Types of Flowlets

Module Objectives

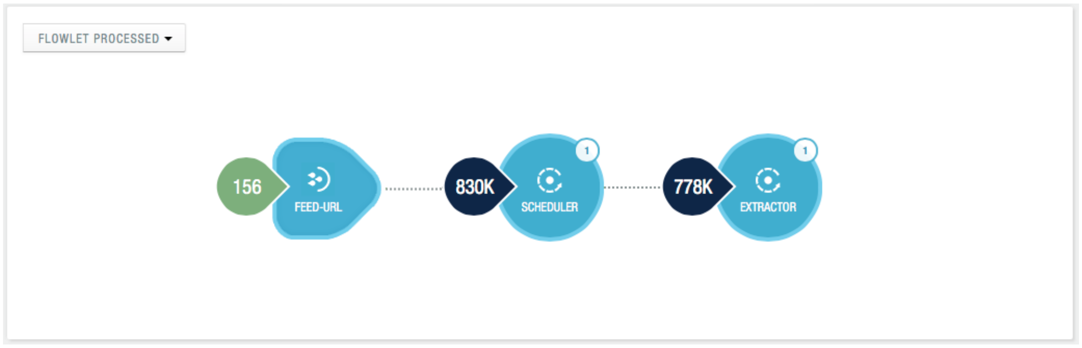
In this module, you will look at:

- Flowlets
- Flowlet initialization
- Processing with Flowlets
- Flowlet input context
- Flowlet annotations

Flowlets

Flowlets are the basic building blocks of a Flow

- Represent individual processing nodes within a Flow
- Consume data objects from their inputs
- Execute custom logic on each data object
- Perform data operations as well as emit data objects to their outputs



Flowlet Example 1/2

- Reads *Double* values, rounds them, and emits the results
- Simple configuration method, neither initialization nor destruction

```
class RoundingFlowlet implements Flowlet {

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

Flowlet Example 2/2

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

@Override
public void destroy() {
}

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

Flowlet Initialization

- Flowlets specify an `initialize()` method
- Executed at the startup of each instance of a Flowlet
- Before it receives any data

Flowlet Processing

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

- `round()`, the method that does the actual processing
- Uses an output emitter to send data to its output
- The only way that a Flowlet can emit output to another connected Flowlet
- Flowlet declares the output emitter but does not initialize it
- The Flow system initializes and injects its implementation at runtime
- Method is annotated with `@ProcessInput`
- Tells the Flow system that this method can process input data

Overloading Flowlet Processing

Overload the process method of a Flowlet by adding multiple methods with different input types

When an input object comes in, the Flowlet will call the method matching the object’s type

A method will be selected based on

- The input object’s origin
- The name of a Stream
- The name of an output of a Flowlet

Overloading Flowlet Processing Example

```
OutputEmitter<Long> output;

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

Method Selection for Flowlets Emitting Data

- Flowlets that emit data can specify a name using an annotation on the output emitter
- In the absence of annotation, the name of the output defaults to "out"

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

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

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

Flowlet Input Context

A process method can have an additional parameter, the `InputContext`

Provides information about the input object:

- Its origin
- The number of times the object has been retried

This next example is a Flowlet that tokenizes text in a smart way and uses the input context to decide which tokenizer to use

---

Flowlet Input Context Example

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

Flowlet Stream Event

Special type of object that comes in via Streams:

- Set of headers represented by a map from String to String; and
- Byte array as the body of the event

To consume a Stream with a Flow, define a Flowlet that processes data of type `StreamEvent`:

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

Flowlet Method and @Tick Annotation

- A Flowlet's method can be annotated with `@Tick`
- Instead of processing data objects from a Flowlet input, the method is invoked periodically, without arguments
- Can be used to generate data
- Can be used to pull data from an external data source periodically on a fixed cadence
- Known as a **Generative Flowlet**

Flowlet Method and @Tick Annotation Example

This `@Tick` method in the Flowlet emits random numbers:

```
public class RandomSource extends AbstractFlowlet {

  private OutputEmitter<Integer> randomOutput;

  private final Random random = new Random();

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

from the Continuuity Reactor CountRandom *example*

Module Summary

You should now be able to:

- Define what a Flowlet is
- Write simple Flowlets using:
    - overloading;
    - method selection; and
    - annotation

Module Completed