HTTP post:

POST http://<hostname>:10010/procedure/<app>/<procedure>/<method>

For example, to invoke the `getCount` method of the RetrieveCounts procedure from Section 4.5, you post:

POST http://<hostname>:10010/procedure/WordCount/RetrieveCounts/getCount

```
{"word":"a"}
```

Return codes:

- *200 OK*: Everything went well, and the body contains the query result.
- *400 Bad Request*: The procedure and method exist, but the arguments are not as expected
- *404 Not found*: The procedure or the method does not exist.

MONITOR REST API

The monitor API lets you get the status and metrics about running flows and procedures. To get the status of a running flow, you submit an HTTP GET request:

GET http://<hostname>:10005/monitor/<app>/<flow>/status

The same works for procedures, just use the procedure name instead of the flow name in the URI. To get metrics about the flow, you change the request slightly:

GET http://<hostname>:10005/monitor/<app>/<flow>/metrics

This will return a comma-separated list of metric names with their values. Note that metrics are only generated by a running flow or procedure. If a flow has never run, or a procedure has never received any requests, the list of metrics is empty.

Return codes:

- *200 OK*: Everything went well, and the body contains the requested status or metrics.
- *404 Not found*: The procedure or the flow does not exist.

Reactor Client REST API

Use the Reactor Client REST API to deploy or delete applications and manage the life cycle of flows, procedures and MapReduce jobs.

Deploy

To deploy an application from your local file system, submit an HTTP POST request with the name of the JAR file as a header and its content as the body of the request:

```
POST http://<hostname>:10007/app
X-Archive-Name: HelloWorld.jar
<JAR binary content>
```

Invoke the same command to update an application to a newer version. However, be sure to stop all of its flows, procedures and MapReduce jobs before updating the application.

Delete

To delete an application together with all of its flows, procedures and MapReduce jobs, submit an HTTP DELETE request:

```
DELETE http://<hostname>:10007/app/HelloWorld      //Note: Don't append ".jar"
```

(This does not delete the streams and datasets associated with the application because they belong to your account, not the application.)

Start, Stop, Status

After an application is deployed, you can start and stop its flows, procedures and MapReduce jobs and also query for their status.

```
POST http://<hostname>:10007/app/HelloWorld/flow/WhoFlow?op=start

POST http://<hostname>:10007/app/HelloWorld/flow/WhoFlow?op=stop

GET http://<hostname>:10007/app/HelloWorld/flow/WhoFlow?op=status
```

For a procedure or a MapReduce job, simply change the word *flow* in the examples above to *procedure* or *mapreduce*.

Scale

You can discover or scale the number of flowlets deployed for a given flow. The following examples illustrate these features using the *HelloWorld* app with a flow named *WhoFlow* and a flowlet named *saver*:

Discover the number of flowlets

```
GET http://<hostname>:10007/app/HelloWorld/flow/WhoFlow/saver?op=instances
```

Deploy a specified number of flowlets

```
PUT http://<hostname>:10007/app/HelloWorld/flow/WhoFlow?op=instances

{ "instances" : <int> }
```

5.3. Command Line Tools

The Reactor Development Kit includes a set of tools that allow you to access and manage local and remote Reactor instances from the command line. The list of tools is outlined below. They all support the --help option to get brief usage information.

The examples in the rest of this section assume you are in the bin directory of the Reactor Development Kit (for example, ~/continuuity-reactor-development-kit-1.6.1/bin/)

Reactor

The Reactor shell script can be used to start and stop the Reactor server:

```
> ./continuuity-reactor start
> ./continuuity-reactor stop
```

To get usage information for the tool invoke it with the --help parameter:

```
> ./continuuity-reactor --help
```

Data Client

The data client command line tool can be used to create tables and to read, write, modify, or delete data in tables.

To create a table:

```
> ./data-client create --table myTable
```

To write data to the table, specify the row key, column keys and values on the command line:

```
> ./data-client write --table myTable --row a --column x --value z
> ./data-client write --table myTable --row a --column x --value z --column y --value q
> ./data-client write --table myTable --row a --columns x,y --values z,q
```

To read from a table, you can read the entire row, specific columns, or a column range (compare Section 4.4 Core Datasets - Tables on page 29:

```
> ./data-client read --table myTable --row a
> ./data-client read --table myTable --row a --column x
> ./data-client read --table myTable --row a --columns x,y
> ./data-client read --table myTable --row a --start x
> ./data-client read --table myTable --row a --stop z
> ./data-client read --table myTable --row a --start y --stop z
```

The `read` command prints the columns that it retrieved to the console:

```
> ./data-client read --table myTable --row a
x:z
y:q
```

If you prefer JSON output, you can use the `--json` argument:

```
> ./data-client read --table myTable --row a --json
{"x":"z","y":"q"}
```

To delete from a table, specify the row and the columns to delete on the command line:

```
> ./data-client delete --table myTable --row a –columns=x,y
```

You can also perform atomic increment on cells of a table. The command prints the incremented values on the console:

```
> ./data-client increment --table myTable --row counts --columns x,y --values 1,4
x:1
y:4
> ./data-client increment --table myTable --row counts --columns x,y --values 2,-6
x:3
y:-2
```

Similarly to the REST interface, the command line allows to use an encoding for binary values or keys. If you specify an encoding, then it applies to all keys and values involved in the command. For example, the following command has the same effect as the REST call on in Section 5.2 Encoding of Keys of Values on page 53:

```
> ./data-client read --table counters --row 61 --hex
78:000000000000002a
```

Other supported encodings are URL-safe Base-64 with `--base64` and URL-encoding ("%-escaping") with `--url`. See Section 5.2 Encoding of Keys of Values on page 53 for more details.

