CONTINUUITY APPFABRIC DEVELOPER GUIDE

Version 0.3.0, Released February 1, 2013

TABLE OF CONTENTS

1. Introduction

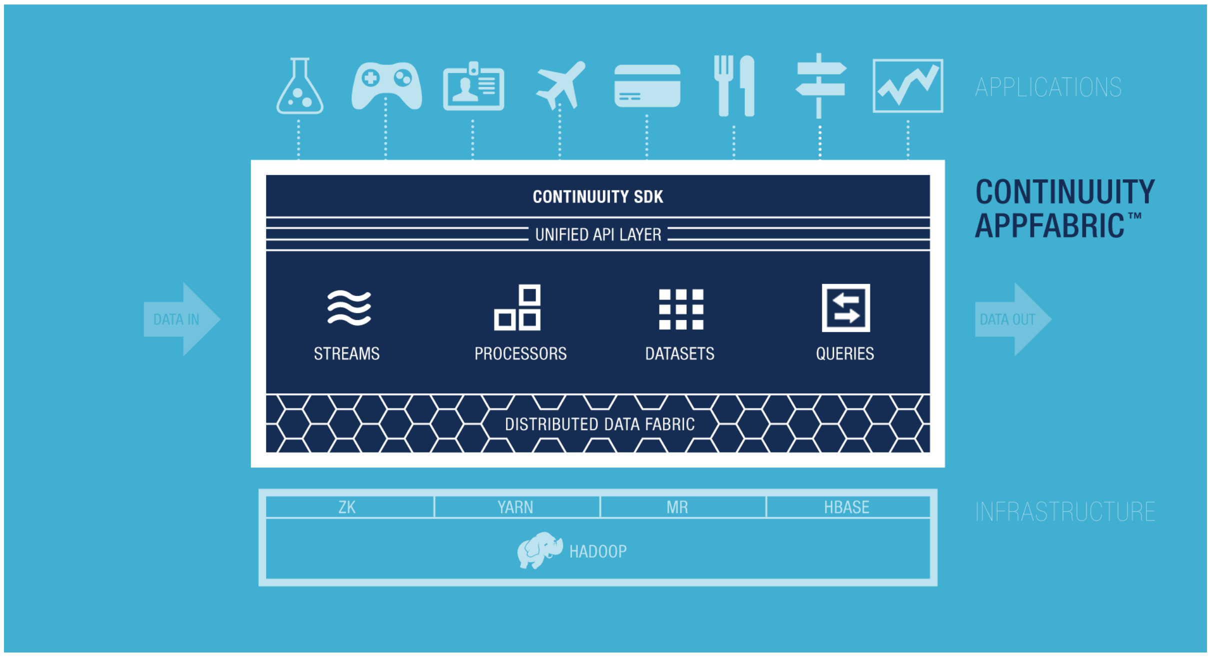
Developing, testing, running and scaling Big Data applications can be a difficult and complicated process requiring specialized expertise and large teams of engineers and operators. It is a massive undertaking to educate oneself about the different projects within the Hadoop ecosystem, understand how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

Continuuity empowers you, the developer, by abstracting away unnecessary complexity and exposing the power of Big Data through higher-level abstractions, simple REST interfaces, powerful developer tools, and the Continuuity AppFabric, a scalable and integrated runtime environment with a rich visual user interface. Using Continuuity, developers can easily and quickly build, run, and scale their own 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 AppFabric, including an overview of the core APIs, libraries, and the local UI. The getting started section will have you running your own instance of the AppFabric and deploying a sample application in minutes. To get you developing your own applications, the programming guide will deep-dive into the Core APIs and walk you through the implementation of an entire application, giving you an understanding of how the capabilities of the Continuuity AppFabric can enable you to quickly and easily build your own custom applications.

What is the Continuuity AppFabric?

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



The production Continuuity AppFabric is a distributed cloud platform, available as a hosted virtual private cloud or in an on-premise environment. For development, a single-node version of the Continuuity AppFabric is available. This runs on your local machine using an emulated version of Hadoop and HBase and makes testing and debugging your application easy. Regardless of the environment and scale, your application code and your interactions with the AppFabric remain the same.

What is the Developer Suite?

The Continuuity Developer Suite gives you everything you need to develop, test, and run your own Big Data applications: a complete set of APIs, libraries, and documentation, an IDE plugin and unit testing framework, sample applications, and a single-node version of the AppFabric to easily test and debug your applications locally. The Developer Suite is your on-ramp to the distributed Continuuity AppFabric, enabling you to develop locally and then push to a remote AppFabric with a single click. In a distributed cloud environment, you use the same interfaces to interact with your Big Data application as when you developed it but can now control the scale of your application to meet its demands, in realtime, with no application downtime.

What's in the Box?

The Continuuity Developer Suite includes the AppFabric Software Development Kit (SDK) and the single-node version of the Continuuity AppFabric.

AppFabric SDK

The SDK includes this developer guide, all of the Continuuity APIs and libraries, Javadoc, command-line tools, Eclipse IDE plugin, and sample applications. See the Getting Started section for more details.

AppFabric Single-Node Runtime

The single-node runtime is a fully functional but scaled-down version of the AppFabric that emulates the typically distributed and large-scale infrastructure in a local and lightweight way. You run the single-node process locally and deploy your applications to it, use the local UI to control and monitor it, and have direct access to your running application, making it easy to attach a debugger or profiler.

How to build Apps using Continuuity

You will build the core of your application in your own IDE using the core Continuuity Java APIs and libraries included in the AppFabric SDK. We help to get you started with an Eclipse plugin and sample projects/builds as well as a set of example applications that utilize all the various features of the AppFabric. The SDK also includes a unit-testing framework that makes it easy to write tests for your application.

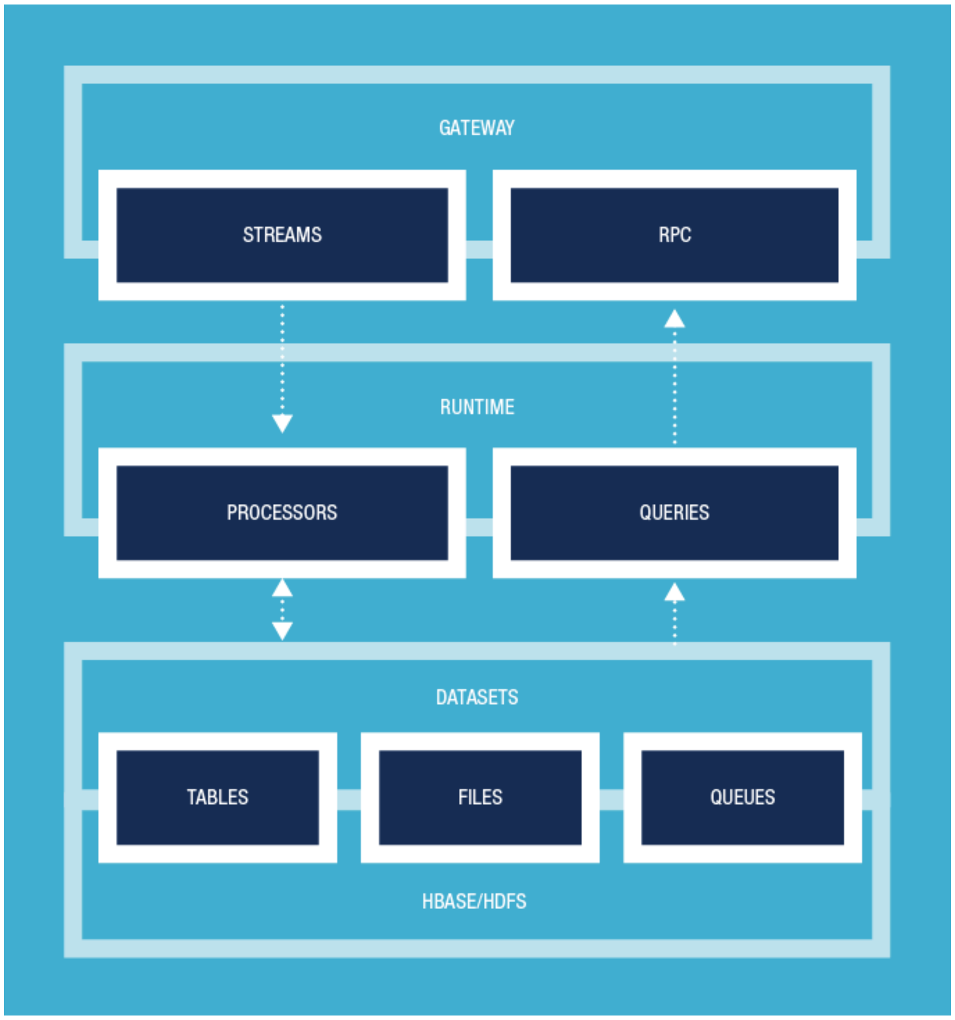
Once the first version of your app has been built, you can deploy it to your single-node AppFabric locally, using the local AppFabric UI or the Eclipse Plugin, and begin the process of testing, debugging, and iterating on your application.

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

When ready for production, your Application can be easily deployed from your local machine to a distributed instance, with no code changes or manual configuration. There it will be highly available and can be scaled to meet the dynamic demands of your app.

2. Understanding the Continuuity AppFabric

The Continuuity AppFabric is a unified Big Data application platform that brings various Big Data capabilities into a single environment and provides an elastic application runtime for user code. Data can be stored in both structured and unstructured forms; ingestion, processing, and serving can be done in realtime. Everything is elastically scalable.



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

As a developer, you will build Applications in Java using the Continuuity Core APIs. An Application is comprised of **Flows** and **Queries**, where all user code is executed, as well as **Streams** and **Datasets**, where all user data is stored.

Once your Application is implemented, you can easily integrate it with virtually any external system by accessing the Streams and Queries using REST or other network protocols.

2.1 Collect with Streams

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

Streams are identified by a *Unique Stream ID* string and must be explicitly created before being used. They can be created using a command-line tool (see Section 5.3), the Management UI, or programmatically within your Application (see Section 4.1). Data written to a Stream can then be consumed by Flows and processed in realtime as described below.

2.2 Process with Flows

Flows are user-implemented realtime stream processors. They are comprised of one or more **Flowlets** that are wired together into a DAG (Directed Acyclic Graph). Flowlets pass **Tuples** between one another and each Flowlet is able to perform custom logic and execute data operations for each individual Tuple it processes, in a consistent and durable way (more about this in the programming guide).

A Tuple is a map of string names to typed values, for example, “name” to String and “id” to Long. Each Flowlet can have multiple **FlowletInputs** and **FlowletOutputs**, with each of these defined by a name and a **TupleSchema**, which defines the field names and types of the Tuples sent over a given Flowlet input or output.

To get data into your Flow, you can either connect the input of the Flow to a Stream, or you can implement a **SourceFlowlet,** which will allow you to write custom code in order to generate data or pull it from an external source.

Flows are deployed into the AppFabric and hosted within containers, such that each instance of a Flowlet runs in its own container. Each Flowlet in the DAG can have multiple concurrent instances, with each consuming a unique partition of the Flowlet’s input Tuples.

To learn more about Flows see Section 4.2

2.3 Store with Datasets

Datasets are the way that you read and write persisted data in the AppFabric. An instance of a Dataset is always of a particular type and includes both the data itself as well as an API to read and write that data. The core Dataset type is Table and all other Datasets are implemented using one or more of them. You may have an instance of a Dataset that is of type *Table*, has the name “*myUserData*”, and contains 5GB of data.

In addition to the core Dataset types, the AppFabric includes a number of system Dataset types such as KeyValueTable, IndexedTable, CounterTable, and TImeseriesTable.

You can also implement your own custom Datasets by composing one or more existing Datasets into a new class with a new or extended API.

In addition to the Java API, all system Datasets are available via REST. However, custom Datasets must utilize a QueryProvider in order to expose their API, as described below.

To learn more about Datasets see Section 4.3

2.4 Query with QueryProviders

QueryProviders allow you to make synchronous calls into the AppFabric from external systems and perform arbitrary server-side processing on-demand, similar to a stored procedure in a traditional database. A QueryProvider 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 queried from any external system.

A QueryProvider can perform all the same operations on Datasets as a Flowlet. You can also perform complex post-processing of data and implement access patterns beyond simple lookup and retrieve, such as filters, aggregations and even joins at query time. QueryProviders are deployed into the same pool of application containers as Flows, and you can run multiple instances to increase the throughput of requests.

2.5 Package with Applications

Applications are the highest-level concept and serve to specify and package all the components and configuration of your Big Data application. Within the Application, you can explicitly indicate (and if necessary, create) your Streams and Datasets and declare all of the Flows and QueryProviders that make up the app.

2.6 AppFabric Runtime Editions

The Continuuity AppFabric can be run in three different modes: in-memory mode for unit testing, local mode for local testing, and distributed mode for staging and production. Regardless of the runtime edition, the AppFabric is fully functional and the code you develop never changes, however performance and scale are limited in in-memory and local modes.

In-Memory AppFabric

The in-memory AppFabric allows you to easily run the AppFabric for use in JUnit tests. In this mode, the underlying Big Data infrastructure is being emulated using in-memory data structures and there is no persistence. There is no UI available in this mode.

You can take advantage of the **AppFabricTestBase** class to make spinning up the in-memory AppFabric easy. See *Section 4.5* in the Programming Guide for more information.

Local AppFabric

The local AppFabric allows you to run the entire AppFabric stack in a single JVM on your local machine and also includes a local version of the UI. The underlying Big Data infrastructure is being emulated on top of your local filesystem and HyperSQL, and all data is persisted.

See *Section 3.3* in the Getting Started section for more information on how to start and manage your local AppFabric.

Distributed AppFabric

All hosted versions of the Continuuity AppFabric run in a fully distributed mode. This includes distributed and highly available deployments of the underlying Big Data infrastructure as well as the other system components of the AppFabric. Production applications should always be run in a Distributed AppFabric.

To learn more about getting your own Distributed AppFabric, check out http://continuuity.com/products!

3. Getting Started with the Local AppFabric

Before diving into the hands-on development of your Application, it can be useful to build, deploy, and run one of the sample Applications using the single-node AppFabric and local UI.

