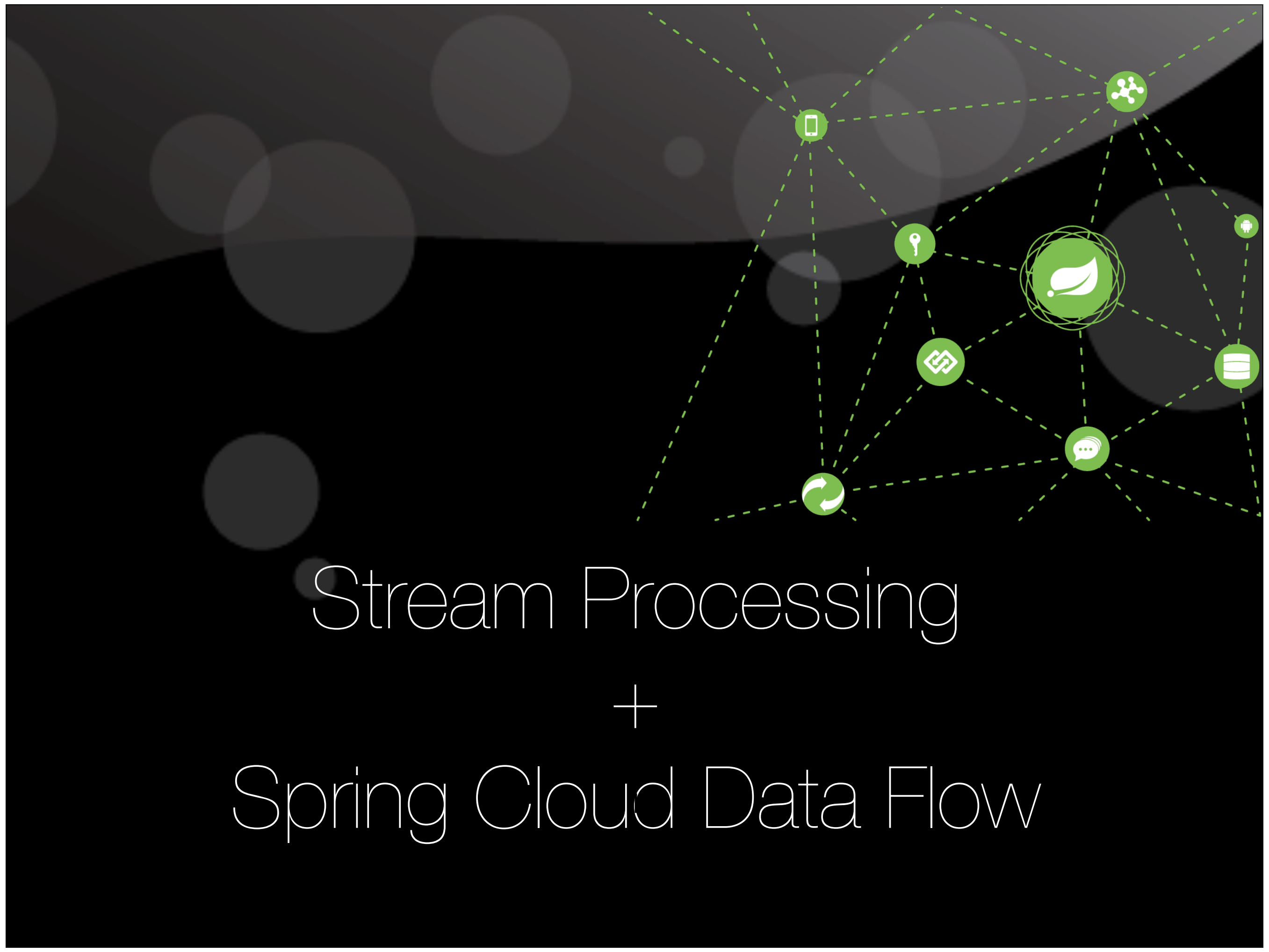
