CONTINUUITY REACTOR DEVELOPER GUIDE

Version 2.0.0

Continuuity, Inc.

Table of Contents

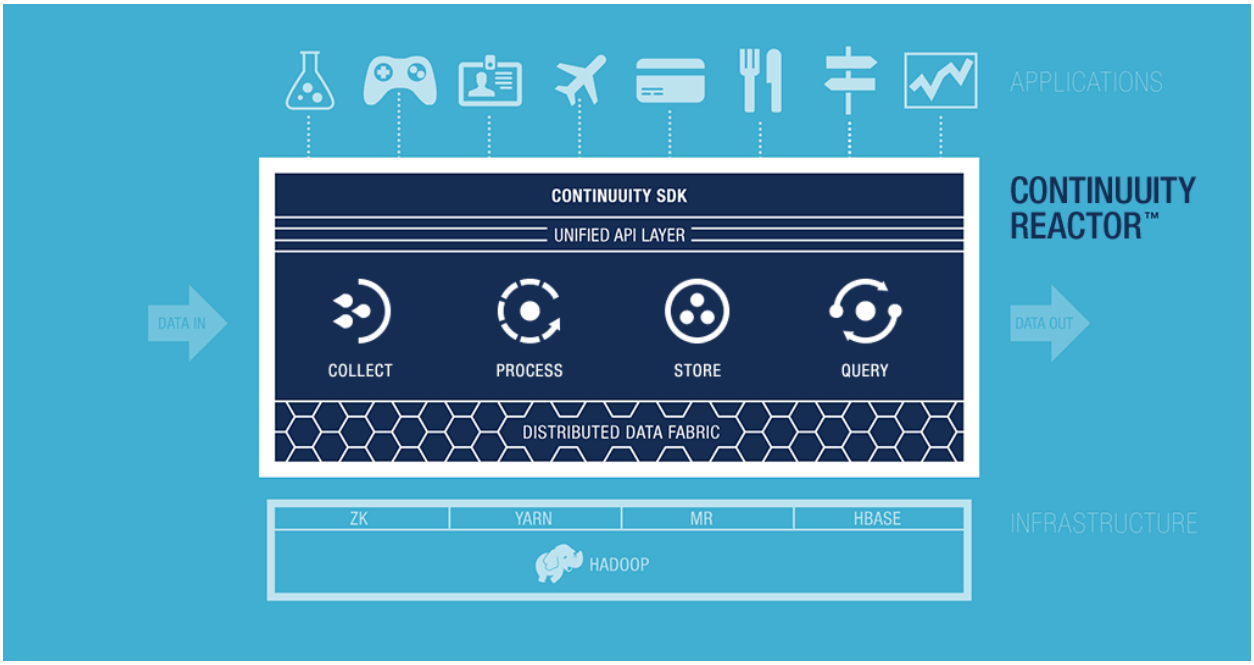
1 Introduction

The Continuuity Reactor™ 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. The Reactor is a scalable and integrated runtime environment and data platform with a rich visual user interface. You can use the Reactor to quickly and easily 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 Reactor™ APIs, libraries, and the Reactor Dashboard. The *Continuuity Reactor Getting Started Guide* will have you running your own instance of the Reactor and deploying a sample Reactor application in minutes. This programming guide deep-dives into the Core Reactor APIs and walks 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 interfaces and shields you from unnecessary complexity. Rather than piecing together different open source frameworks and runtimes to assemble your own Big Data infrastructure stack, the Reactor provides an integrated platform that makes it easy to create the different elements of your Big Data application: collecting, processing, storing, and querying data.



The Reactor is 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 Reactor on your own machine, which makes testing and debugging easy, and then you'll push your app to a free Sandbox Reactor account on 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, documentation, sample applications, and the Local Reactor. The Reactor Development Kit is your on-ramp to the Continuuity Enterprise Reactor, enabling you to develop locally and then push to your Sandbox Reactor with a single click. Your interactions with the Enterprise Reactor are the same as with the Local Reactor and the Sandbox, but you can control the scale of your application to meet production demands.

1.2.1 Requirements

For system requirements, see the *Continuuity Reactor Getting Started Guide*.

1.2.2 What's in the Box?

The Continuuity Reactor Development Kit includes Local Reactor and the Reactor Software Development Kit (SDK) with the Reactor APIs, example code and documentation.

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, and example applications. See the *Continuuity Reactor Getting Started Guide* for more details.

Please keep in mind that Developer Guide only gives a brief overview and a few examples. Typically for the APIs there is a great deal more information in the Javadocs. You can find them in the javadocs directory in your Reactor installation directory.

Local Reactor

The Local 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 laptop or desktop. 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 experiment and 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 Javadocs, a set of example applications that utilize various features of the Reactor, and instructions about how to build, deploy, and run applications using the Reactor. See the *Continuuity Reactor Getting Started Guide* for more information.

Once the first version of your application is ready you can deploy it to your Local Reactor using the Local Reactor Dashboard or the command line tools. 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 interfaces 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 a friendly greeting using one stream, one dataset, one flow (with one flowlet) and one procedure. The next section introduces these concepts more thoroughly.