3.1 Unpacking the Developer Suite

The Continuuity AppFabric Developer Suite is bundled as a tar or zip file and contains everything you need to build and run Big Data applications on your local machine. It includes the AppFabric API Jar, documentation, tools, IDE plugin, and sample applications:

`DeveloperGuide.pdf`	`AppFabric Developer Guide`
`LICENSE`	`Developer Suite License`
`README`	`Developer Suite Readme File`
`VERSION`	`Developer Suite Release Version`
`continuuity-api-1.3.0.jar`	`AppFabric Core API Jar`
`continuuity-api-javadoc-1.3.0.jar`	`AppFabric Core API Javadoc`
`continuuity-api-source-1.3.0.jar`	`AppFabric Core API Source`
`continuuity-eclipse-plugin-1.0.0.jar`	`AppFabric Eclipse Plugin Jar`
`bin/benchmark`	`Benchmarking Tool`
`bin/continuuity-app-fabric`	`Single-Node AppFabric Daemon`
`bin/data-client`	`Command-Line Dataset Client`
`bin/data-format`	`Single-Node Data Format Tool`
`bin/flow-client`	`Command-Line Flow Client`
`bin/query-client`	`Command-Line Query Client`
`bin/stream-client`	`Command-Line Stream Client`
`conf/continuuity-site.xml`	`Single-Node AppFabric Configuration`
`conf/logback.xml`	`Single-Node AppFabric Log Configuration`
`examples/WordCounter`	`Sample Word Count Application`
`examples/CountAndFilterWords`	`Sample Word Filter Application`
`examples/CountCounts`	`Sample Number Counter Application`
`examples/CountOddAndEven`	`Sample Odd/Even Number Application`
`examples/CountRandom`	`Sample Random Number Generator App`
`examples/CountTokens`	`Sample String Counting Application`
`examples/DependencyRandomNumber`	`Sample Application w/ Extra Dependency`
`examples/SimpleWriteAndRead`	`Sample Dataset using Application`
`examples/WordCountApp`	`Sample App used in Programming Guide`

3.2 Building Example Applications

Building the example applications is simple using ant:

```
> cd ~/continuuity-developer-edition-1.3.0/examples
> ant
```

This will generate a JAR file for each of the sample apps.

You can also individually build a single example:

```
> cd ~/continuuity-developer-edition-1.3.0/examples/WordCounter
> ant
```

3.3 Starting the Single-Node AppFabric

To start the single-node AppFabric, simple run the *continuuity-app-fabric* daemon with the start command:

```
> cd ~/continuuity-developer-edition-1.3.0/
> ./bin/continuuity-app-fabric start
```

Your single-node AppFabric is now up-and-running. You can check on it's status or stop it as follows:

```
> ./bin/continuuity-app-fabric status
...
> ./bin/continuuity-app-fabric stop
...
```

The URL for the local AppFabric UI should have been printed to your screen upon starting single-node. It is also usually accessible via http://localhost:9999.

3.4 Deploying and Running Applications using the AppFabric UI

Now that the single-node AppFabric instance is running and you have accessed your local (but empty) AppFabric UI, you can easily deploy and run one of the bundled sample apps. In this example, deploy the *WordCountApp* by drag-and-dropping the *WordCountApp.jar* file onto the UI.

1. Click on "Create Application" to open the new Application dialog box
2. Drag-and-drop your WordCountApp.jar onto the new Application dialog
3. Your Application is now uploaded, deployed, and verified

With your Application deployed, you can now start the different Flows and Queries within it.

1. Click on the WordCounter Application in the Dashboard Application list
2. You should see all of the different components of this app: a Stream, a Flow, four Datasets, and a Query
3. Click on the WordCounter Flow to open the Flow Visualizer page
4. Press the *START* button in the top-right corner to run the Flow

The Flow is running, so now it can begin to process incoming data from the Stream. In order to send a sample event using the UI:

1. Click on the "*wordStream"* Stream icon to open the Stream dialog.
2. Type a string of words into the top-right textbox and click the INJECT button
3. Watch the event get processed by the Flowlets in the DAG

You can also send events to a Stream using REST (see Section 5.2) or the command-line (see Section 5.3).

After you have processed some data off the Stream and with the Flow, use the QueryProvider to read the results of the processing:

1. Return to the WordCount Application page
2. Click on the *"wordcount"* QueryProvider to open the QueryProvider page
3. Press the START button in the top-right corner to run the QueryProvider

The QueryProvider is now running and ready to accept new requests. QueryProviders bind to REST interfaces so it is easy to query them using any HTTP-based tools or libraries (see Section 5.2). Additionally, you can use the QueryProvider UI to send REST requests and receive the response directly in the UI.

1. Enter *"getStats"* in the method box and press the REQUEST button
2. JSON results should be displayed in the large text box

4. AppFabric Programming Guide

This section dives into more detail around each of the different AppFabric core capabilities and how you will work with each of these in Java to build your Big Data Application: Streams, Flows, Datasets, and QueryProviders.

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

4.1 AppFabric Core APIs

Application An Application is a collection of Streams, Flows, Datasets, and Queries. To create an Application you implement the *Application* interface and it's configure() method. This allows you to specify the metadata of the Application, declare and configure the Streams and Datasets used, and add the Flows and Queries that are part of the application.

```
public class MyApplication implements Application {
  @Override
  public ApplicationSpecification configure() {
    return ApplicationSpecification.builder()
        .setApplicationName(...)
        .addStream(...)
        .addDataset(...)
        .addFlow(...)
        .addQuery(...)
        .create();
  }
}
```

Stream Streams are the primary means for pushing data into the AppFabric. You can create a Stream in your Application using the following API:

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

Flow Flows are a collection of connected Flowlets, wired into a DAG. To create a Flow you implement the *Flow* interface and it's *configure()* method. This allows you to specify its metadata, Flowlets, Flowlet connections, Stream to Flowlet connections, and any Datasets used in the Flow using a FlowSpecifier:

```
public class MyExampleFlow implements Flow {
  @Override
  public void configure(FlowSpecifier specifier) {
    specifier.name("ExampleFlow");
    specifier.flowlet("ExampleFlowlet", MyExampleFlowlet.class, 1);
    specifier.flowlet("ExampleFlowlet2", MyExampleFlowlet2.class, 1);
    specifier.input("myStream", "ExampleFlowlet");
    specifier.connection("ExampleFlowlet", "ExampleFlowlet2");
    specifier.dataset("myDataset");
  }
}
```

Flowlet

The basic building block of a Flow, these represent each individual processing node within a Flow. Flowlets consume Tuples from their **TupleInput**s and execute custom logic on each Tuple, allowing you to perform data operations as well as emit Tuples to their **TupleOutput**s. The input and output tuple streams are set in the *configure()* method. Flowlets also specify an *initialize()* method which is executed at the startup of each instance of a Flowlet before it receives any Tuples.

There are two types of Flowlets. The **ComputeFlowlet** is the primary Flowlet type and should be used for every Flowlet except for cases when you want to generate or pull new Tuples using a **SourceFlowlet**. The only difference is that a ComputeFlowlet specifies a *process()* method and is given a Tuple one-by-one, whereas a SourceFlowlet specifies a *generate()* method that is called and expected to return new Tuples.

In the example below, a source flowlet generates random doubles while a compute flowlet takes these random doubles and rounds them to the nearest long.

```
public class RandomDoubleGeneratingFlowlet extends SourceFlowlet {

  @Override
  public void configure(FlowletSpecifier specifier) {
    specifier.getDefaultFlowletInput().setSchema(
        new TupleSchemaBuilder().add("randomDouble", Double.class)
            .create());
  }

  private Random random = new Random();

  @Override
  public void generate(OutputCollector out) {
    out.add(new TupleBuilder().set(“randomDouble”, random.nextDouble())
        .create());
  }
}

public class DoubleRoundingFlowlet extends ComputeFlowlet {
  @Override
  public void configure(FlowletSpecifier specifier) {
    specifier.getDefaultFlowletInput().setSchema(
        new TupleSchemaBuilder().add("randomDouble", Double.class)
            .create());
    specifier.getDefaultFlowletOutput().setSchema(
        new TupleSchemaBuilder().add("randomLong", Long.class)
            .create());
  }
  @Override
  public void process(Tuple tuple, TupleContext tupleContext,
      OutputCollector out) {
    out.add(new TupleBuilder().set(“randomLong”,
        Math.round((double)tuple.get(“randomDouble”))).create());
  }
}
```

Tuple

The core data structure in which information is passed through a Flow. It is an immutable map of named fields to typed values, for example it could contain a field of name “age” with an Integer value of 21. The schema of a tuple is defined in a *TupleSchema* (defined using a *TupleSchemaBuilder*) and tuples themselves can be created using a *TupleBuilder*.

```
TupleSchema userTupleSchema = new TupleSchemaBuilder()
       .add(“name”, String.class)
       .add(“age”, Integer.class)
       .create();

Tuple userTuple = new TupleBuilder()
       .add(“name”, “Joe”)
       .add(“age”, 21)
       .create();
```

Event

A special type of Tuple that comes in via Streams. It has a predefined schema with exactly two fields: a “body” field with a value of type byte array and a “headers” field with a value of type map from string to string.

```
TupleSchema eventSchema = new TupleSchemaBuilder()
       .add(“body”, byte[].class)
       .add(“headers”, Map<String,String>.class)
       .create();
```

FlowletInput /

FlowletOutput

The inputs and outputs of a Flowlet that connect to other Flowlets or an input that connects to a Stream are called FlowletInputs and FlowletOutputs. They are specified by a string name and a TupleSchema. You configure these in the *Flowlet.configure()* method as described in Flowlet section above.

QueryProvider

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

To create a Query you extend the *QueryProvider* base class, which looks similar to a ComputeFlowlet but instead of a *process()* method it specifies an *execute()* method which is given a method name and map of string arguments, translated from the external request.

```
public class MyHelloWorldQuery extends QueryProvider {
  @Override
  public void configure(QuerySpecifier specifier) {
    specifier.provider(MyHelloWordQuery.class);
    specifier.service("helloworld");
    specifier.timeout(10000);
    specifier.type(QueryProviderContentType.JSON);
  }

  @Override
  public QueryProviderResponse execute(String method,
      Map<String,String> args) {
    return new QueryResponse("{'msg':'Hello World'}");
  }
}
```

Dataset Datasets are the way you store and retrieve data. You can create instances of Datasets within your Flows and QueryProviders using the Dataset's constructor.

```
Dataset myKVtable = new KeyValueDataset("myKVtable");
```

You can also implement your own Datasets by extending the *Dataset* base class or extending any other existing type of Dataset. More about this in the programming example in Section 4.4

4.2 The Flow System

The Flow system is a realtime stream-processing engine where you specify a DAG of Flowlets. These Flowlets pass Tuples between each other and can perform arbitrary logic for each Tuple, including performing Dataset operations, in a consistent and durable manner. Flowlets pass Tuples using their FlowletInputs and FlowletOutputs which are connected using a queue.

A unique and core property of the Flow system is that it is transactional and guarantees exactly-once-execution of each Tuple by each Flowlet in the DAG. It also guarantees ordering of the Tuples and exactly-once-execution of any Dataset write operations you perform asynchronously in the Flowlet process() method. See below for more detail and the example in Section 4.4 will also help to illustrate how Flows and Flowlets behave.

Flowlet Behavior: Tuples, Transactions, and Datasets

A Flowlet processes the tuples from its FlowletInputs one at a time. For each input tuple, the following occurs:

1. A Tuple is removed from one of the FlowletInputs by marking it as consumed from the source queue
   (If a Flowlet has multiple inputs, they are consumed from them in a non-blocking round-robin)
2. The *process()* method is invoked on the Flowlet, passing the removed Tuple

Within the *process()* method you implement in each Flowlet, you can perform the following work:

1. Custom Java processing logic
2. Performing synchronous Dataset read operations
3. Sending Tuples to downstream Flowlets using FlowletOutputs
4. Perform asynchronous Dataset write operations

   Note: You should not perform synchronous write operations in your Flowlet. See below.

Once your *process()* finishes, the following occurs:

1. All the outputted Tuples and asynchronous Dataset write operations are performed within a transaction
2. If the transaction **fails**
   a. None of the Tuples outputted will be sent downstream
   b. None of the asynchronous Dataset write operations will be performed
   c. The *onFailure()* callback of the Flowlet is invoked which allows you to retry or ignore
      i. If you retry, process() will be invoked with the same Tuple again
      ii. If you ignore, the original input Tuple from the source queue is acknowledged
3. If the transaction **succeeds**
   a. The outputted Tuples are sent downstream to destination queues
   b. The Dataset operations will be performed
   c. The original input Tuple from the source queue is acknowledged
   d. The *onSuccess()* callback of the Flowlet is invoked

With the help of transactions, we ensure that either all the operations succeed or all of them fail or are rolled back. In case of failure, processing of the Tuple can be reattempted. This ensures exactly-once processing of each input Tuple.

Flows and Instances

You can have one or *more* instances of any given flowlet, and you can control this programmatically, using the UI, or using the command line interface. This enables you to shape your application to meet capacity at runtime the same way you will do it in production. In the single-node version provided with the Developer Suite, multiples instances of a Flowlet are run in threads, so in some cases actual performance may not be affected. In production, each instance is a separate JVM with independent compute resources.

Getting Data In

Input data can be pushed to a Flow using **Streams** or pulled from within a Flow using a **SourceFlowlet**.

- A **Pull** source actively retrieves or generates data, and the logic to do so is coded into the Flowlet's generate() method. For instance, a SourceFlowlet can connect to the Twitter firehose or another external data source.
- A **Push** source is passively receiving events from a stream (remember that Streams exist outside the scope of a Flow) and converts the events into tuples on its output queues. 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 app, because your test driver can send mock data to the stream that covers all your test cases.

4.3 The Dataset System

A Dataset represents both an API to read and write data using a specific pattern as well as the instance of the data itself. In other words, a Dataset Class is a reusable, generic Java implementation of a common data pattern. A Dataset Instances is a named collection of data with associated metadata, and it is manipulated through a Dataset Class.

Dataset Types

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 instance, the "indexed table" Dataset can be implemented by two underlying "table" Datasets, on holding the data itself, and one holding the index. We distinguish three categories of Datasets: basic, system, and custom Datasets:

- A **Basic Dataset** is a primitive of the data fabric, such as table or a file. Its implementation is hidden from developers, and it may use data fabric operations that are not available to developers.
- A **System Dataset** is bundled with the AppFabric but implemented in the same way as a custom Dataset, relying on one or more underlying basic or system Datasets.
- A **Custom Dataset** is implemented by the developer and can have arbitrary code, but it can only interact with the data fabric through its underlying Datasets.

The only difference between custom and system Datasets is that system Datasets are implemented (and tested) by Continuuity and as such they are reliable and trusted code.

Each Dataset instance has exactly one Dataset class to manipulate it - we can 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 meta data that defines its behavior. For instance, every "indexed table" has a name and indexes a particular column of its primary table: The name of that column is a metadata property of each instance. Datasets are created (implicitly) when an application is deployed that declares the Dataset in its own specification. The declaration of the Dataset must include it name and all its metadata (and hence, all names and meta data of its underlying Datasets). This allows creating the Dataset - if it does not exist yet - and store its metadata. Application code (for instance a flow or query provider) can then open a Dataset by giving only its name and type - the execution context can use the stored metadata to create an instance of the Dataset class with all required metadata.

4.4 End-to-End Programming Example

To illustrate how all the capabilities of the AppFabric can be put together to build an Application, a simple end-to-end example will be implemented using a slightly modified version of the classic Word Count[1].