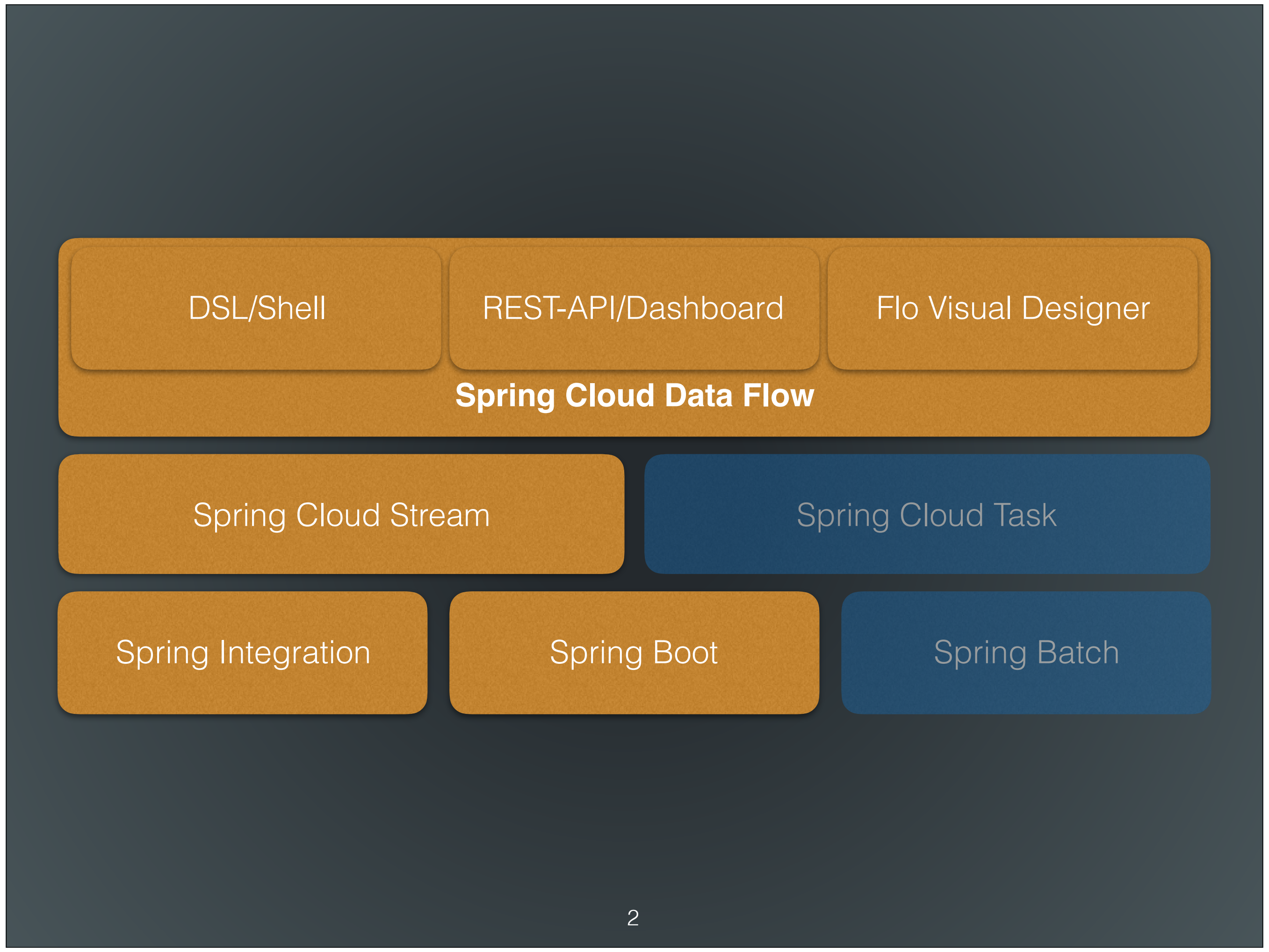