The HelloWorld 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 HelloWorld! program for the Reactor")
      .withStreams().add(new Stream("who"))
      .withDataSets().add(new KeyValueTable("whom"))
      .withFlows().add(new WhoFlow())
      .withProcedures().add(new Greeting())
      .noMapReduce()
      .noWorkflow()
      .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;

    @ProcessInput
    public void processInput(StreamEvent event) {
      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 request, ProcedureResponder responder)
        throws Exception {
      byte[] name = whom.read(NameSaver.NAME);
      String toGreet = name != null ? new String(name) : "World";
      responder.sendJson("Hello " + toGreet + "!");
    }
  }
}
```

This code is included along with other examples in the Reactor Development Kit. You can find the precompiled application at:

```
continuuity-reactor-development-kit-2.0.0/examples/HelloWorld/HellowWorld.jar
```

To deploy the application, start the Reactor:

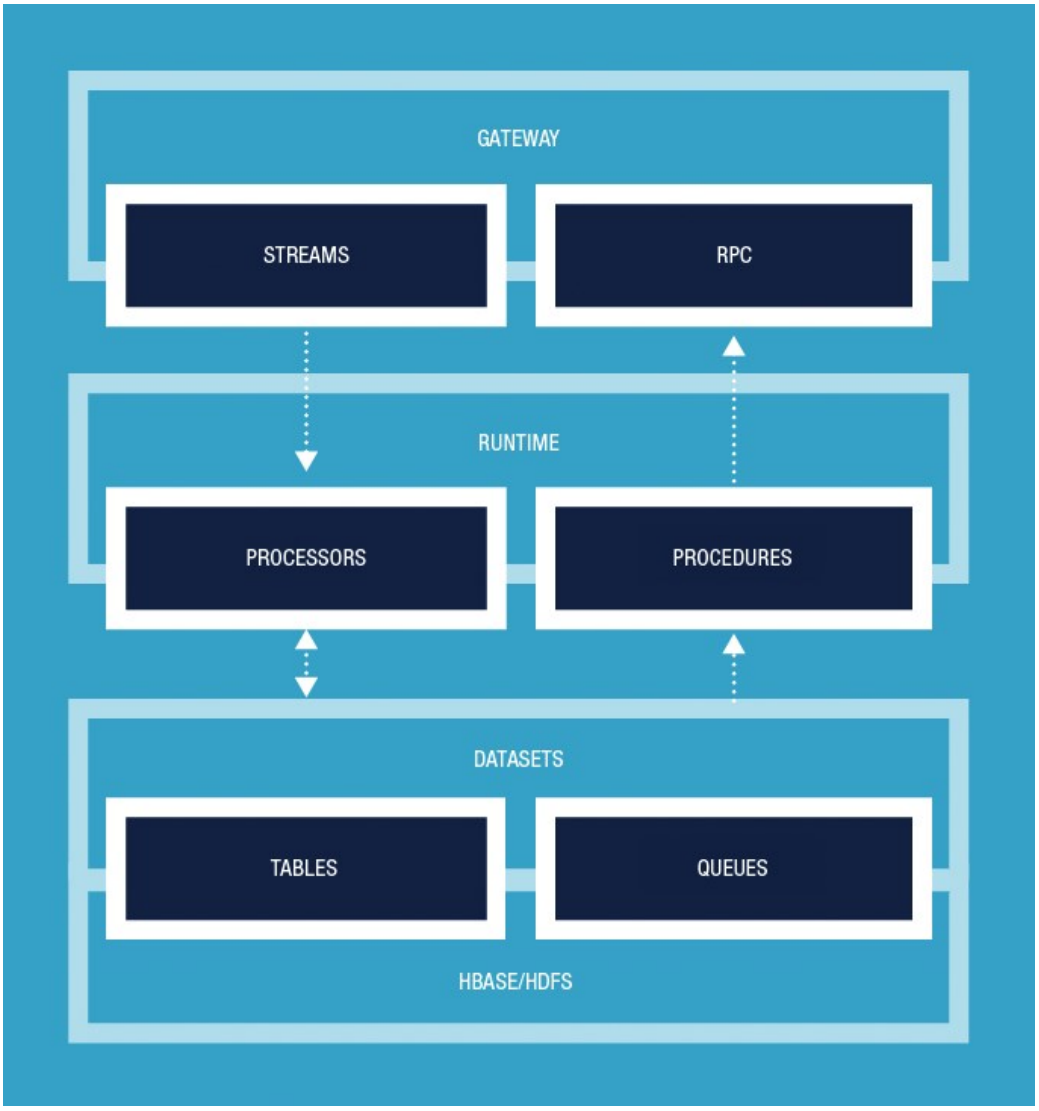
```
> continuuity-reactor start
```

Go to the local Reactor Dashboard at http://localhost:9999/ and drag the HelloWorld.jar onto the Dashboard. The HelloWorld app will appear in the Overview page. Click it to see the stream, flow, flowlet, 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, then click the stream item labeled “who”, enter a name in the text box, and press Enter or click Inject.

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 the method name “greet” into the METHOD box and click **EXECUTE** to see the response. The name you entered into the “who” stream displays on the page.

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 ingested and stored in both structured and unstructured forms, then processed in real-time or in batch, and the results can be made available for retrieval.



The Reactor provides **Streams** for real-time data ingestion from any external system, **Processors** for performing elastically scalable real-time stream or batch 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 through 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 bringing data from external systems into the Reactor in real time. You can write to streams easily using REST or command line tools, 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, the Management Dashboard, or programmatically within your application. Data written to a stream can be consumed by flows and processed in real-time as described below.

3.2 Process with Flows, MapReduce and Workflows

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.

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 flowlet to generate data or pull it from an external source.

MapReduce programs are used to process data in batch. MapReduce jobs can be written the same way as in a conventional Hadoop system. In addition, Reactor datasets can be accessed from MapReduce jobs as input and output.

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

3.3 Store with Datasets

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

3.3.1 Types of Datasets

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

Other datasets are built on top of 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 datasets 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.

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.

3.5 Package with Applications

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

4 Reactor Runtime Editions

The Continuuity Reactor can be run in different modes: in-memory mode for unit testing, Local Reactor for local testing, and Hosted or Enterprise Reactor 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.

4.1.1 In-Memory Reactor

The in-memory Reactor allows you to easily run the Reactor for use in unit 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.

4.1.2 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. All data is persisted.

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

4.1.3 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 or 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 or Enterprise Reactor without charge.

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

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

5 Reactor Programming Guide

This section dives into more detail around each of the different Reactor core elements - Streams, Datasets, Flows, Procedures, MapReduce, and Workflows, 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 and, procedure systems, datasets, transactions, MapReduce, and workflows will give an understanding of how these systems function. Finally an example application is implemented to help illustrate these concepts and describe how an entire application is built.

For more information, see the Javadocs in the javadocs directory and the example code in the examples directory, both of which are in your Reactor installation directory.

5.1 Reactor Core APIs

This section briefly discusses the Reactor core APIs.

5.1.1 Application

An application is a collection of streams, datasets, flows, procedures, MapReduce, and workflows. To create an application, implement the Application interface. This is where you specify the application metadata and declare and configure each application element:

```
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(...) ...
      .withMapReduce()
        .add(...) ...
      .withWorkflows()
        .add(...) ...
      .build();
  }
}
```

You can also specify that an application does not use an element, for example, a stream:

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

and so forth for all of the other constructs.

5.1.2 Stream

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

```
      .withStreams()
        .add(new Stream("myStream")) ...
```

5.1.3 Flow

Flows are composed of connected flowlets wired into a DAG. To create a flow, implement the Flow interface. 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, and 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 Exception {
}

@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 output emitter 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.

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

```
OutputEmitter<Long> output;

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

If you define multiple process methods, a method 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 of the receiving flowlet 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. 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 {
    ...
    @ProcessInput
    public void processEvent(StreamEvent event) {
      ...
    }
```

Flowlet Method @Tick Annotation

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

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

```
public class RandomSource extends AbstractFlowlet {

  private OutputEmitter<Integer> randomOutput;

  private final Random random = new Random();

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

Connection

There are multiple ways to connect the flowlets of a flow. The most common form is to use the flowlet name. Because the name of each flowlet defaults to its class name, 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")
```

5.1.4 PROCEDURE

Procedures receive calls from external systems and perform arbitrary server-side processing on demand.

To create a procedure, implement the Procedure interface, or more conveniently, extend the AbstractProcedure class. A procedure is configured and initialized similarly to a flowlet, but instead of a process method you'll define a handler method. Upon external call, the handler method receives the request and sends a response. The most generic way to send a response is to obtain a Writer and stream out the response as bytes. Make sure to close the writer when you are done:

```
class HelloWorld extends AbstractProcedure {

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

This uses the most generic way to create the response, which allows you to send arbitrary byte content as the response body. In many cases you will actually respond with JSON. Procedures have 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(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;
    }
```

5.1.5 DATASET

Datasets store and retrieve data. If your application uses a dataset, 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"))
      ...
```

To use the dataset in a flowlet or a procedure, instruct the runtime system to inject an instance of the dataset with the @UseDataSet annotation:

```
Class MyFowlet extends AbstractFlowlet {

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

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

You can also implement custom datasets by extending the DataSet base class or by extending existing dataset types.

5.1.6 MAPREDUCE

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

```
    public ApplicationSpecification configure() {
    return ApplicationSpecification.Builder.with()
      ...
      .withMapReduce()
        .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 Application. It defines the name and description of the MapReduce job. You can also specify datasets to be used as input or output for the job.

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

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

The onFinish() method is invoked after the MapReduce job has finished, 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));
    }
  }
}
```

MapReduce and Datasets

Mapper or Reducer can also directly read from a dataset or write to a dataset similar to the way a flowlet or procedure would.

To access dataset directly in Mapper or Reducer, you need:

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

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

- Inject the dataset into the mapper or reducer that uses it:

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

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

5.1.7 WORKFLOW

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

```
public ApplicationSpecification configure() {
  return ApplicationSpecification.Builder.with()
    ...
    .withWorkflows()
      .add(new PurchaseHistoryWorkflow())
```

You must implement the Workflow interface, which requires the configure() method. Use the addSchedule() method to run a workflow job periodically:

```
public static class PurchaseHistoryWorkflow implements Workflow {

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

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

```
public static class PurchaseHistoryWorkflow implements Workflow {

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

5.1.8 User-Defined Metrics

You can embed user-defined metrics in methods. They will emit their metrics and you can retrieve them (and system metrics) via the Metrics Explorer in the Dashboard or via the Reactor's REST interfaces.

Adding metrics to the HelloWorld example can be done as follows:

```
public static class NameSaver extends AbstractFlowlet {

  static final byte[] NAME = { 'n', 'a', 'm', 'e' };

  @UseDataSet("whom")
  KeyValueTable whom;
  Metrics flowletMetrics; // Declare custom metrics

  @ProcessInput
  public void processInput(StreamEvent event) {
    byte[] name = Bytes.toBytes(event.getBody());
    if (name != null && name.length > 0) {
      whom.write(NAME, name);
    }
    if (name.length > 10) {
      flowletMetrics.count("names.longnames", 1);
    }
    flowletMetrics.count("names.bytes", name.length);
  }
}
```

For more information, see the Metrics section on page 60.

5.1.9 Logging

The Reactor supports logging through standard SLF4J APIs. For instance, in a flowlet you can write:

```
private static Logger LOG = LoggerFactory.getLogger(WordCounter.class);
...
@ProcessInput
public void process(String line) {
  LOG.info(this.getContext().getName() + ": Received line " + line);
  ... // processing
  LOG.info(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 the Logs button in the Flow or Procedure screens.
- Using REST interface describe in section 6.2

5.1.10 Runtime Arguments

Flows, procedures, MapReduce and workflows can receive runtime arguments:

- For flows and procedures, runtime arguments are available to the initialize method in the context.
- For MapReduce, runtime arguments are available to the beforeSubmit and onFinish methods in the context. The beforeSubmit method can pass them to the mappers and reducers through the job config.
- When a workflow receives runtime arguments it passes them to each MapReduce in the workflow.

The initialize() method in this example accepts a runtime argument for the HelloWorld procedure. For example, we can change the greeting from “Hello”, to “Good Morning” by passing a runtime argument:

```
public static class Greeting extends AbstractProcedure {

  @UseDataSet("whom")
  KeyValueTable whom;
  private String greeting;

  public void initialize(ProcedureContext context) {
    Map<String, String> args = context.getRuntimeArguments();
    greeting = args.get("greeting");
    if (greeting == null) {
      greeting = "Hello";
    }
  }

  @Handle("greet")
  public void greet(ProcedureRequest request, ProcedureResponder responder) throws
      Exception {
    byte[] name = whom.read(NameSaver.NAME);
    String toGreet = name != null ? new String(name) : "World";
    responder.sendJson(greeting + " " + toGreet + "!");
  }
}
```

5.1.11 Best Practices

Initialization

There are three ways to initialize a non-static member variables used in Datasets, Flowlets and Procedures:

- Using the default constructor
- Using the initialize method of the Datasets, Flowlets and Procedures.