This Application, named **WordCountApp**, consists of the following:

1. A Stream named *"wordStream"* that we will send strings of words to be counted
2. A Flow named *"WordCountFlow"* that will process the strings from the Stream to calculate the word counts and other word statistics using four Flowlets and four Datasets:
    1. A Flowlet named *"Splitter"* that will split the input string into words and perform any scrubbing
    2. A Flowlet named *"Counter"* that will take the input word and perform a set of Dataset operations in order to calculate the required word count statistics
    3. A Flowlet named *"UniqueCounter"* that is used to calculate the unique number of words seen
    4. A Flowlet named *"Associater"* that is used to store word associations between all the words in each input string.
3. Four Datasets will be used by the Flow and Query in order to model, store, and serve the necessary data
    1. A primitive *Table* Dataset named "*wordCounts*" used to count the occurrences of each word
    2. A primitive *Table* Dataset named "*wordStats*" used to track other word statistics
    3. A custom *UniqueCountTable* Dataset named "*uniqueCount*" used to determine and count the number of unique words seen
    4. A custom *WordAssocTable* Dataset named "*wordAssocs*" used to track associations between words in the input strings
4. A QueryProvider named "*WordCountQuery*" which will utilize the four Datasets in order to process and serve read requests for the calculated word counts, statistics, and associations. It will support two methods:

    getCount for accessing the word count of a specified word and it's word associations
    getStats for accessing the global word statistics

Defining an Application

The definition of the Application is straightforward and simply wires together all of the different components described above:

```
public class WordCountApp implements Application {
  @Override
  public ApplicationSpecification configure() {
    return ApplicationSpecification.builder()
        .setApplicationName("WordCounter")
        .addStream(new Stream("wordStream"))
        .addDataSet(new Table("wordStats"))
        .addDataSet(new Table("wordCounts"))
        .addDataSet(new UniqueCountTable("uniqueCount"))
        .addDataSet(new WordAssocTable("wordAssocs"))
        .addFlow(WordCountFlow.class)
        .addQuery(WordCountQuery.class)
        .create();
  }
}
```

Defining a Flow

You define a Flow by creating a Class that implements the Flow interface.

```
public class WordCountFlow implements Flow { ... }
```

This interface defines a single configure() method where you will define all the metadata, Streams, Flowlets, and connections. In the example below, we specify the name and e-mail, the four Flowlets, the Stream input, and the connections between Flowlets.

```
@Override
  public void configure(FlowSpecifier specifier) {
    // Specify meta data fields
    specifier.name("WordCounter");
    specifier.email("demo@continuuity.com");
    // Specify flowlets
    specifier.flowlet("Splitter", WordSplitterFlowlet.class, 1);
    specifier.flowlet("Counter", WordCounterFlowlet.class, 1);
    specifier.flowlet("Associater", WordAssociaterFlowlet.class, 1);
    specifier.flowlet("UniqueCounter", UniqueCounterFlowlet.class, 1);
    // Specify input stream
    specifier.input("wordStream", "Splitter");
    // Connect flowlets
    specifier.connection("Splitter", "wordArrays", "Associater", "in");
    specifier.connection("Splitter", "words", "Counter", "in");
    specifier.connection("Counter", "UniqueCounter");
    // Specify datasets used
    specifier.dataset("wordStats");
    specifier.dataset("wordCounts");
    specifier.dataset("uniqueCount");
    specifier.dataset("wordAssocs");
  }
```

It can also be handy to define the different TupleSchemas that you will use across the FlowletInputs and FlowletOutputs in your Flow Class so that you can keep them in one place and reference them in the Flowlets. Below the schemas used between the Splitter and Associater and the Splitter and Counter Flowlets are defined.

```
  public static final TupleSchema SPLITTER_ASSOCIATER_SCHEMA =
      new TupleSchemaBuilder()
          .add("wordArray", String[].class)
          .create();
  public static final TupleSchema SPLITTER_COUNTER_SCHEMA =
      new TupleSchemaBuilder()
          .add("word", String.class)
          .create();
```

Implementing Flowlets

With the Application and Flow defined, it's now time to dig into the actual logic of our app. The processing logic and data write operations occur within Flowlets. For each Flowlet you create a Class that extends the abstract *ComputeFlowlet* base class.

The WordCountApp contains four different Flowlets, the Splitter, Counter, UniqueCounter, and Associater. The implementation of each of these is shown below with an overview of each.

Splittler Flowlet

SplitterFlowlet is the first in the Flow and it's FlowletInput connects to the Stream "*wordStream*".

It takes an arbitrary String of data from the Stream and splits it into individual words, removing any non-alphabet characters from the words.

Each word is sent downstream to the *Counter* Flowlet and the entire array of words is sent to the *Associater.*

```
public class WordSplitterFlowlet extends ComputeFlowlet {
  @Override
  public void configure(FlowletSpecifier specifier) {
    specifier.getDefaultFlowletInput().setSchema(
        TupleSchema.EVENT_SCHEMA);
    specifier.addFlowletOutput("wordArrays").setSchema(
        WordCountFlow.SPLITTER_ASSOCIATER_SCHEMA);
    specifier.addFlowletOutput("words").setSchema(
        WordCountFlow.SPLITTER_COUNTER_SCHEMA);
  }
  @Override
  public void process(Tuple tuple, TupleContext context,
      OutputCollector collector) {
    // Input is a String, need to split it by whitespace
    byte [] rawInput = tuple.get("body");
    String inputString = new String(rawInput);
    String [] words = inputString.split("\\s+");
    // We have an array of words, now remove all non-alpha characters
    for (int i=0; i<words.length; i++) {
      words[i] = words[i].replaceAll("[^A-Za-z]", "");
    }
    // Send the array of words to the associater
    collector.add("wordArrays",
        new TupleBuilder().set("wordArray", words).create());
    // Then emit each word to the counter
    for (String word : words) {
      collector.add("words",
          new TupleBuilder().set("word", word).create());
    }
  }
}
```

COUNTER FLOWLET

CounterFlowlet is sent Tuples that contain a single word. It uses Datasets to perform the core word count, calculate other word statistics, and perform the first step necessary to use the UniqueCountTable Dataset (described more below).

```
public class WordCounterFlowlet extends ComputeFlowlet {
```

```
@Override
public void configure(FlowletSpecifier specifier) {
  specifier.getDefaultFlowletInput().setSchema(
      WordCountFlow.SPLITTER_COUNTER_SCHEMA);
  specifier.getDefaultFlowletOutput().setSchema(
      UniqueCountTable.UNIQUE_COUNT_TABLE_TUPLE_SCHEMA);
}
private Table wordStatsTable;
private Table wordCountsTable;
private UniqueCountTable uniqueCountTable;
@Override
public void initialize() {
  try {
    this.wordStatsTable = getFlowletContext().getDataSet("wordStats");
    this.wordCountsTable = getFlowletContext().getDataSet("wordCounts");
    this.uniqueCountTable = getFlowletContext().getDataSet("uniqueCount");
  } catch (Exception e) {
    e.printStackTrace();
    throw new RuntimeException(e);
  }
}
@Override
public void process(Tuple tuple, TupleContext context,
    OutputCollector collector) {
  String word = tuple.get("word");
  try {
    // Count number of times we have seen this word
    this.wordCountsTable.stage(
        new Increment(Bytes.toBytes(word), Bytes.toBytes("total"), 1L));
    // Count other word statistics (word length, total words seen)
    this.wordStatsTable.stage(
        new Increment(Bytes.toBytes("totals"),
            new byte [][] {
              Bytes.toBytes("total_length"), Bytes.toBytes("total_words")
            },
            new long [] { word.length(), 1L }));

    // For the unique counter, writing the entry to the table will create a

    // Tuple that we send to the UniqueCounter Flowlet

    Tuple outputTuple =
        this.uniqueCountTable.writeEntryAndCreateTuple(word);

    collector.add(outputTuple);


  } catch (OperationException e) {

    // Fail the process() call if we get an exception

    e.printStackTrace();

    throw new RuntimeException(e);

  }

}
```

UniqueCounter Flowlet

UniqueCounterFlowlet receives a special Tuple that is generated using the UniqueCountTable Dataset. This Tuple generated in the *CounterFlowlet* is then fed back to the UniqueCountTable in this Flowlet. See the implementation of this Dataset for more information.

```
public class UniqueCounterFlowlet extends ComputeFlowlet {


  @Override

  public void configure(FlowletSpecifier specifier) {

    specifier.getDefaultFlowletInput()

        .setSchema(UniqueCountTable.UNIQUE_COUNT_TABLE_TUPLE_SCHEMA);

  }


  private UniqueCountTable uniqueCountTable;


  @Override

  public void initialize() {

    try {

      this.uniqueCountTable = getFlowletContext().getDataSet("uniqueCount");

    } catch (Exception e) {

      e.printStackTrace();

      throw new RuntimeException(e);

    }
  }

  @Override
  public void process(Tuple tuple, TupleContext context,
      OutputCollector collector) {
    try {
      this.uniqueCountTable.updateUniqueCount(tuple);
    } catch (OperationException e) {
      // Fail the process() call if we get an exception
      e.printStackTrace();
      throw new RuntimeException(e);
    }
  }
}
```

Associater Flowlet

AssociaterFlowlet receives an array of words and stores associations between each of them using a custom Dataset, the WordAssocTable, described below.

```
public class WordAssociaterFlowlet extends ComputeFlowlet {
  @Override
  public void configure(FlowletSpecifier specifier) {
    specifier.getDefaultFlowletInput().setSchema(
        WordCountFlow.SPLITTER_ASSOCIATER_SCHEMA);
  }

  private WordAssocTable wordAssocTable;

  @Override
  public void initialize() {
    try {
      this.wordAssocTable = getFlowletContext().getDataSet("wordAssocs");
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
  @Override
  public void process(Tuple tuple, TupleContext context,
      OutputCollector collector) {
    String [] words = tuple.get("wordArray");
    try {
      // Store word associations
      Set<String> wordSet = new TreeSet<String>(Arrays.asList(words));
      this.wordAssocTable.writeWordAssocs(wordSet);

    } catch (OperationException e) {
      // Fail the process() call if we get an exception
      throw new RuntimeException(e);
    }
  }
}
```

Implementing Custom Datasets

This Application uses four Datasets, two of which are the basic Table types, and two of which are custom Datasets built using the basic Table types.

The first custom Dataset is the *WordAssocTable* and it tracks associations between a bag (array) of words. Rather than requiring that this pattern be implemented within our Flowlets and Queries, it can be implemented using a custom Dataset, exposing a simple and specific API rather than a complex and generic one. In this case, it exposes two primary methods, *writeWordAssocs()* and *readWordAssocs()*.

```
public class WordAssocTable extends DataSet {
  private Table table;
  public WordAssocTable(String name) {
    super(name);
    this.table = new Table("word_assoc_" + name);
  }
  public WordAssocTable(DataSetSpecification spec) {
    super(spec);
    this.table = new Table(
        spec.getSpecificationFor("word_assoc_" + this.getName()));
  }
  @Override
  public DataSetSpecification configure() {
```

```
  return new DataSetSpecification.Builder(this)
      .dataset(this.table.configure())
      .create();
}
public Map<String,Long> readWordAssocs(String word, int limit)
    throws OperationException {
  // Read all columns
  OperationResult<Map<byte[], byte[]>> result =
      this.table.read(new Read(Bytes.toBytes(word), null, null));
  Map<String,Long> wordAssocs = new TreeMap<String,Long>();
  if (!result.isEmpty()) {
    for (Map.Entry<byte[],byte[]> entry : result.getValue().entrySet()) {
      String assocWord = Bytes.toString(entry.getKey());
      Long assocCount = Bytes.toLong(entry.getValue());
      wordAssocs.put(assocWord, assocCount);
    }
    if (wordAssocs.size() > limit) {
      wordAssocs = sortAndLimit(wordAssocs, limit);
    }
  }
  return wordAssocs;
}

// Some helper function code removed, see examples for full implementation


public void writeWordAssocs(Set<String> words) throws OperationException {
  for (String rootWord : words) {
    for (String assocWord : words) {
      if (!rootWord.equals(assocWord)) {
        writeWordAssoc(rootWord, assocWord);
      }
    }
  }
}
public void writeWordAssoc(String word, String assocWord)
```

```
      throws OperationException {
    this.table.stage(
        new Increment(Bytes.toBytes(word), Bytes.toBytes(assocWord), 1L));
  }
}
```

The second custom Dataset is the UniqueCountTable and it is fed String entries and continuously calculating the unique number of String entries it has seen.

```
public class UniqueCountTable extends DataSet {
  public static final TupleSchema UNIQUE_COUNT_TABLE_TUPLE_SCHEMA =
      new TupleSchemaBuilder()
          .add("entry", String.class)
          .add("count", Long.class)
          .create();

  private Table uniqueCountTable;
  private Table entryCountTable;

  public UniqueCountTable(String name) {
    super(name);
    this.uniqueCountTable = new Table("unique_count_" + name);
    this.entryCountTable = new Table("entry_count_" + name);
  }

  public UniqueCountTable(DataSetSpecification spec) {
    super(spec);
    this.uniqueCountTable = new Table(
        spec.getSpecificationFor("unique_count_" + this.getName()));
    this.entryCountTable = new Table(
        spec.getSpecificationFor("entry_count_" + this.getName()));
  }

  @Override
  public DataSetSpecification configure() {
    return new DataSetSpecification.Builder(this)
```

```
      .dataset(this.uniqueCountTable.configure())
      .dataset(this.entryCountTable.configure())
      .create();
}



/** Row and column names 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");

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

public Tuple writeEntryAndCreateTuple(String entry)
    throws OperationException {
  Closure closure = this.entryCountTable.closure(
      new Increment(Bytes.toBytes(entry), ENTRY_COUNT, 1L));
  return new TupleBuilder()
      .set("entry", entry)
      .set("count", closure)
      .create();
}

public void updateUniqueCount(Tuple tuple)
    throws OperationException {
  Long count = tuple.get("count");
  if (count == 1L) {
      this.uniqueCountTable.stage(
          new Increment(UNIQUE_COUNT, UNIQUE_COUNT, 1L));
    }
  }
}
```