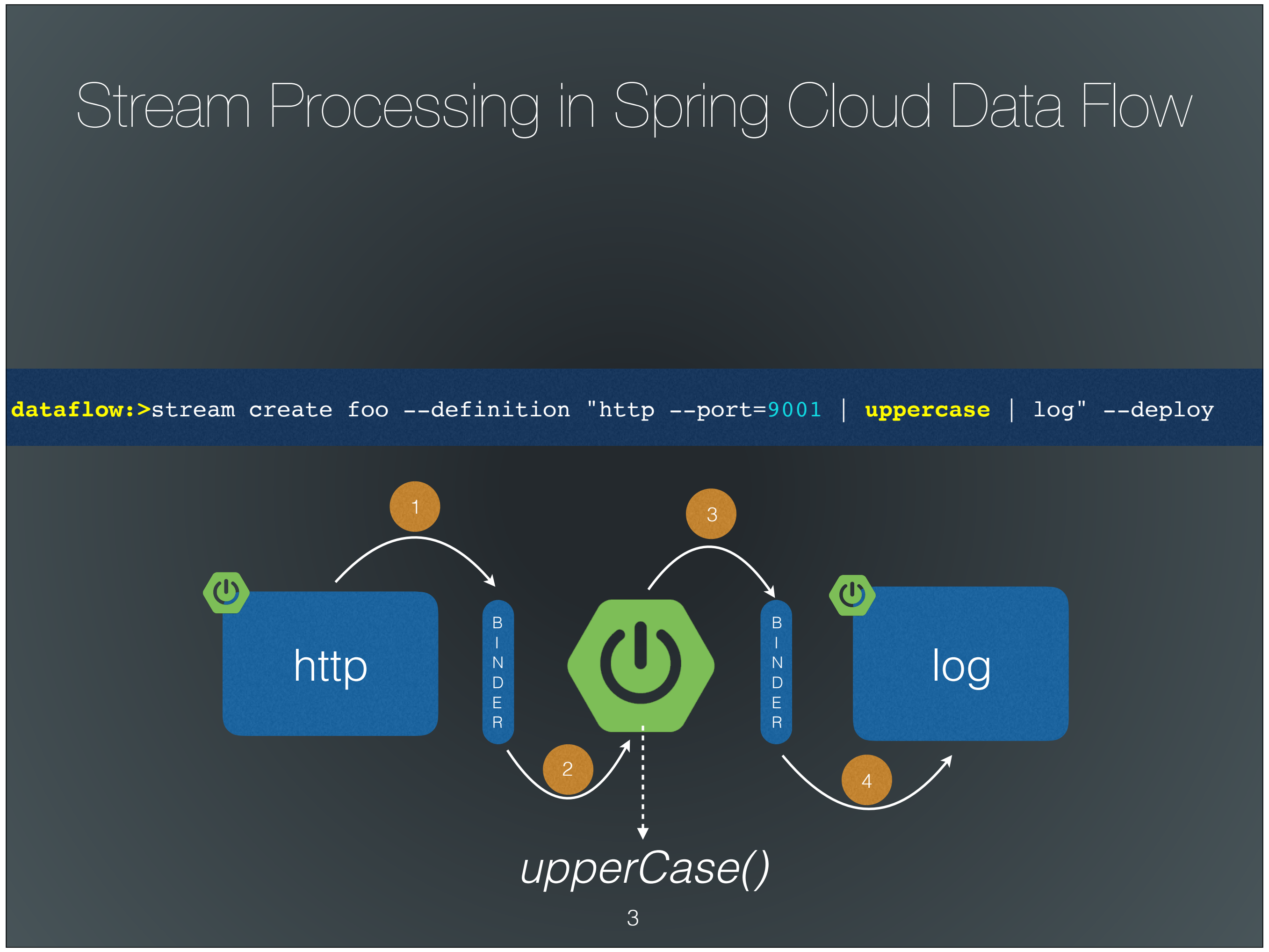
Stream Processing

+

Spring Cloud Data Flow



We are going to revisit stream processing in the context of SCDF. The benefits of orchestration aspects of SCDF is what we are going to cover.



Stream Processing in Spring Cloud Data Flow

```
dataflow:>stream create foo --definition "http --port=9001 | uppercase | log" --deploy
```

This is the same example that we saw in Streams labs. Let’s revisit it in the context of SCDF.

First and foremost, the DSL. This is where the power is. Inspired by UNIX’s pipes and filter syntaxes, the SCDF DSL includes a powerful representation of both linear and complex streaming pipeline topologies.

The same uppercase() can be defined multiple ways. We are all about options.

1) you can register the same application that you had developed in the lab and reuse it here
2) you can use SpEL expressions to compute simple to a complex transformations

“Easy to build or customize; fast you can build topologies; quick you can promote to production”

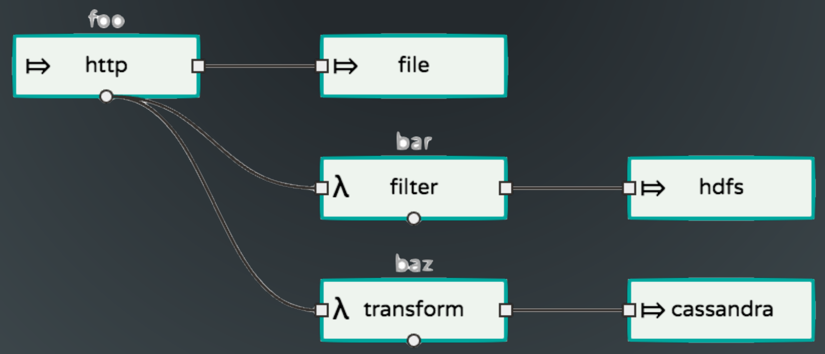
Same as:

```
dataflow:> stream create --definition "http | transform --expression=payload.toUpperCase() | log" --name mystream --deploy
```