- Using @Property annotations

5.2 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 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 Atomicity, Consistency, Isolation, and Durability (ACID) properties, and helps assure a unique and core property of the flow system: it guarantees atomic and exactly-once-processing of each input object by each flowlet in the DAG.

5.2.1 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)
@ProcessInput
public void process(Iterator<String> words) {
  ...
```

For the batch example above, up to 100 data objects can be read from the input and processed at one time.

5.2.2 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 via the REST interfaces or via the Dashboard. This enables you to scale your application to meet capacity at runtime.

In the Local Reactor, multiple flowlet instances are run in threads, so in some cases actual performance may not be affected. However, in the Hosted and Enterprise Reactors each flowlet instance runs in its own Java Virtual Machine (JVM) with independent compute resources, so scaling the number of flowlets can improve performance, and depending on your implementation this can have a big impact.

5.2.3 Partitioning Strategies in Flowlets

As mentioned earlier, if you have multiple instances of a flowlet the input queue is partitioned among the flowlets. 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*: With this strategy, the number of items is distributed evenly among the instances. Round robin is, in general, the most efficient partitioning. Though more efficient than FIFO, it is not always ideal, such as in cases where the application needs to group objects into buckets according to business logic. In these cases, hash-based partitioning is preferable.
- *Hash-based*: 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.

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) {
    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) {
    this.wordCountsTable.increment(Bytes.toBytes(word), 1L);
  }
```

Now, if we have three 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) {
  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 the partitioning key:

```
@Output("wordOut")
private OutputEmitter<String> wordOutput;
...
public void process(StreamEvent event) {
  ...
  // emit the word with the partitioning 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 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) {
  ...
```

5.2.4 Getting Data In

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

- 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.
- A **Flowlet** method with an @Tick annotation can be used to actively generate data or retrieve it from an external data source. For instance, it can pull data from the Twitter firehose.

5.2.5 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 data 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 data 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.

5.2.6 Disabling Transaction in Flows

Transaction can be disabled for a Flow by annotating the flow class with the @DisableTransaction annotation. While this may speed up performance, if a flowlet fails, for example, the system would not be able to roll back to its previous state.

```
@DisableTransaction
class MyExampleFlow implements Flow {
  ...
}
```

5.2.7 Transactions in MapReduce

When you run a MapReduce that interacts with datasets, the system creates a long-running transaction. 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 transaction 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. It is not efficient to fail the long running job based on a single conflict. Because of this, it is not recommended to write to the same datasets from both real-time and MapReduce programs. It is better to use different datasets, or at least ensure that the real-time processing writes to a disjoint set of columns.

Important to note that MapReduce framework will reattempt a task (mapper or reducer) if it fails. If the task is writing to a dataset, 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.

5.3 The Dataset System

Datasets are your interface to the data. Instead of having to manipulate data with low-level 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.

5.3.1 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 and one holding the index. We distinguish three categories of datasets: core, system, and custom datasets:

- The **core** dataset of the Reactor is a **Table**. Its implementation is hidden from developers and it may use private Dataset 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 manipulate data 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.

Each dataset instance has exactly one dataset class to manipulate it - 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 of its metadata, including the specifications of its underlying datasets. This creates the dataset - if it does not exist yet - and stores 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.

5.3.2 Core Datasets - Tables

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

- Tables have no fixed schema. Unlike relational database 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 the 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.

TABLE API

The table API provides basic methods to perform read, write and delete operations, plus a special atomic increment and compare-and-swap operations:

```
// Read
public Row get(Get get)
public Row get(byte[] row)
public byte[] get(byte[] row, byte[] column)
public Row get(byte[] row, byte[][] columns)
public Row get(byte[] row, byte[] startColumn, byte[] stopColumn, int limit)

// Scan
public Scanner scan(byte[] startRow, byte[] stopRow)

// Write
public void put(Put put)
public void put(byte[] row, byte[] column, byte[] value)
public void put(byte[] row, byte[][] columns, byte[][] values)

// Increment
public Row increment(Increment increment)
public long increment(byte[] row, byte[] column, long amount)
public Row increment(byte[] row, byte[][] columns, long[] amounts)

// Compare And Swap
public boolean compareAndSwap(byte[] row, byte[] column,
                              byte[] expectedValue, byte[] newValue)

// Delete
public void delete(Delete delete)
public void delete(byte[] row)
public void delete(byte[] row, byte[] column)
public void delete(byte[] row, byte[][] columns)
```

Every basic operation has a method that takes operation type object as a parameter and also handy methods for working with byte arrays directly. If your application code already deals with byte arrays you can use the latter ones to save on conversion. Otherwise methods with parameters of specialized type could be more convenient to use as they provide reach API to work with different types.