Implementing a QueryProvider

The Flow and Flowlets using the above four Datasets are handling all of the processing and storing of data. Now to read this data and serve it back out of the system, a QueryProvider must be implemented, which will bind a handler to a REST endpoint to get external access.

```
public class WordCountQuery extends QueryProvider {

  @Override
  public void configure(QuerySpecifier specifier) {
    specifier.provider(WordCountQuery.class);
    specifier.service("wordcount");
    specifier.timeout(10000);
    specifier.type(QueryProviderContentType.JSON);
    // Specify datasets used
    specifier.dataset(“wordStats”);
    specifier.dataset(“wordCounts”);
    specifier.dataset(“uniqueCount”);
    specifier.dataset(“wordAssocs”);
  }

  private Table wordStatsTable;
  private Table wordCountsTable;
  private UniqueCountTable uniqueCountTable;
  private WordAssocTable wordAssocTable;

  @Override
  public void initialize() {
    try {
      this.wordStatsTable = getQueryProviderContext().getDataSet("wordStats");
      this.wordCountsTable =
          getQueryProviderContext().getDataSet("wordCounts");
      this.uniqueCountTable =
          getQueryProviderContext().getDataSet("uniqueCount");
      this.wordAssocTable =
          getQueryProviderContext().getDataSet("wordAssocs");
    } catch (Exception e) {
      e.printStackTrace();
      throw new RuntimeException(e);
    }
  }

  @Override
  public QueryProviderResponse process(String method,
      Map<String, String> arguments) {
    try {
      if (method.equals("getStats")) {
        // query 1 is 'getStats' and returns all the global statistics, noargs
        // wordsSeen, uniqueWords, avgWordLength
        // Read the total_length and total_words to calculate average length
        OperationResult<Map<byte[],byte[]>> result = this.wordStatsTable.read(
            new Read(Bytes.toBytes("total"), Bytes.toBytes("total_length")));
        Long totalLength = result.isEmpty() ? 0L :
          Bytes.toLong(result.getValue().get(Bytes.toBytes("total_length")));
        result = this.wordStatsTable.read(
            new Read(Bytes.toBytes("total"), Bytes.toBytes("total_words")));
        Long totalWords = result.isEmpty() ? 0L :
          Bytes.toLong(result.getValue().get(Bytes.toBytes("total_words")));
        Double avgLength = new Double(totalLength) / new Double(totalWords);
        // Read the unique word count
        Long uniqueWords = this.uniqueCountTable.readUniqueCount();
        // Construct and return the JSON string
        String ret = "{'wordsSeen':" + totalWords + ",'uniqueWords':" +
              uniqueWords + ",'avgLength':" + avgLength + "}";
        return new QueryProviderResponse(ret);
      } else if (method.equals("getCount")) {
        // query 2 is 'getCount' with argument of word='', optional limit=#
        // returns count of word and top words associated with that word,
        // up to specified limit (default limit = 10 if not specified)
        if (!arguments.containsKey("word")) {
          String msg = "Method 'getCount' requires argument 'word'";
          getQueryProviderContext().getLogger().error(msg);
          return new QueryProviderResponse(Status.FAILED, msg, "");
        }
        // Parse the arguments
        String word = arguments.get("word");
        Integer limit = arguments.containsKey("limit") ?
            Integer.valueOf(arguments.get("limit")) : 10;
        // Read the word count
        OperationResult<Map<byte[],byte[]>> result =
            this.wordCountsTable.read(
              new Read(Bytes.toBytes(word), Bytes.toBytes("total")));
        Long wordCount = result.isEmpty() ? 0L :
          Bytes.toLong(result.getValue().get(Bytes.toBytes("total")));
        // Read the top associated words
        Map<String,Long> wordsAssocs =
            this.wordAssocTable.readWordAssocs(word, limit);
        // Construct and return the JSON string
        StringBuilder builder = new StringBuilder("{'word':'" + word +
            "','count':" + wordCount + "," + "'assocs':[");
        boolean first = true;
        for (Map.Entry<String, Long> wordAssoc : wordsAssocs.entrySet()) {
          if (!first) builder.append(",");
          else first = false;
          builder.append("{'word':'" + wordAssoc.getKey() + "',");
          builder.append("'count':" + wordAssoc.getValue() + "}");
        }
        builder.append("]}");
        return new QueryProviderResponse(builder.toString());
      } else {
        getQueryProviderContext().getLogger().error("Invalid method: " +
method);
        return new QueryProviderResponse(Status.FAILED,
            "Invalid method: " + method, "");
      }
    } catch (OperationException e) {
      getQueryProviderContext().getLogger().error(
          "Exception during query", e);
      return new QueryProviderResponse(Status.FAILED,
          "Exception occurred: " + e.getLocalizedMessage(), "");
    }
  }
}
```

4.5 Testing your Application

The AppFabric test framework simplifies the process of writing tests using JUnit4 for Flows and QueryProviders by providing the following features:

- Start in-memory Flows/Queries in test cases
- Introspect running Flows/Queries during tests
- Monitoring of exceptions in Flows/Queries during testing and reporting them
- A single continuuity-test.jar that pulls in all Continuuity dependencies required for writing tests

Writing Tests in Eclipse

1. Add *continuuity-developer-edition-1.3.0/lib/flow-1.3.0-testhelper.jar* and *JUnit 4.x* jar to the project classpath.
   a. Right click on project, and select "Build Path" > "Add External Archives..."
   b. Navigate to continuuity-developer-edition-1.3.0/lib/, and select flow-1.3.0-testhelper.jar
   c. Click "Open" to add the jar
2. Write Flow/Query tests.
3. Create a new run configuration for JUnit in Eclipse with the runtime libraries required for running the tests.
   a. Right click on project, and select "Run As" -> "Run Configurations...".
   b. Click on the JUnit tab in the left bar. Click on "New" button to create a new configuration
   c. Click on "Classpath" in the opened dialog, and select "User Entries".
   d. Click "Add External JARs...". Navigate to dir continuuity-developer-edition-1.3.0/lib/, and select all the jars in it. Click "Open", click "Apply" and "Run".
4. The tests should now start running.

Maven Dependency

Alternatively, if you do not want to use the Eclipse Plugin, you can create your project using Maven and easily declare the necessary dependency in your Maven pom file. This will pull in continuuity-test.jar and any other dependencies required for writing/running tests:

```
<dependency>

    <groupId>com.continuuity</groupId>

    <artifactId>continuuity-test</artifactId>

    <version>1.3.0</version>

    <scope>test</scope>

</dependency>
```

maven-bundle-plugin config needs to be added to pom to fetch Apache MINA jars:

```
<build>

    <plugins>

        <plugin>

            <groupId>org.apache.felix</groupId>
            <artifactId>maven-bundle-plugin</artifactId>
            <version>2.3.7</version>
            <extensions>true</extensions>
            <inherited>true</inherited>
            <configuration/>
        </plugin>
    </plugins>
</build>
```

Testing a Flow

A typical Flow test case will be similar to class FlowTest shown below:

```
public class FlowTest extends AppFabricTestBase {
  @Test(timeout = 20000)
  public void testClusterWriterFlow() throws Exception {

    // Start the flow
    TestFlowHandle flowHandle = startFlow(ClusterWriterFlow.class);
    assertTrue(flowHandle.isRunning());

    // Write data to flow
    writeToStream("<stream_name>", new String("<test_data>").getBytes());

    // Get Flowlet TestInfo object
    final ComputeFlowletTestInfo parserInfo =
        flowHandle.getComputeFlowletTestInfo("<flowlet_name>");

    // Wait for flowlet to process data
    waitForCondition(flowHandle, "Waiting for flowlet to process tuple",
        new Condition() {
          @Override
          public boolean evaluate() {
            return parserInfo.getNumProcessed() >= 1;
          }
        });

    // Verify test
    assertEquals(...);

    flowHandle.stop();  // stop flow
  }
}
```

The FlowTest class extends class AppFabricTestBase that contains methods to start and interact with in-memory Flows/Queries.

1. The flow to be tested is started in-memory using AppFabricTestBase.startFlow(FlowClass). This method returns a TestFlowHandle that can be used to interact with this flow.

2. The TestFlowHandle has methods like isSuccess(), isRunning(), stop(), etc, that can be used to know/change the state of flow.

3. Once the flow is confirmed to be running, data can be written to the stream of the flow using AppFabricTestBase.writeToStream(StreamName, data).

4. The Flowlets of the flow will now start processing the data written to the stream. The processing can be validated using FlowletTestInfo objects, which is ComputeFlowletTestInfo object in the above example.

5. ComputeFlowletTestInfo class has the following methods:

    - isInitialized(), isConfigured() and isDestroyed() that can be used to check the state of a flow.

    - getNumProcessed(), getNumSuccessful(), etc, that gives the number of tuples processed, number of tuples processed successfully, etc.

    - popExceptions() that gives the exceptions thrown by flowlets if any while processing the stream.

    - popProcessedTuples(), popSuccessfulTuples(), etc. that gives the list of tuples processed from the stream.

6. AppFabricTestBase.waitForCondition is used to wait till a flowlet has processed the stream so that test assertions can be run. The waitForCondition method also reports and fails the test if any flowlets in the flow throw exception during stream processing. It uses FlowletTestInfo.popExceptions() to fetch the exceptions.

7. After the waitForCondition method returns successfully, the test assertions can be run.

8. Finally, the flow needs to be stopped. Note that if the flow is not stopped when the test is complete, then any other tests that instantiate the same flow again may throw exceptions.

9. After all the tests in the class have been run, AppFabricTestBase checks to make sure that no exceptions have been thrown by the flowlets during the test. It fails the test if if finds any exceptions. It also stops all flows started in the test class.

Testing a Query

A Query test case is very similar to the Flow test case discussed previously:

```
public class SampleQueryTest extends AppFabricTestBase {

  @Test(timeout = 20000)
  public void testQueryProvider() throws Exception {
    // Start the query provider
    TestQueryHandle queryHandle = startQuery(HelloWorldQueryProvider.class);
    assertTrue(queryHandle.isRunning());
    // Create some data for querying
    createTestData();
    // Get Query TestInfo object
    QueryTestInfo queryTestInfo = queryHandle.getQueryTestInfo("<queryname>");
    // Submit a query, verify that it returns correct status and content
    QueryResult queryResult = runQuery(queryHandle, "method",
        ImmutableMultimap.<String, String>of("param","value"));
    Assert.assertEquals(1, queryTestInfo.getNumQueries());
    Assert.assertEquals(200, queryResult.getReturnCode());
    Assert.assertEquals("<sample output>", queryResult.getContent());
    // Other assertions
    assertEquals(...);
    queryHandle.stop(); // stop query
  }

  private void createTestData() {
    // Register the key value table used by the query provider
    // normally, this would be done by the AppFabric when the app is deployed
    registerDataSet(new KeyValueTable("simple"));
    // get a runtime instance of the Dataset to write data
    KeyValueTable kv = getDataSet("simple");
    kv.exec(new KeyValueTable.WriteKey(row, Bytes.toBytes("<sample data>")));


  }
}
```

Again Query test class extends AppFabricTestBase and behaves in a similar way.

1. The query class is started using AppFabricTestBase.startQuery(QueryProviderClass) method. This method returns TestQueryHandle that is used to interact with the started query.
2. The TestQueryHandle has methods like isSuccess(), isRunning(), stop(), etc, that can be used to check/change the state of started query.
3. QueryTestInfo object can be used to get information on the running query. It has the following methods:
    - isInitialized(), isConfigured(), isDestroyed(), etc. that can be used to check the state of the query.
    - getNumQueries() gives the number of queries it processed.
    - popExceptions() gives the exceptions thrown by the query.
    - popQueries() gives the queries that were processed.
4. Data on which the query can perform data operations is created using a DataSet.
    - The DataSet is registered using AppFabricTestBase.registerDataSet(DataSet).
    - Obtain a runtime instance of the DataSet using AppFabricTestBase.getDataSet(String). This runtime instance will be fully configured to perform data operations.
5. Once some data is stored, queries can be run using AppFabricTestBase.runQuery(TestQueryHandle, QueryMethod, Parameters).
6. runQuery returns a QueryResult object on successful execution of the query. It also fails the test if the QueryProvider throws any exceptions during query execution. It uses QueryTestInfo.popExceptions to fetch the exceptions.
7. The test assertions can be run on the returned query results.
8. Finally, the query needs to be stopped.
9. After all the tests in the class have been run, AppFabricTestBase checks to make sure that no exceptions have been thrown by query providers during the test. It fails the test if if finds any exceptions. It also stops all queries started in the test class.

5. API and Tool Reference

5.1 Java APIs

The Javadoc for all Core AppFabric Java APIs is included in the Developer Suite:

./continuuity-developer-edition-1.3.0/docs/javadoc/index.html

5.2 REST APIs

The Continuuity AppFabric Platform currently exposes three REST interfaces:

1. **Stream**: To send data events to a stream or to inspect the contents of a stream.
2. **Query**: To execute a query.
3. **Data**: To read, write, or delete a single data value that was persisted to the Data Fabric by user application. It also offers a way to return the Data Fabric to an empty state (clear the fabric).

Common return codes for all REST calls:

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

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

REST Endpoint Port Configuration

By default, each of the three REST interfaces binds to a set of default ports on localhost. You can modify these ports by modifying your *continuuity-site.xml* within your Developer Suite conf directory. The three properties you will modify are:

Description	Property	Default Value
Stream REST Port	*colletor.rest.port*	10000
Data REST Port	*accessor.rest.port*	10002
Query REST Port	*query.rest.port*	10003

The descriptions below will assume the default ports.

Stream REST API

This interface allows sending events to a stream and reading single events from a stream.

Sending Events

A stream send request is implemented as an HTTP post. The URI is formed as

```
http://<hostname>:10000/rest-stream/<stream_name>
```

where the stream name is the name of an existing stream. The body of the request must contain the event in binary form.

Additional HTTP headers recognized are

```
<stream_name>.<property_name> : <string_value>
```

After receiving the request, the REST connector will transform it into an 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 name, as in <stream_name>.<property_name>, then the event will also have these headers but only with the property name, the stream name prefix will be stripped.

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 (e.g. multiple flows), each of which may be a group of different agents (e.g. multiple instances of a flowlet). In order to read, the client must first obtain a consumer (group) id, which needs to be passed to subsequent read requests.

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

```
http://<hostname>:10000/rest-stream/<stream_name>?q=newConsumer
```

The consumer id is returned in a specific header. For convenience the body of the response also contains this ID

com.continuuity.stream.consumer: <consumer_id>

Return codes:

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

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

```
http://<hostname>:<port>/rest-stream/<stream_name>?q=dequeue
```

and the request must have a header of the form

```
        com.continuuity.stream.consumer: <consumer_id>
```

in order to pass in the consumer id. The response will contain the binary body of the event in its body and a header

```
        <stream_name>.<property> : <value>
```

for each header named <property> of the event. This is analogous to the way that headers are posted with events.

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. See the command-line guide in Section 5.3.

Query REST API

This interface allows invoking queries.

Executing Query

To execute a query

```
        http://<hostname>:10003/rest-query
              /<query_provider_name>/<query_method_name>?<query_arguments>
```

Return codes:

- *200 OK*: Everything went well. Body will contain query result.
- *204 No Content*: The query exists but query result is empty.
- *404 Not found*: The query does not exist.

Data REST API

The data API allows writing, reading and deleting keys in the data fabric, as well as formatting the data fabric (clearing it of all data).