For your convenience when representing counter values, you may use the `--counter` option. In this case, the values are interpreted as long numbers and converted to 8 bytes, without affecting the encoding of the other parameters.

To use the data-client with your Sandbox Reactor, you need to provide the host name of your Sandbox Reactor and the API key that authenticates you with it:

```
> ./data-client create --table myTable --host <hostname> --apikey <apikey>
```

If you configured your Local Reactor to use different REST ports (see Section 5.2 on page 47), then you also need to specify the default data REST port on the command-line:

```
> ./data-client create –table myTable --host <hostname> --apikey <apikey> --port 10002
```

Stream Client

The stream client is a utility to send events to a stream or to view the current content of a stream. To send a single event to a stream:

```
> ./stream-client send --stream text --header number "101" --body "message 101"
```

The stream must already exist when you submit this. The send command supports adding multiple headers:

```
> ./stream-client send --stream text --header number "102" --header category "info"
>> --body "message 102"
```

Since the body of an event is binary, it is not always printable text. You can use the `--hex` option to specify body in hexadecimal (the default is URL-encoding). If the body is too long or too inconvenient to specify it on the command line, you can use `--body-file <filename>` as an alternative to `--body` to read it from a binary file.

To inspect the contents of a stream, you can use the `view` command:

```
> ./stream-client view –-stream msgStream --last 50
```

This retrieves and prints the last (that is, the latest) 50 events from the stream. Alternatively, you can use `–first` to see the first (oldest) events, or `--all` to see all events in the stream.

As with the send command, you can use `--hex` to print the body of each event in hexadecimal form. Also, similar to the data client (see the previous section), you can use the `--host`, `--port`, and `--apikey` options to use the stream client with your Sandbox Reactor (the default stream REST port is 10000):

```
> ./stream-client view –-stream text --host <hostname> --apikey <apikey> --port 10000
```

In order to create a stream that does not exist yet, invoke:

```
> ./stream-client create –-stream newStream
```

Reactor Client

Use the Reactor client to deploy or delete applications, manage the life cycle of flows, procedures and MapReduce jobs, and promote Local Reactor applications to your Sandbox Reactor. Unlike the other command line clients, the Reactor client works only with your Local Reactor.

Deploy

To deploy an application to your Local Reactor from your local file system, invoke the deploy command:

```
> ./reactor-client deploy --archive HelloWorld.jar
```

Invoke the deploy command to update an application on the Local Reactor to a newer version. However, before updating an application stop all of its flows, procedures and MapReduce jobs.

Delete

The delete command deletes an application and its flows, procedures and MapReduce jobs from your Local Reactors. (This does not delete the streams and datasets associated with the application because they belong to your account, not the application.)

```
> ./reactor-client delete --application HelloWorld    //Note: Don't append ".jar"
```

Start, Stop, Status, Scale

After the application is deployed to your Local Reactor, you can start or stop its flows, procedures and MapReduce jobs and you can query their status. For a flow, you can also specify the number of instances for a flowlet:

```
> ./reactor-client start --application HelloWorld --flow WhoFlow
> ./reactor-client status --application HelloWorld --flow WhoFlow
> ./reactor-client scale --application HelloWorld --flow WhoFlow --flowlet saver --instances 4
> ./reactor-client stop --application HelloWorld --flow WhoFlow
```

For a procedure or a MapReduce job, simply change the word *flow* in the examples above to *procedure* or *mapreduce* (except for the scale command, which only applies to flows).

Promote to Sandbox Reactor (Push-to-Cloud)

To promote a Local Reactor application to the Sandbox Reactor (Push-to-Cloud) from the command line, invoke the promote command with the hostname of your Sandbox Reactor and the API key that authenticates you:

```
> ./reactor-client promote --application HelloWorld
                           --host mysandbox.continuuity.net --apikey <yourApiKey>
```

6. Next Steps

Thanks for downloading the Continuuity Reactor Development Kit. By now you should be well on your way to building your Big Data applications using the Continuuity Reactor.

Once you have built and tested your application, be sure to push it to your Sandbox Reactor. To get your Sandbox Reactor, go to: https://accounts.continuuity.com/.

7. Technical Support

If you need any help from us along the way, you can reach us at http://support.continuuity.com.

8. Glossary

ACID	Atomicity, Consistency, Isolation, and Durability
DAG	Directed Acyclic Graph