Read

A get operation reads all columns or selection of columns of a single row:

```
Table t;
byte[] rowKey1;
byte[] columnX;
byte[] columnY;

// read all columns of a row
Row row = t.get(new Get("rowKey1"));

// read specified columns from the row
Row rowSelection = t.get(new Get("rowKey1").add("column1").add("column2"));

// reads a column range from x to y, with a limit of n return values
rowSelection = t.get(rowKey1, columnX, columnY);

// read only one column in one row
byte[] value = t.get(rowKey1, columnX);
```

The Row object provides access to the Row data including its columns. If only a selection of a row columns is requested, the returned Row object will contain only these columns. Row object provides rich API for accessing returned column values:

```
// get column value as byte array
byte[] value = row.get("column1");

// get column value of specific type
String valueAsString = row.getString("column1");
Integer valueAsInteger = row.getInt("column1");
```

When requested, value of a column is converted to specific type automatically. If column is absent in a Row, the returned value is null. To return primitive type correspondent methods accept default value to be returned when column is absent:

```
// get column value of primitive type or 0 if column is absent
long valueAsLong = row.getLong("column1", 0);
```

Scan

A scan operation fetches a subset of rows or all rows of a table:

```
byte[] startRow;
byte[] stopRow;
Row row;

// Scan all rows from startRow (inclusive) to stopRow (exclusive)
Scanner scanner = t.scan(startRow, stopRow);
try {
  while ((row = scanner.next()) != null) {
    LOG.info("column1: " + row.getString("column1"));
  }
} finally {
  scanner.close();
}
```

To scan set of rows not bounded by startRow and/or stopRow you can pass null as their value:

```
byte[] startRow;
// scan all rows of a table
Scanner allRows = t.scan(null, null);
// scan all columns up to stopRow (exclusive)
Scanner headRows = t.scan(null, stopRow);
// scan all columns starting from startRow (inclusive)
Scanner tailRows = t.scan(startRow, null);
```

Write

A put operation writes data into a row:

```
// write set of columns with their values
t.put(new Put("rowKey1").add("column1", "value1").add("column2", 55L));
```

Compare and 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. The operation returns true if it succeeds and false otherwise.

```
byte[] expectedCurrentValue;
byte[] newValue;
if (!t.compareAndSwap(rowKey1, columnX, expectedCurrentValue, newValue)) {
  LOG.info("Current value was different from expected");
}
```

INCREMENT

An increment operation increments a long value of one or more columns. If a column doesn't exist it assumes the value before the increment was 0.

```
// write long value to a column of a row
t.put(new Put("rowKey1").add("column1", 55L));
// increment values of several columns in a row
t.increment(new Increment("rowKey1").add("column1", 1L).add("column2", 23L));
```

If the existing value of the column cannot be converted to long, a NumberFormatException will be thrown.

DELETE

A delete operation removes a whole row or subset of its columns.

```
// delete the whole row
t.delete(new Delete("rowKey1"));

// delete a set of columns from the row
t.delete(new Delete("rowKey1").add("column1").add("column2"));
```

Note that specifying a set of columns helps to perform delete operation faster. Thus, when you know all columns of a row you want to delete, passing them will make deletion faster.

5.3.3 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 of 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 Javadocs for of these classes to learn more about these datasets.

5.3.4 Custom Datasets

You can define your own dataset classes to implement common data patterns specific to your code. For example, suppose you 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 Table entryCountTable;
```

Custom datasets can also optionally implement configure() and initialize() methods. The configure() method returns a specification which we can use to save metadata about the dataset (such as configuration params). The initialize() method is called at execution time. It should be noted that any operations on the data of this dataset are prohibited in initialize().

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

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

The dataset stores a counter for each word in its own row of the word count table. For every word the counter is incremented. If the result of increment 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 entry) {
    long newCount = entryCountTable.increment(Bytes.toBytes(entry), ENTRY_COUNT, 1L);
    if (newCount == 1L) {
      uniqueCountTable.increment(UNIQUE_COUNT, UNIQUE_COUNT, 1L);
    }
  }
```

Finally, we write a method to retrieve the number of unique words seen.

```
  public Long readUniqueCount() {
    return uniqueCountTable.get(new Get(UNIQUE_COUNT, UNIQUE_COUNT))
                           .getLong(UNIQUE_COUNT, 0);
  }
```

5.3.5 MapReduce integration

A MapReduce job can interact with a dataset by using it as an input or an output. The dataset should implement specific interfaces to support this.

When you run a MapReduce job, you can configure it to read its input from a dataset. The destination dataset must implement the BatchReadable interface, which requires two methods:

```
public interface BatchReadable<KEY, VALUE> {
  List<Split> getSplits();
  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);
}
```

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.

5.4 The WordCount Application

This section discusses the entire WordCount application.

5.4.1 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 Application")
      .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())
      .noMapReduce()
      .noWorkflow()
      .build();
  }
}
```

5.4.2 Defining The Flow

The flow must define a configure() method that wires up the flowlets with their connections. Note that here we don't need to 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.

5.4.3 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-alphabetical characters from the words. It counts the number of words and computes the aggregate length of all words, then writes both of these values as increments to the wordStats table that keeps track of the total word and character counts. 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 consumption by 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;

  @ProcessInput
  public void process(StreamEvent event) {
    // 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;
    long 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.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) {

    // 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. 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) {
    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) {
    // 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!

5.4.4 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.

The other custom dataset is the AssociationTable. 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 with all datasets, it must define two constructors and a configure() method

```
public class AssociationTable extends DataSet {

  private Table table;

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

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) {
    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 database 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

supports incrementing multiple columns in a single operation, and we can make use of that to optimize this method:

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

  // 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's already implemented):

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

  // Retrieve all columns of the word's row
  Row result = this.table.get(Bytes.toBytes(word));
  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();
}
```

5.4.5 Implementing a Procedure

To read the data written by the flow, we implement a procedure that binds 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 by 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
    Row result = this.wordStatsTable.get(new Get("totals", "total_length",
                                                 "total_words"));
    if (!result.isEmpty()) {
      // Extract the total sum of lengths
      long totalLength = result.getLong("total_length", 0);
      // Extract the total count of words
      totalWords = result.getLong("totalWords", 0);
      // 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(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(results);
}
```

This concludes the WordCount example. You can find it in the /Reactor-install-dir/examples directory.