Reading the Value for a Key

A point read request is an HTTP GET request to the following URI

```
        http://<hostname>:10002/rest-data/<table_name>/<key>
```

where the key is the URL-encoded binary key of the value to be retrieved.

The gateway will read from the named table and retrieve the value persisted for that key. The value will be returned in binary form in the response body.

Return codes:

- *200 OK*: Everything went well. The value is returned with the response.
- *404 Not Found*: The table was not found, or an entry for the key does not exist in the table.

Writing a New Value for a Key

A write request is an HTTP PUT request to the URI

```
http://<hostname>:10002/rest-data/<table_name>/<key>
```

where the request body contains the binary value to be written for this key.

Return codes:

- *200 OK*: Everything went well.
- *404 Not Found*: The table was not found.

Deleting a Key

A delete request is an HTTP DELETE request to the URI

```
http://<hostname>:10002/rest-data/<table_name>/<key>
```

Return codes:

- *200 OK*: Everything went well.
- *404 Not Found*: The table was not found.

Listing the Keys in a Table

It is useful to see what keys have been written to a table. This API lists all keys in the table in the body of the response, one per line.

- Because a table can be large, the API limits the number of keys returned to the first 100 by default. A different limit can be specified in the request. The request can also specify to start with the Nth key.
- Because keys are binary, the API uses URL-encoding to convert them into printable strings. A different encoding can be specified by the request.

The request URL for listing keys is

```
http://<hostname>:10002/rest-data/<table_name>?q=list&<other_parameters>
```

Valid parameters are

- enc=url to use URL encoding for keys
- enc=hex to use Hexadecimal notation for keys
- enc=<encoding> to specify an encoding for converting bytes into Strings. The encoding must be supported by standard Java 1.6.
- limit=<n> to request n keys to be listed.
- start=<i> to request starting with the ith key.

Return codes:

- *200 OK*: Everything went well. The response lists all keys, one per line.

- *404 Not Found*: The table was not found.

Data Cleanup

Caution, this will delete all data specified. A clear request is issued as an HTTP GET to the URL:

```
http://<hostname>:10002/rest-data/?clear=<types_to_clear>
```

where <types_to_clear> is a comma-separated list of one or more of the following types:

- *tables* to remove all tables
- *streams* to remove all event streams
- *queues* to remove all intra-flow tuple streams
- *all* for all of the above three.

Return codes:

- *200 OK*: Everything went well.

5.3 Command Line Tools

The Developer Suite includes a suite of tools that allow you to access and manage local and remote AppFabric 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 Developer Suite (e.g. ~/continuuity-developer-suite-1.3.0/bin/)

AppFabric

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

```
> continuuity-app-fabric start
> continuuity-app-fabric stop
```

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

```
> continuuity-app-fabric --help
```

Data Client

The data client is a command line tool to access persisted data in the Continuuity Data Fabric. However, today you should use QueryProviders to query your data from outside the system, and the data client tool is primarily limited to performing data cleanup operations, as defined below:

```
> data-client clear --all
> data-client clear --data
> data-client clear --queues
> data-client clear --streams
> data-client clear --tables
> data-client clear --meta
```

Caution: Once you clear the Data Fabric, its contents cannot be restored.

Flow Client

The flow client command line tool can be used to deploy and manage flows and queries. The execution of the examples in this section requires that the AppFabric is running and an application with the name “myapp” has been created through the Dashboard.

To deploy a new flow in the “myapp” application

```
> flow-client deploy --resource CountTokens.jar --application myapp
```

The type can be omitted since flow is the default type.

To deploy a new query in the “myapp” application:

```
> flow-client deploy --resource MyQuery.jar --application myapp --type QUERY
```

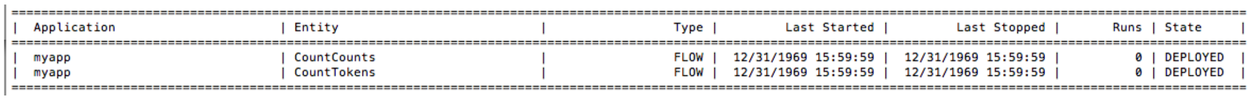
If successfully deployed, the tool will confirm it by printing

```
Successfully deployed.
```

After new flows and queries got deployed you can lookup their entity ids by issuing the list command:

```
> flow-client list
```

The output of the command lists all the deployed entities of both types, flows and queries:



```
===================================================================================================================================================
| Application                          | Entity                               |                  Type |         Last Started |         Last Stopped |       Runs | State      |
===================================================================================================================================================
| myapp                                | CountCounts                          |                  FLOW |  12/31/1969 15:59:59 |  12/31/1969 15:59:59 |          0 | DEPLOYED   |
| myapp                                | CountTokens                          |                  FLOW |  12/31/1969 15:59:59 |  12/31/1969 15:59:59 |          0 | DEPLOYED   |
===================================================================================================================================================
```

Flows can be started, stopped or deleted by the following commands:

```
> flow-client start --application myapp --entity CountTokens

> flow-client stop --application myapp --entity CountTokens

> flow-client remove --application myapp --entity CountTokens
```

Stream Client

The stream-client is a utility to send events to a stream or to view the current content of a stream. The tool only supports event stream, not intra-flow queues which contain tuples. To send a single event to a stream:

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

The send command supports adding multiple headers:

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

You can use hexadecimal notation or URL-encoding to specify the body of an event (default is URL-encoding). For convenience the body can also be read from a binary file.

All or ranges of the current events of a stream can be inspected through the view command:

```
> stream-client view —-stream msgstream --last 50
```

prints the last 50 events that were added to the stream “msgstream”.

5.4 Eclipse IDE Plugin

This section walks through creating, running, and debugging a simple application using the Continuuity Eclipse IDE plugin.

Prerequisites

- Eclipse Juno (available at http://eclipse.org/downloads/)
- The Continuuity Developer Suite (available from http://www.continuuity.com/download/)
- The Continuuity Eclipse Plugin (packaged with the Developer Suite as a jar file)

    continuuity-eclipse-plugin-1.0.0.jar

Getting Eclipse and Setting up Plugin

1. Download and unpack the Eclipse IDE.
2. Install the Continuuity Eclipse plugin by copying the plugin jar into the "plugins" subdirectory of the unpacked Eclipse directory. For example:

```
> cp ~/continuuity-developer-edition-1.3.0/plugins/continuuity-eclipse-plugin-
1.0.0.jar /Applications/eclipse/plugins/
```

3. Start Eclipse.

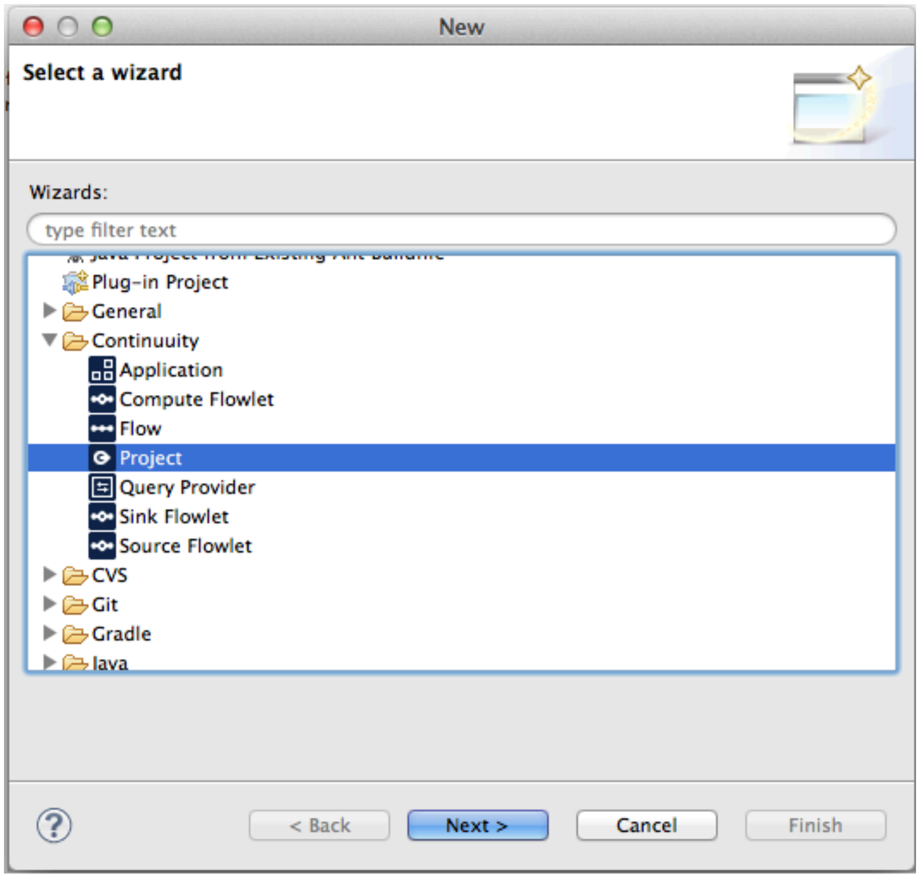
Creating a Simple Application

This section guides you through creating a simple application from scratch in Eclipse IDE.

Create a New Continuuity Project

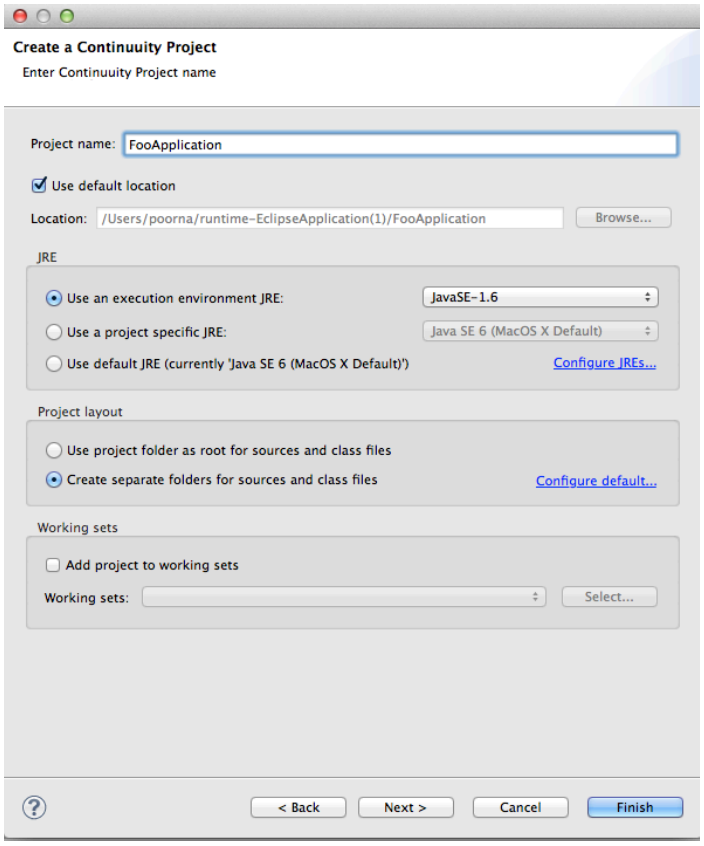
1. Create a new project by following the path

   File -> New -> Other... -> Continuuity -> Project



NOTE: if you don't see the "Continuuity" menu item in this dialog, the plugin was not installed correctly. Please refer to the first section of the document for the installation steps.

2. Enter the name of your application, e.g. "FooApplication" and click Next.



Project name: FooApplication

[✓] Use default location

Location: /Users/poorna/runtime-EclipseApplication(1)/FooApplication

JRE

(●) Use an execution environment JRE: JavaSE-1.6

( ) Use a project specific JRE: Java SE 6 (MacOS X Default)

( ) Use default JRE (currently 'Java SE 6 (MacOS X Default)')

Project layout

( ) Use project folder as root for sources and class files

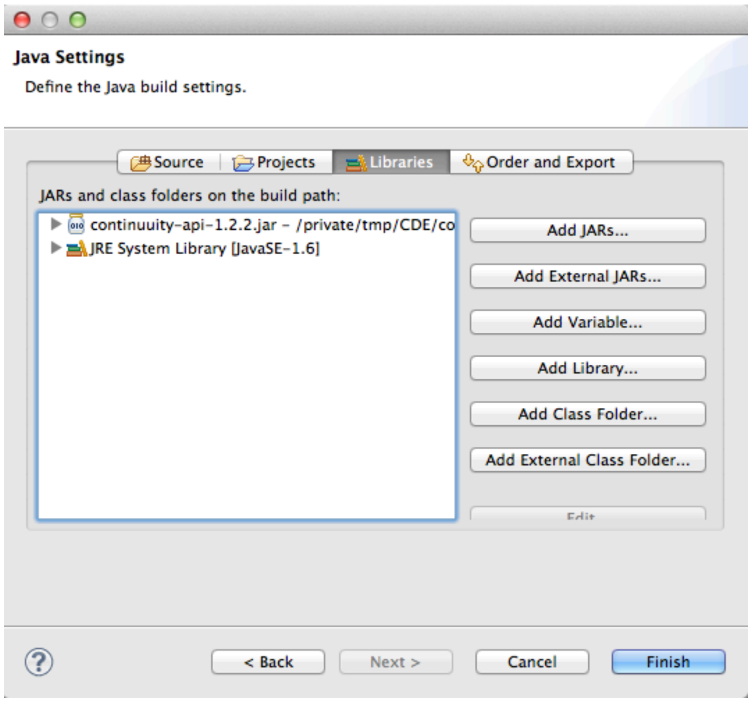
(●) Create separate folders for sources and class files

Working sets

[ ] Add project to working sets

Working sets:

3. Add continuuity-api.jar into the class path. Use the standard "Add External JARs..." dialog on the "Libraries" tab to do this.



JARs and class folders on the build path:

- continuuity-api-1.2.2.jar - /private/tmp/CDE/co
- JRE System Library [JavaSE-1.6]

Create Flow

The simple application we are building will contain a simple Flow. The Flow will contain a single ComputeFlowlet that consumes data from a Stream.

Add Flow And Flowlet Classes

To quickly create a Flow class stub:

1. Use the Eclipse File menu. Go to:

   File -> New -> Other... -> Continuuity -> Flow

2. In the opened dialog, enter package and class names. E.g. package "my" and class name "MyFlow"

To quickly create a ComputeFlowlet class stub:

1. Use the Eclipse File menu. Go to:

   New -> Other... -> Continuuity -> ComputeFlowlet

2. In the opened dialog, enter package and class names. E.g. package "my" and class name "MyComputeFlowlet"

Configure Flow

The flow will consist of one Flowlet that consumes data from a Stream and outputs consumed data to STDOUT.

First, we need to configure the Flow.

Go to my/MyFlow.java in the editor and add to *configure()* method (in addition to generated code):

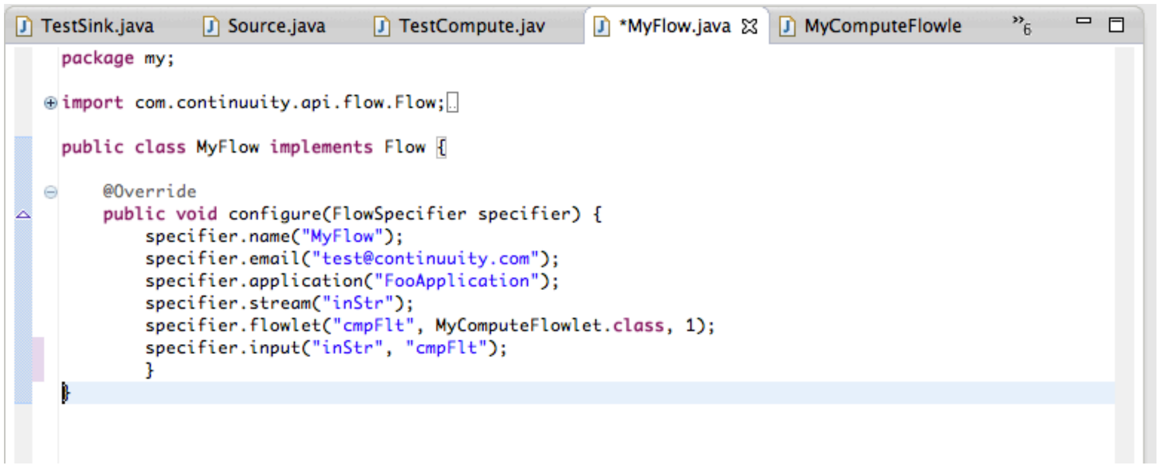
```
specifier.stream("inStr");
specifier.flowlet("cmpFlt", MyComputeFlowlet.class, 1);
specifier.input("inStr", "cmpFlt");
```

- “inStr” is the name of the stream that is consumed by the Flowlet.
- “cmpFlt” is the name of the Flowlet

  NOTE: you can use any names instead of the ones in the example.

The first line defines a Stream that our Flowlet will consume from, while the second line defines the ComputeFlowlet. This creates one instance of the Flowlet. You can create multiple instances to parallelize the work. The third line connects the Stream and the Flowlet.

Below is an example of the class content you should have after doing this:



```
package my;

import com.continuuity.api.flow.Flow;

public class MyFlow implements Flow {

    @Override
    public void configure(FlowSpecifier specifier) {
        specifier.name("MyFlow");
        specifier.email("test@continuuity.com");
        specifier.application("FooApplication");
        specifier.stream("inStr");
        specifier.flowlet("cmpFlt", MyComputeFlowlet.class, 1);
        specifier.input("inStr", "cmpFlt");
        }
}
```

Configure ComputeFlowlet and Add Processing Logic

The Flowlet will consume tuples from the Stream and outputs information about each one into STDOUT. First we should define the schema of the input and then add the tuple processing logic.

To do this, go to my/MyComputeFlowlet.java in the editor and define the input schema by adding the following into *configure()*:

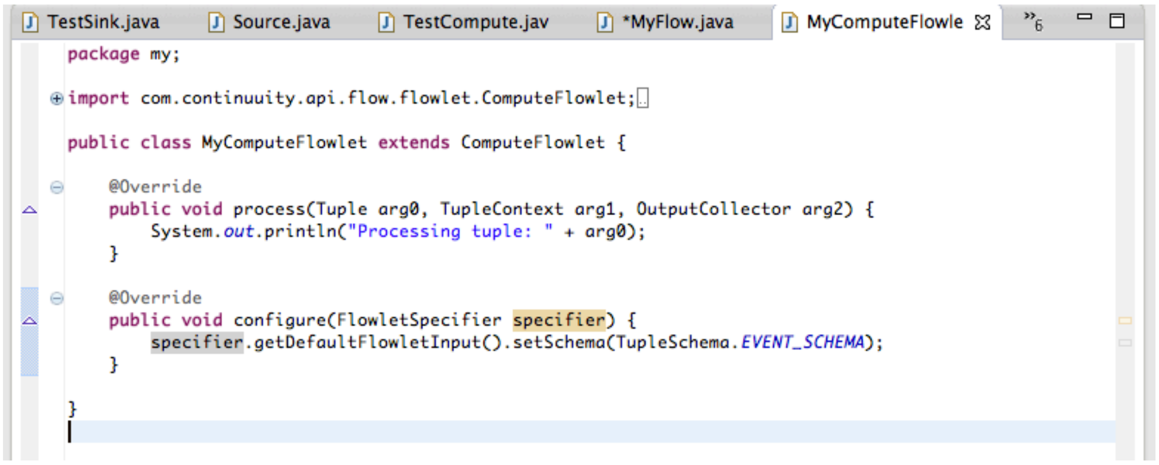
```
specifier.getDefaultFlowletInput().setSchema(TupleSchema.EVENT_SCHEMA);
```

More information on how to configure the input/output schema of a flowlet during its creation time is available in section on configuring Flowlet schemas at the end.

Then insert simple processing logic into process(), e.g.:

```
System.out.println("Processing tuple: " + arg0);
```

Below is an example of the class contents you should have after adding this:



```
package my;

import com.continuuity.api.flow.flowlet.ComputeFlowlet;

public class MyComputeFlowlet extends ComputeFlowlet {

    @Override
    public void process(Tuple arg0, TupleContext arg1, OutputCollector arg2) {
        System.out.println("Processing tuple: " + arg0);
    }

    @Override
    public void configure(FlowletSpecifier specifier) {
        specifier.getDefaultFlowletInput().setSchema(TupleSchema.EVENT_SCHEMA);
    }

}
```

Create Application

Finally, create an application that ties all of the flowlets together. To create an Application, use the Eclipse File menu. Go to:

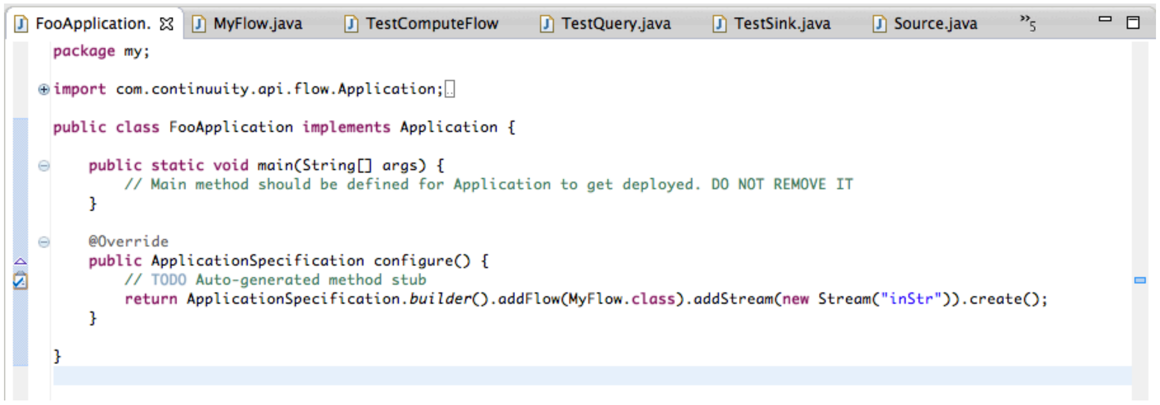
File -> New -> Other... -> Continuuity -> Application

In the opened dialog enter the package and class names, e.g. package "my" and class name "FooApplication".

Configure Application

Add the following line to the configure() method in the just-created Application class –

```
return ApplicationSpecification.builder().addFlow(Flow1.class).addStream(new
Stream("inStr")).create();
```



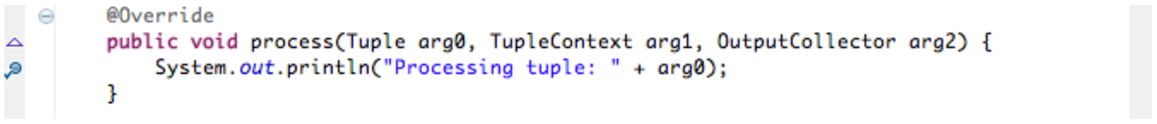
```
package my;

import com.continuuity.api.flow.Application;

public class FooApplication implements Application {

    public static void main(String[] args) {
        // Main method should be defined for Application to get deployed. DO NOT REMOVE IT
    }

    @Override
    public ApplicationSpecification configure() {
        // TODO Auto-generated method stub
        return ApplicationSpecification.builder().addFlow(MyFlow.class).addStream(new Stream("inStr")).create();
    }

}
```

Running and Debugging Application

This section guides through running and debugging steps for the application created above.

Set Breakpoint

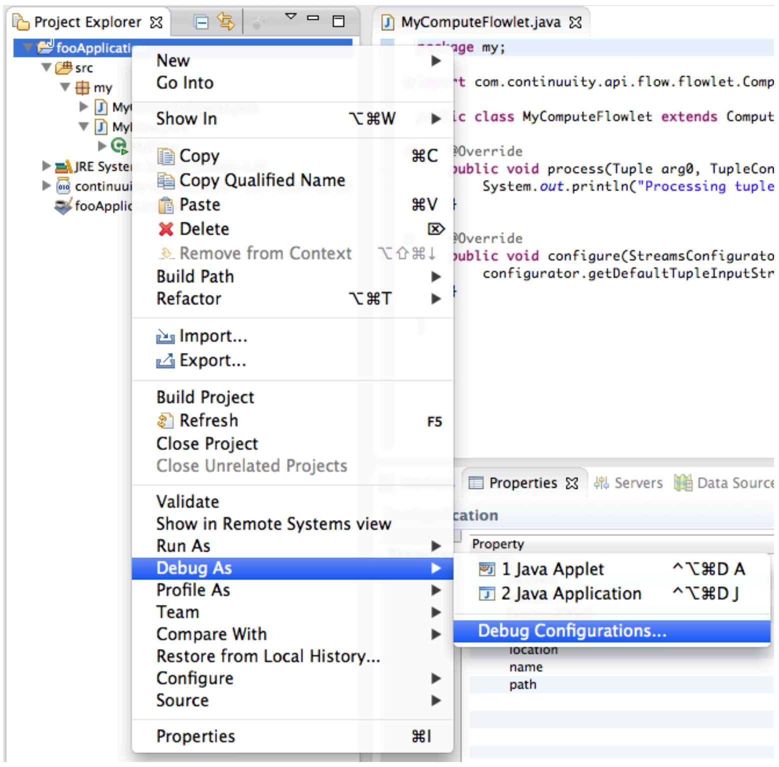
We can set breakpoints anywhere in application code before debugging. E.g. it may be useful to inspect execution of tuples processing logic by adding a breakpoint in MyComputeFlowlet.process() method:



```
@Override
public void process(Tuple arg0, TupleContext arg1, OutputCollector arg2) {
    System.out.println("Processing tuple: " + arg0);
}
```

Create Launch Configuration

Right-click anywhere on the project in the Project Explorer (usually the left panel), choose Debug As -> Debug Configurations from the popup menu.

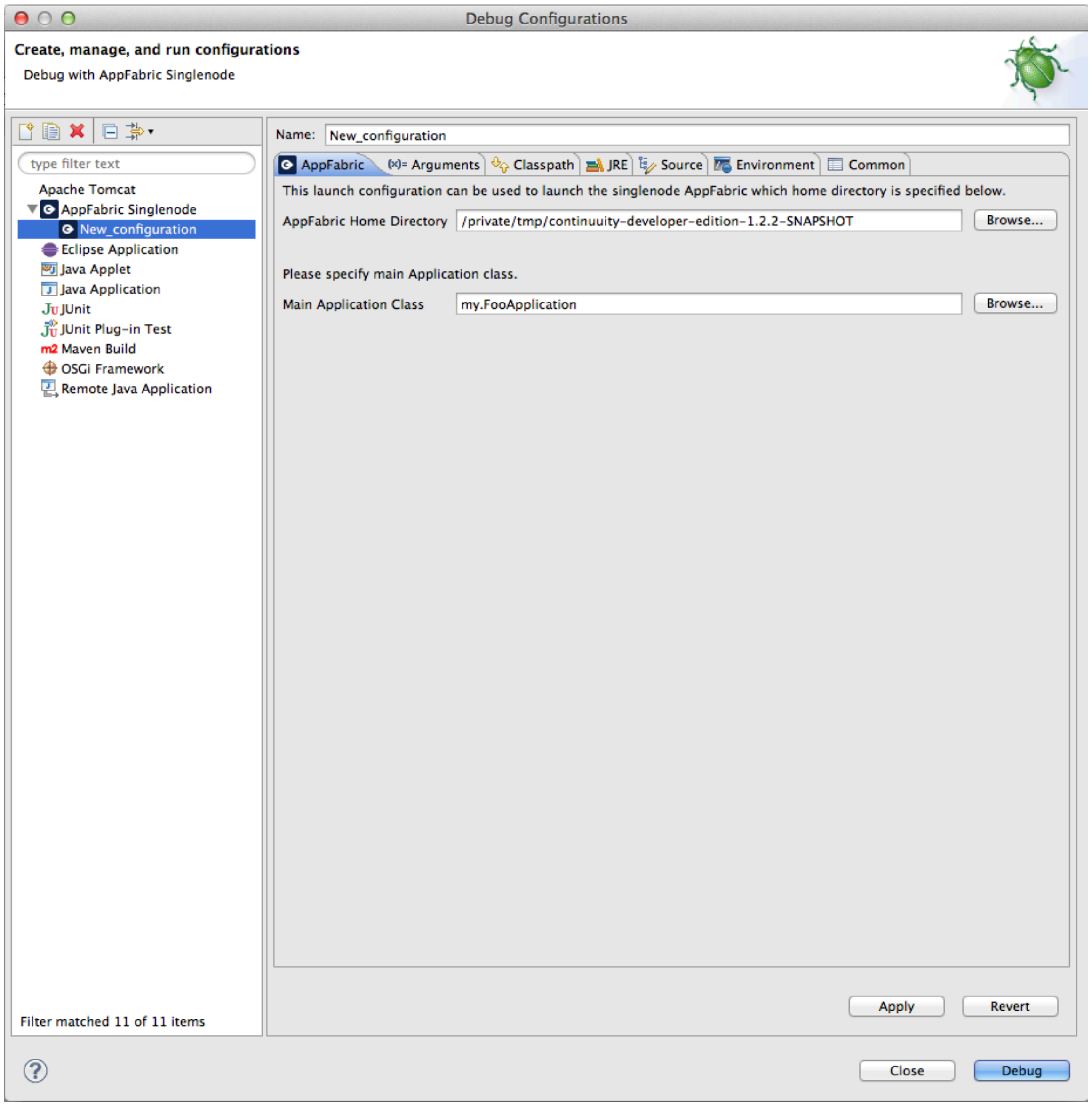


```
ckage my;

t com.continuuity.api.flow.flowlet.Comp

ic class MyComputeFlowlet extends Comput

@Override
ublic void process(Tuple arg0, TupleCon
    System.out.println("Processing tuple
}

@Override
ublic void configure(StreamsConfigurato
    configurator.getDefaultTupleInputStr
}
```

Create a configuration for the "AppFabric Singlenode" top-level entry by right-clicking on it or by clicking the leftmost icon above the configurations list.