5.5 Testing Your Applications

The Reactor comes with a convenient way to unit test your applications. The base for these tests is ReactorTestBase, 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 file of the WordCount 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 ReactorTestBase class. Let's write a test case for the WordCount example:

```
public class WordCountTest extends ReactorTestBase {

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

The first thing we do in this test is deploy the application, then we'll start 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 by obtaining a client for the procedure, and then submitting a query for the global statistics:

```
    // 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 string 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 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"));
  }
```

6 API AND TOOL REFERENCE

6.1 JAVA APIS

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

./continuuity-reactor-development-kit-2.0.0/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.

6.2 REST APIS

The Continuuity Reactor has an HTTP interface for the following purposes:

1. **Stream**: send data events to a stream, or to inspect the contents of a stream.
2. **Data:** interact with datasets (currently only Tables).
3. **Procedure**: send queries to a procedure.
4. **Reactor:** deploy and manage applications.
5. **Logs:** retrieve application logs.
6. **Metrics:** retrieve metrics for system and application metrics (user-defined metrics).

The HTTP interface binds to port 10000. This port cannot be changed.

Common return codes for all HTTP 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 that is not supported by this API

Note: These return codes may not be included in the descriptions below, but any request may return them.

When you interact with a Sandbox Reactor, all HTTP 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: <APIKey>*

6.2.1 Stream HTTP 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:

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

The <new-stream-id> should only contain ASCII letters, digits and hyphens. If the stream already exists, no error is returned, and the existing stream remains in place. The request returns *200 OK* if successful.

Sending Events

A request to send an event to a stream is an HTTP POST:

POST /v2/streams/<stream-id>

where *<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*:

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

After receiving the request, the HTTP handler transforms 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, the 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 POST to the URL:

POST /v2/streams/<stream-id>/consumer-id

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:

- *200 OK*: 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 the same way that a flow reads events. That is, the read will always return the event from the stream that was inserted first and has not been read yet (FIFO semantics). For example, 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 POST to the URI:

POST /v2/streams/<stream-id>/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 for each header of the stream event, analogous to how you send headers when posting an event to the stream:

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

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 Stream Client section on page 66.

6.2.2 Data HTTP API

The data API allows you to interact with tables (the core datasets) through HTTP. You can create tables and read, write, modify, or delete data. For datasets other than tables, you can truncate the dataset through this API.

Create a Table

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

PUT /v2/tables/<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 URI:

PUT /v2/tables/<table-name>/rows/<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.

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 /v2/tables/<table-name>/rows/<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 example, 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. For example:

GET ... /rows/<row-key>?columns=x,y	returns only columns x and y.
GET ... / rows/<row-key>?start=c5	returns all columns greater or equal to c5.
GET ... / rows/<row-key>?stop=c5	returns all columns less than (exclusive) c5.
GET .../rows/<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 row's URL:

POST /v2/tables/<table-name>/rows/<row-key>/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 HTTP call has the same effect as the corresponding table Increment operation. 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, submit an HTTP delete request:

DELETE /v2/tables/<table-name>/rows/<row-key>

Similarly to reading from a table, explicitly list the columns that you want to delete by adding a parameter of the form *?columns=<column-key,…>*. The expected return codes are:

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

Deleting all Data from a Dataset

To clear a dataset from all data, submit an HTTP POST request:

POST /v2/datasets/<dataset-name>/truncate

Note that this works not only for tables, but also for other datasets, including user-defined datasets. The expected return codes are:

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

Encoding of Keys and Values

The URLs and JSON bodies of your HTTP requests contain row keys, column keys and values, all of which are binary byte arrays in the Java API (see the Core Datasets - Tables section on page 33). Therefore you need a way to encode binary keys and values as strings in the URL and the JSON body (the exception is increment, which always interprets values as long integers). 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 requests also has the keys and values encoded that way. Be aware that the encoding applies to all keys and values involved in the request. Suppose you incremented a column in a new table by 42:

POST /v2/tables/counters/rows/a/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 /v2/tables/counters/rows/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 hexadecimal notation, and that will require that you also encode the row key and the column key in your request as hexadecimal:

GET /v2/tables/counters/rows/6?columns=78&encoding=hex

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 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 /v2/tables/counters/rows/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 when writing to a table. This allows you to specify values as numeric strings while using a different encoding for row and column keys.

6.2.3 Procedure HTTP API

This interface supports sending queries to an application's procedures.

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, send the method name as part of the request URL and the arguments as a JSON string in the body of the request. The request is an HTTP POST:

```
POST /v2/apps/<app-id>/procedures/<procedure-id>/methods/<method-id>
```

For example, to invoke the getCount() method of the RetrieveCounts procedure, send a POST request:

```
POST /v2/apps/WordCount/procedures/RetrieveCounts/methods/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.

6.2.4 Reactor Client HTTP API

Use the Reactor Client HTTP 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 /v2/apps
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.

To list all deployed applications, issue an HTTP GET request:

GET /v2/apps

This will return a JSON response that lists each application with its name and description.

DELETE

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

DELETE /v2/apps/HelloWorld

Note that the *HelloWorld* in this URL is the name of the application as configured by the application specification, and not necessarily the same as the name of the jar file that was used to deploy the app. Note also that this does not delete the streams and datasets associated with the application because they belong to your account, not the application.

START, STOP, STATUS, AND RUNTIME ARGUMENTS

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

POST /v2/apps/HelloWorld/flows/WhoFlow/start

POST /v2/apps/HelloWorld/flows/WhoFlow/stop

GET /v2/apps/HelloWorld/flows/WhoFlow/status

For a procedure, MapReduce job, or workflow, simply change the word *flows* in the examples above to *procedures, mapreduce, or workflows*.

When starting a program, you can optionally specify runtime arguments as a JSON map in the request body:

POST /v2/apps/HelloWorld/flows/WhoFlow/start
{"foo":"bar","this":"that"}

The Reactor will use these these runtime arguments only for this invocation of the program.

If you want to save the runtime arguments so that the Reactor will use them every time you start the flow, procedure, MapReduce program, or workflow, you can issue an HTTP PUT as follows:

PUT /v2/apps/HelloWorld/flows/WhoFlow/runtimeargs
{"foo":"bar","this":"that"}

To find out what runtime arguments are saved for a program, issue an HTTP GET request to the same URL:

GET /v2/apps/HelloWorld/flows/WhoFlow/runtimeargs

This will return the saved runtime arguments in the same JSON format.