Optionally, set the Name of the launch configuration (e.g. "FooApplication"), since it will be used to run this specific project.

On the first tab of the configuration, set the AppFabric Home Directory to the SDK home and select the Main Application class (e.g. "my.FooApplication"). The class selection dialog autocomplete feature will help you once you start typing the class name.



Name: New_configuration

This launch configuration can be used to launch the singlenode AppFabric which home directory is specified below.

AppFabric Home Directory /private/tmp/continuuity-developer-edition-1.2.2-SNAPSHOT

Please specify main Application class.

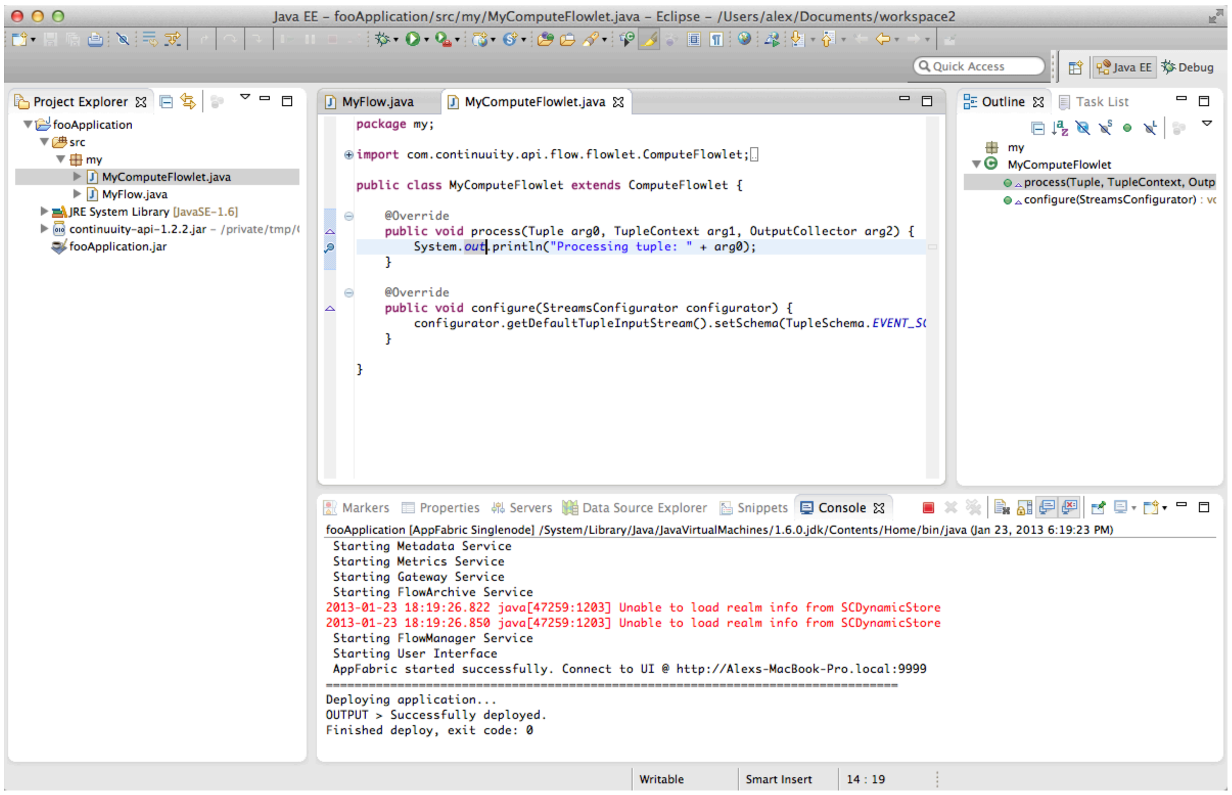
Main Application Class my.FooApplication

Click Apply to save configuration.

Start Debug

To start a debug session click Debug and you will see the output of the launched AppFabric in the Eclipse console. Wait for AppFabric to start and for the application to be deployed.

The last message in console that you should see is "Finished deploy, exit code: 0".



```java
package my;

import com.continuuity.api.flow.flowlet.ComputeFlowlet;

public class MyComputeFlowlet extends ComputeFlowlet {

    @Override
    public void process(Tuple arg0, TupleContext arg1, OutputCollector arg2) {
        System.out.println("Processing tuple: " + arg0);
    }

    @Override
    public void configure(StreamsConfigurator configurator) {
        configurator.getDefaultTupleInputStream().setSchema(TupleSchema.EVENT_SC
    }

}
```

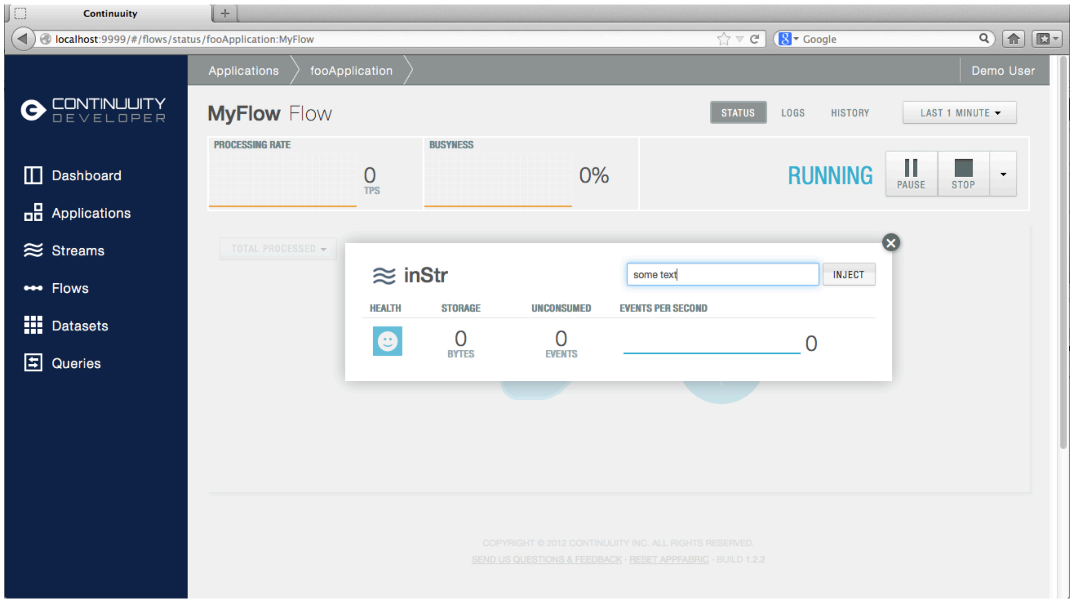
```
fooApplication [AppFabric Singlenode] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Jan 23, 2013 6:19:23 PM)
 Starting Metadata Service
 Starting Metrics Service
 Starting Gateway Service
 Starting FlowArchive Service
2013-01-23 18:19:26.822 java[47259:1203] Unable to load realm info from SCDynamicStore
2013-01-23 18:19:26.850 java[47259:1203] Unable to load realm info from SCDynamicStore
 Starting FlowManager Service
 Starting User Interface
 AppFabric started successfully. Connect to UI @ http://Alexs-MacBook-Pro.local:9999
================================================================================
Deploying application...
OUTPUT > Successfully deployed.
Finished deploy, exit code: 0
```

Work with Application

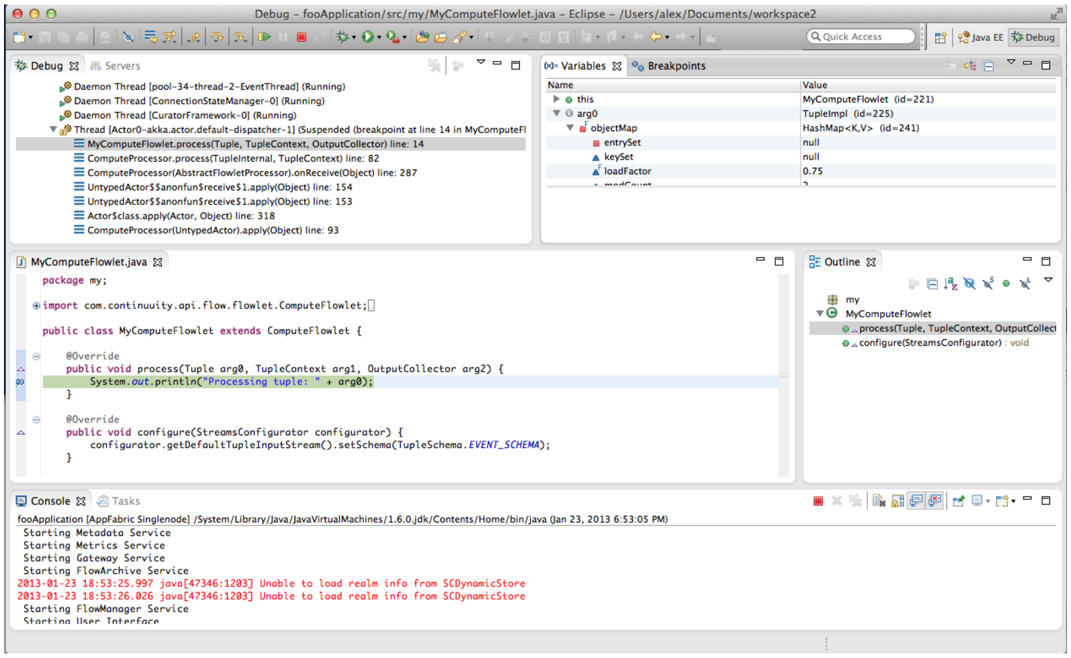
Go to "http://localhost:9999" in your web browser. Navigate to Flows using the menu on the left. Note the "DEPLOYED" status of MyFlow we just deployed. Navigate to the Flow.

Start the Flow.

To inject data into the Stream, click on "INSTR", enter some text and click "INJECT".



Eclipse will "catch" a breakpoint in MyComputeFlowlet.process() method.


Debug – fooApplication/src/my/MyComputeFlowlet.java – Eclipse – /Users/alex/Documents/workspace2
Quick Access
Java EE
Debug
Servers
Variables
Breakpoints
Outline
my
MyComputeFlowlet
process(Tuple, TupleContext, OutputCollect
configure(StreamsConfigurator) : void
Tasks

```
Daemon Thread [pool-34-thread-2-EventThread] (Running)
Daemon Thread [ConnectionStateManager-0] (Running)
Daemon Thread [CuratorFramework-0] (Running)
Thread [Actor0-akka.actor.default-dispatcher-1] (Suspended (breakpoint at line 14 in MyComputeFl
    MyComputeFlowlet.process(Tuple, TupleContext, OutputCollector) line: 14
    ComputeProcessor.process(TupleInternal, TupleContext) line: 82
    ComputeProcessor(AbstractFlowletProcessor).onReceive(Object) line: 287
    UntypedActor$$anonfun$receive$1.apply(Object) line: 154
    UntypedActor$$anonfun$receive$1.apply(Object) line: 153
    Actor$class.apply(Actor, Object) line: 318
    ComputeProcessor(UntypedActor).apply(Object) line: 93
```

Name	Value
this	MyComputeFlowlet (id=221)
arg0	TupleImpl (id=225)
objectMap	HashMap<K,V> (id=241)
entrySet	null
keySet	null
loadFactor	0.75

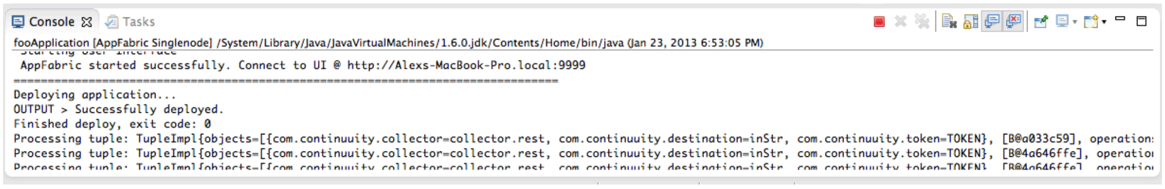
MyComputeFlowlet.java

```java
package my;

import com.continuuity.api.flow.flowlet.ComputeFlowlet;

public class MyComputeFlowlet extends ComputeFlowlet {

    @Override
    public void process(Tuple arg0, TupleContext arg1, OutputCollector arg2) {
        System.out.println("Processing tuple: " + arg0);
    }

    @Override
    public void configure(StreamsConfigurator configurator) {
        configurator.getDefaultTupleInputStream().setSchema(TupleSchema.EVENT_SCHEMA);
    }
```

Console

```
fooApplication [AppFabric Singlenode] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Jan 23, 2013 6:53:05 PM)
 Starting Metadata Service
 Starting Metrics Service
 Starting Gateway Service
 Starting FlowArchive Service
2013-01-23 18:53:25.997 java[47346:1203] Unable to load realm info from SCDynamicStore
2013-01-23 18:53:26.026 java[47346:1203] Unable to load realm info from SCDynamicStore
 Starting FlowManager Service
 Starting User Interface
```

Note that the output that was written to STDOUT by ComputeFlowlet is visible in the Eclipse console.



```
fooApplication [AppFabric Singlenode] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Jan 23, 2013 6:53:05 PM)
 AppFabric started successfully. Connect to UI @ http://Alexs-MacBook-Pro.local:9999
================================================================================
Deploying application...
OUTPUT > Successfully deployed.
Finished deploy, exit code: 0
Processing tuple: TupleImpl{objects=[{com.continuuity.collector=collector.rest, com.continuuity.destination=inStr, com.continuuity.token=TOKEN}, [B@a033c59], operation:
Processing tuple: TupleImpl{objects=[{com.continuuity.collector=collector.rest, com.continuuity.destination=inStr, com.continuuity.token=TOKEN}, [B@4a646ffe], operatio
```

Stop Debug

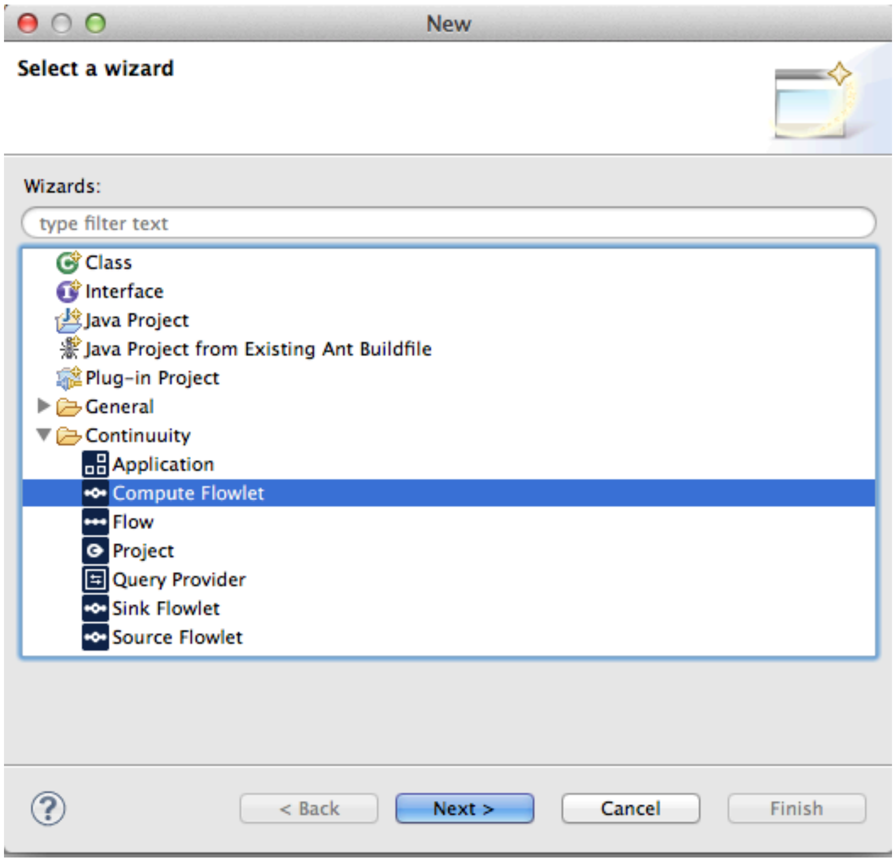
The Debug session can be stopped in a standard way, e.g. clicking the Terminate button in the top menu icons panel. This will stop AppFabric process.

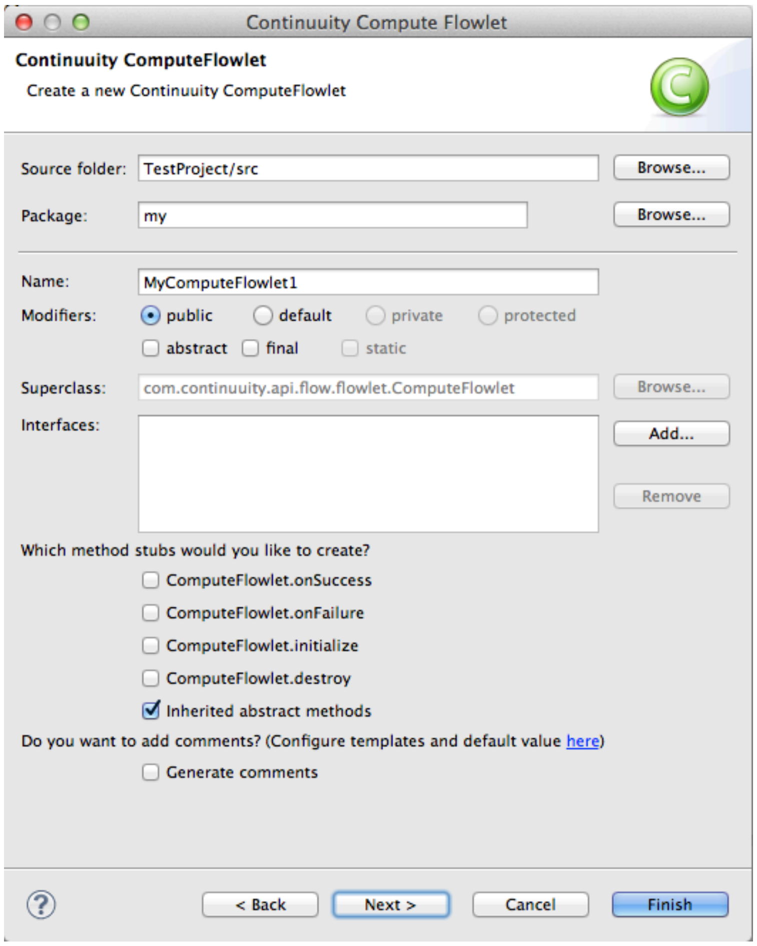
Run Mode

The steps for running an application are the same as for debugging it. Select the “Run As” menu item instead of “Debug As”.

Configuring Flowlet Schema

The Flowlet creation wizards can be used to configure the Flowlet input/outut schema while creating Flowlets. Below are the steps for the ComputeFlowlet. The steps are similar for SourceFlowlet and SinkFlowlet.





Continuuity Compute Flowlet

Continuuity ComputeFlowlet

Create a new Continuuity ComputeFlowlet

Source folder: TestProject/src

Package: my

Name: MyComputeFlowlet1

Modifiers: [✓ public] default private protected

abstract final static

Superclass: com.continuuity.api.flow.flowlet.ComputeFlowlet

Interfaces:

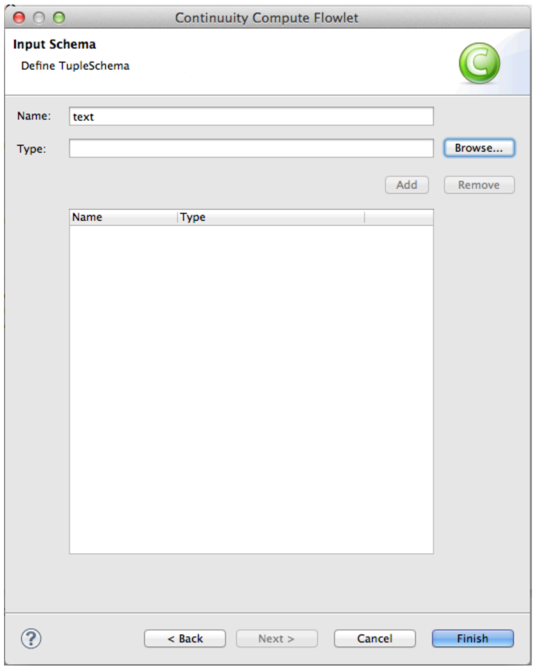
Which method stubs would you like to create?

- [ ] ComputeFlowlet.onSuccess
- [ ] ComputeFlowlet.onFailure
- [ ] ComputeFlowlet.initialize
- [ ] ComputeFlowlet.destroy
- [x] Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

- [ ] Generate comments

Enter the Flowlet name and click "Next >" to define the Input Schema.



Continuuity Compute Flowlet

Input Schema

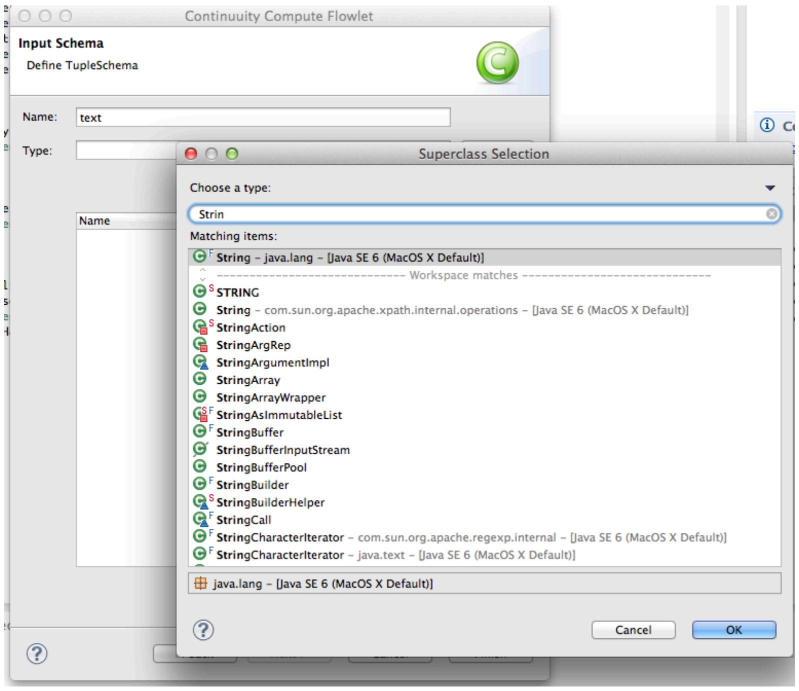
Define TupleSchema

Name: text

Type:

Name	Type

Enter a name for the Tuple field. Click on "Browse..." to select the Tuple field class.


Continuity Compute Flowlet

Input Schema

Define TupleSchema

Name: text

Type:

Superclass Selection

Choose a type:

Strin

Matching items:

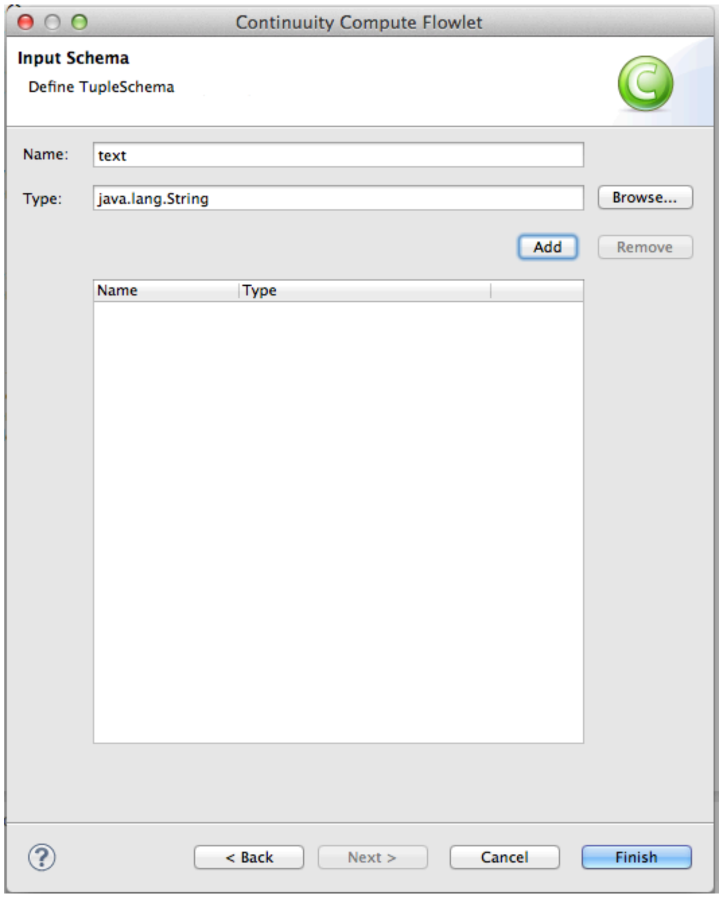
String – java.lang – [Java SE 6 (MacOS X Default)]

Workspace matches

STRING
String – com.sun.org.apache.xpath.internal.operations – [Java SE 6 (MacOS X Default)]
StringAction
StringArgRep
StringArgumentImpl
StringArray
StringArrayWrapper
StringAsImmutableList
StringBuffer
StringBufferInputStream
StringBufferPool
StringBuilder
StringBuilderHelper
StringCall
StringCharacterIterator – com.sun.org.apache.regexp.internal – [Java SE 6 (MacOS X Default)]
StringCharacterIterator – java.text – [Java SE 6 (MacOS X Default)]

java.lang – [Java SE 6 (MacOS X Default)]

Use autocomplete to select a class for the Tuple field. Click "Ok".


Continuuity Compute Flowlet

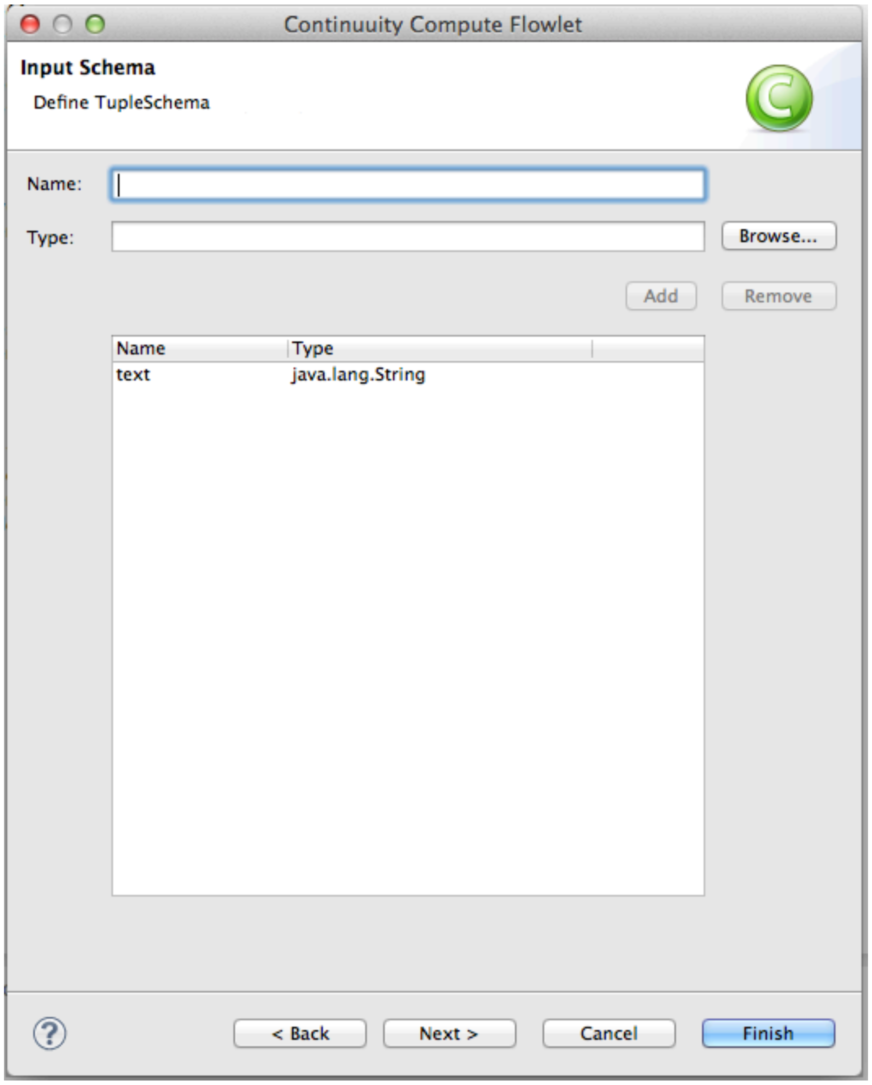
Input Schema

Define TupleSchema

Name: text

Type: java.lang.String

Click "Add" to add the field to the Input Schema.



Name	Type
text	java.lang.String

More fields can be added to the Input Schema. After defining the Input Schema is completed, click "Next >" to define the Output Schema. Finally, click "Finish" to generate the Flowlet with the schema configured.

6. APPENDIX

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