Scale

You can query or set the number of instances executing a given flowlet. The following examples illustrate these features using the *HelloWorld* app with a flow named *WhoFlow* and a flowlet named *saver*. To find out how many instances of this flowlet are currently running, issue an HTTP GET request:

```
GET /v2/apps/HelloWorld/flows/WhoFlow/flowlets/saver/instances
```

To change the number of instances, send a PUT request to the same URL:

```
PUT /v2/apps/HelloWorld/flows/WhoFlow/flowlets/saver/instances
{ "instances" : 2 }
```

In a similar way, you can query or change the number of instances of a procedure using the URL:

```
PUT /v2/apps/<app-id>/procedures/<procedure-id>/instances
```

Run History and Schedule

To see the history of all runs of a program, you can issue an HTTP GET to the programs' URL. This will return a JSON list of all completed runs, each with a start and end time and the termination status:

```
GET /v2/apps/HelloWorld/flows/WhoFlow/history
[{"runid":"…","start":1382567447,"end":1382567492,"status":"STOPPED"},
 {"runid":"…","start":1382567383,"end":1382567397,"status":"STOPPED"}]
```

The runid field is a UUID that uniquely identifies this run within the Reactor, and the start and end times are in seconds since the epoch.

For workflows, you can also retrieve the schedules defined for that workflow as well as the next time that the workflow is scheduled to run:

```
GET  /v2/apps/PurchaseHistory/workflows/PurchaseHistoryWorkflow/schedules

GET /v2/apps/PurchaseHistory/workflows/PurchaseHistoryWorkflow/nextruntime
```

Promote

To promote an application from your local Reactor to your Sandbox, send a POST request with the host name of your Sandbox in the request body. You must also include the API key for the Sandbox in the request:

```
POST /v2/apps/HelloWorld/promote
X-Continuuity-ApiKey: <APIKey>
{"hostname":"mysandbox.continuuity.net"}
```

6.2.5 Logs

You can download the logs that are emitted by any of the programs running in the Reactor. To do that, you send an HTTP GET request:

GET /v2/apps/<app id>/<prog type>/<prog id>/logs?start=<ts>&end=<ts>

Where the program type is one of flows, mapreduce, procedures, or workflows, and the start and end time are given as seconds since the epoch.

For example:

GET /v2/apps/CountTokens/flows/CountTokens/logs?start=1382576400&end=1382576700

This returns the logs of the CountTokens flow between the times of 6:00pm to 6:05pm on Oct 23, 2013. The output is formatted as HTML-embeddable text, that is, characters that have a special meaning in HTML will be escaped. For example, a line of the log may look like this:

2013-10-23 18:03:09,793 - INFO [FlowletProcessDriver-source-0-executor:c.c.e.c.StreamSource@-1] – source: Emitting line: this is a & character

Note how the context of the log line shows name of the flowlet (“source”) and its instance number (0) as well as the original line in the application code. Note also that the character “&” is escaped as & - if you don’t desire this escaping, you can turn it off by adding the parameter &escape=false to the request URL.

6.2.6 Metrics

As applications process data, the Reactor collects metrics about the application’s behavior and performance. Some of these metrics are the same for every application, for example, how many events are processed, or how many data operations are performed, and are therefore called system or Reactor metrics.

Other metrics are user-defined and can differ from application to application (see the User-Defined Metrics section on page 25). Both types of metrics are made available through the HTTP interface. For example, to retrieve the number of input data objects (“events”) processed by a flowlet over the last 5 seconds, you can issue an HTTP request:

GET /v2/metrics/reactor/apps/CountRandom/flows/CountRandom/flowlets/splitter/process.events?start=now-5s&count=5

This returns a JSON response that has one entry for every second in the requested time interval, but it will only have values for the seconds where the metric was actually emitted (pretty-printed, unlike the actual response):

HTTP/1.1 200 OK
Content-Type: application/json
{"start":1382637108,"end":1382637112,"data":[

{"time":1382637108,"value":6868},
{"time":1382637109,"value":6895},
{"time":1382637110,"value":6856},
{"time":1382637111,"value":6816},
{"time":1382637112,"value":6765}]}

What if you want to know the number of input objects processed across all flowlets of the flow? You simply address the metrics API at the flow context, as in:

GET /v2/metrics/reactor/apps/CountRandom/flows/
CountRandom/process.events?start=now-5s&count=5

Similarly, you can address the context of all flows of an application, an entire application, or the entire Reactor:

GET /v2/metrics/reactor/apps/CountRandom/flows/process.events?start=now-5s&count=5
GET /v2/metrics/reactor/apps/CountRandom/process.events?start=now-5s&count=5
GET /v2/metrics/reactor/process.events?start=now-5s&count=5

To request user-defined metrics instead of system metrics, replace “reactor” in the URL with “user” and specify the user-defined metric at the end of the request.

The time range of a query can be specified in various ways:

start=now-30s&end=now	The last 30 seconds. The begin time is given in seconds, relative to the current time. You can apply simple math using s for seconds, m for minutes, h for hours and d for days. For example now-5d-12h is 5 days and 12 hours ago.
start=1385625600&end=1385629200	From midnight 11/28/2013 to 1:00 A.M. the same day, both given as since the epoch.
start =1385625600&count=3600	The same as before, with the count given as a number of seconds.

Instead of getting the values for each second of a time range, you can also retrieve the aggregate of the metric over time. The following request will return the total number input objects processed since the application was deployed, assuming that the Reactor has not been stopped or restarted (you cannot specify a time range for aggregates):

GET /v2/metrics/reactor/apps/CountRandom/process.events?aggregate=true

If you would like to retrieve multiple metrics at once, you issue an HTTP post instead of a get, with a JSON list as the request body that enumerates the name and attributes for each metrics. For example:

```
POST /v2/metrics
Content-Type: application/json
[ "/reactor/collect.events?aggregate=true",
  "/reactor/apps/HelloWorld/process.events?start=1380323712&count=6000" ]
```

Available Contexts

The general form of a metrics request is:

```
GET /v2/metrics/<scope><context>/<metric>?<parameters>
```

Example for a System Metric:

```
GET /v2/metrics/reactor/apps/HelloWorld/flows/WhoFlow/flowlets/
                                        saver/process.bytes?aggregate=true
```

Example for a User-Defined Metric:

```
GET /v2/metrics/user/apps/HelloWorld/flows/WhoFlow/flowlets/
                                        saver/names.bytes?aggregate=true
```

The scope must be either *reactor* for system metrics, or *user* for user-defined metrics. System metrics are either application metrics, that is, about applications and their flows, procedures, MapReduce and workflows, or they are data metrics relating to streams or datasets. User metrics are always in the application context. The context of a metric is typically enclosed into a hierarchy of contexts, for example the flowlet context is enclosed in the flow context, which in turn is enclosed in the application context. A metric can always be queried (and aggregated) relative to any enclosing context. These are the available application contexts of the Reactor:

/apps/<app-id>/flows/<flow-id>/flowlets/<flowlet-id>	One flowlet of a flow.
/apps/<app-id>/flows/<flow-id>	All flowlets of a flow.
/apps/<app-id>/flows	All flowlets of all flows of an app.
/apps/<app-id>/procedures/<procedure-id>	One procedure.
/apps/<app-id>/procedures	All procedures of an application.
/apps/<app-id>/mapreduce/<mapreduce-id>/mappers	All mappers of a MapReduce.
/apps/<app-id>/mapreduce/<mapreduce-id>/reducers	All reducers of a MapReduce.
/apps/<app-id>/mapreduce/<mapreduce-id>	One MapReduce.
/apps/<app-id>/mapreduce	All MapReduce of an application.

/apps/<app-id>	All programs of an application.
/	All programs of all applications.

Stream metrics are only available at the stream level and the only available context is:

/streams/<stream-id>	A single stream

Dataset metrics are available at the dataset level, but they can also be queried down to the flowlet, procedure, mapper, or reducer level:

/datasets/<dataset-id>/apps/<app-id>/ flows/<flow-id>/flowlets/<flowlet-id>	A single dataset in the context of a single flowlet
/datasets/<dataset-id>/apps/<app-id>/ flows/<flow-id>	A single dataset in the context of a single flow
/datasets/<dataset-id><any application context>	A single dataset in that context
/datasets/<dataset-id>	A single dataset across all apps
/	All datasets across all apps

Available Metrics

For reactor metrics, the available metrics depend on the context. User-defined metrics will be available at whatever context that they are emitted from.

The following metrics are available at the flowlet context:

process.busyness	A number from 0 to 100 indicating how "busy" flowlet is. Note that you cannot aggregate over this metric.
process.errors	Number of errors while processing.
process.events.processed	Number of events/data objects processed.
process.events.in	Number of events read in by the flowlet.
process.events.out	Number of events emitted by the flowlet.
store.bytes	Number of bytes written to datasets.
store.ops	Operations (writes and read) performed on datasets.
store.reads	Read operations performed on datasets.
store.writes	Write operations performed on datasets.

The following metrics are available at the mappers and reducers context:

process.completion	A number from 0 to 100 indicating progress of the map or reduce phase.
process.entries.in	Number of entries read in by the map or reduce phase.
process.entries.out	Number of entries written out by the map or reduce phase.

The following metrics are available at the procedures context:

query.requests	Number of requests made to the procedure.
query.failures	Number of failures seen by the procedure.

The following metrics are available at the streams context:

collect.events	Number of events collected by the stream.
collect.bytes	Number of bytes collected by the stream.

The following metrics are available at the datasets context:

store.bytes	Number of bytes written.
store.ops	Operations (writes and read) performed.
store.reads	Read operations performed.
store.writes	Write operations performed.

6.3 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-2.0.0/bin/).

6.3.1 Reactor

The Reactor shell script can be used to start, restart, and stop the Reactor server and check its status:

```
$ ./continuuity-reactor start
$ ./continuuity-reactor restart
$ ./continuuity-reactor stop
$ ./continuuity-reactor status
```

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

```
$ ./continuuity-reactor --help
```

6.3.2 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 –host localhost --port 10000
```

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 –host localhost
                      --port 10000

$ ./data-client write --table myTable --row a --column x --value z --column y --value q
                     –host localhost --port 10000

$ ./data-client write --table myTable --row a --columns x,y --values z,q
                     –host localhost --port 10000
```

To read from a table, you can read the entire row, specific columns, or a column range:

```
$ ./data-client read --table myTable --row a –host localhost --port 10000

$ ./data-client read --table myTable --row a --column x –host localhost --port 10000

$ ./data-client read --table myTable --row a --columns x,y –host localhost
                     --port 10000

$ ./data-client read --table myTable --row a --start x –host localhost --port 10000

$ ./data-client read --table myTable --row a --stop z –host localhost --port 10000

$ ./data-client read --table myTable --row a --start y --stop z –host localhost
                     --port 10000
```

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

```
$ ./data-client read --table myTable --row a –host localhost --port 10000
x:z
y:q
```

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

```
$ ./data-client read --table myTable --row a --json –host localhost –port 10000
{"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 –host localhost
                       –port 10000
```

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
                          --host localhost –-port 10000
x:1
y:4
$ ./data-client increment --table myTable --row counts --columns x,y --values 2,-6
                          --host localhost –-port 10000
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.

```
$ ./data-client read --table counters --row 61 --hex --host localhost
                     --port 10000
78:000000000000002a
```

Other supported encodings are URL-safe Base-64 with --base64 and URL-encoding ("%-escaping") with --url.

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> --port 10000
```

If you configured your Local Reactor to use different REST ports 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 10000
```

6.3.3 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"
                       --host localhost --port 10000
```

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"
                       --host localhost --port 10000 --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 on the command line, you can use --body-file <filename> as an alternative to --body to read from a binary file.

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

```
$ ./stream-client view –-stream msgStream --last 50 --host localhost --port 10000
```

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 –-host <hostname> --port 10000
```

7 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/.

8 Technical Support

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