Demo: upperCase()

Wiretaps in Streams

```
dataflow:>stream create foo --definition "http --port=9001 | file”

dataflow:>stream create bar --definition “:foo.http > filter | hdfs”

dataflow:>stream create baz --definition “:foo.http > transform | cassandra”
```



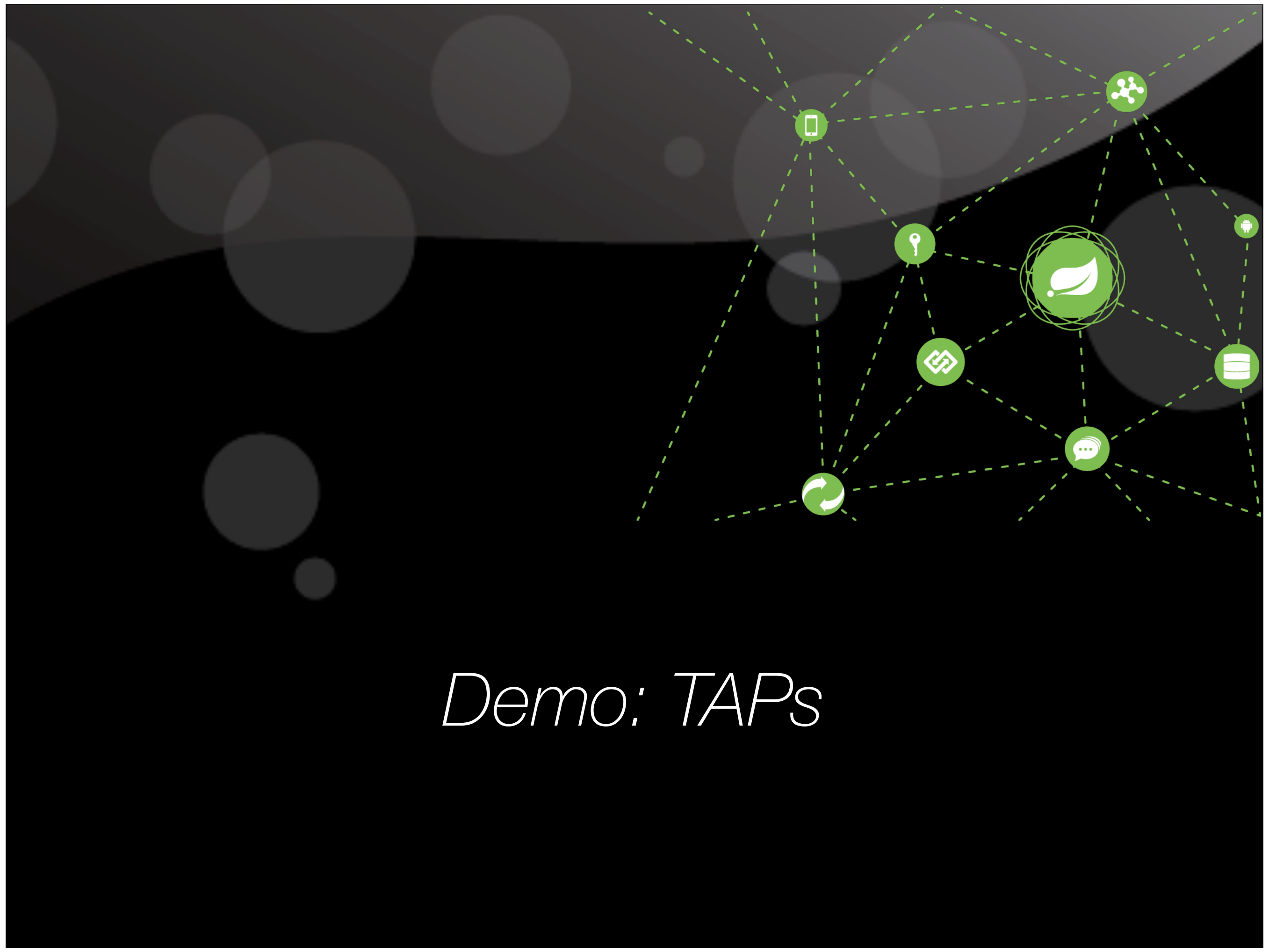
There are some responsibilities for primary data pipeline. The pipeline that writes RAW data to a master dataset; something like hadoop or s3 for example.

While that’s happening, if you’d want to fork and get a copy of same data and do some ad-hoc analytics or some other downstream processing, yes, you can in SCDF.

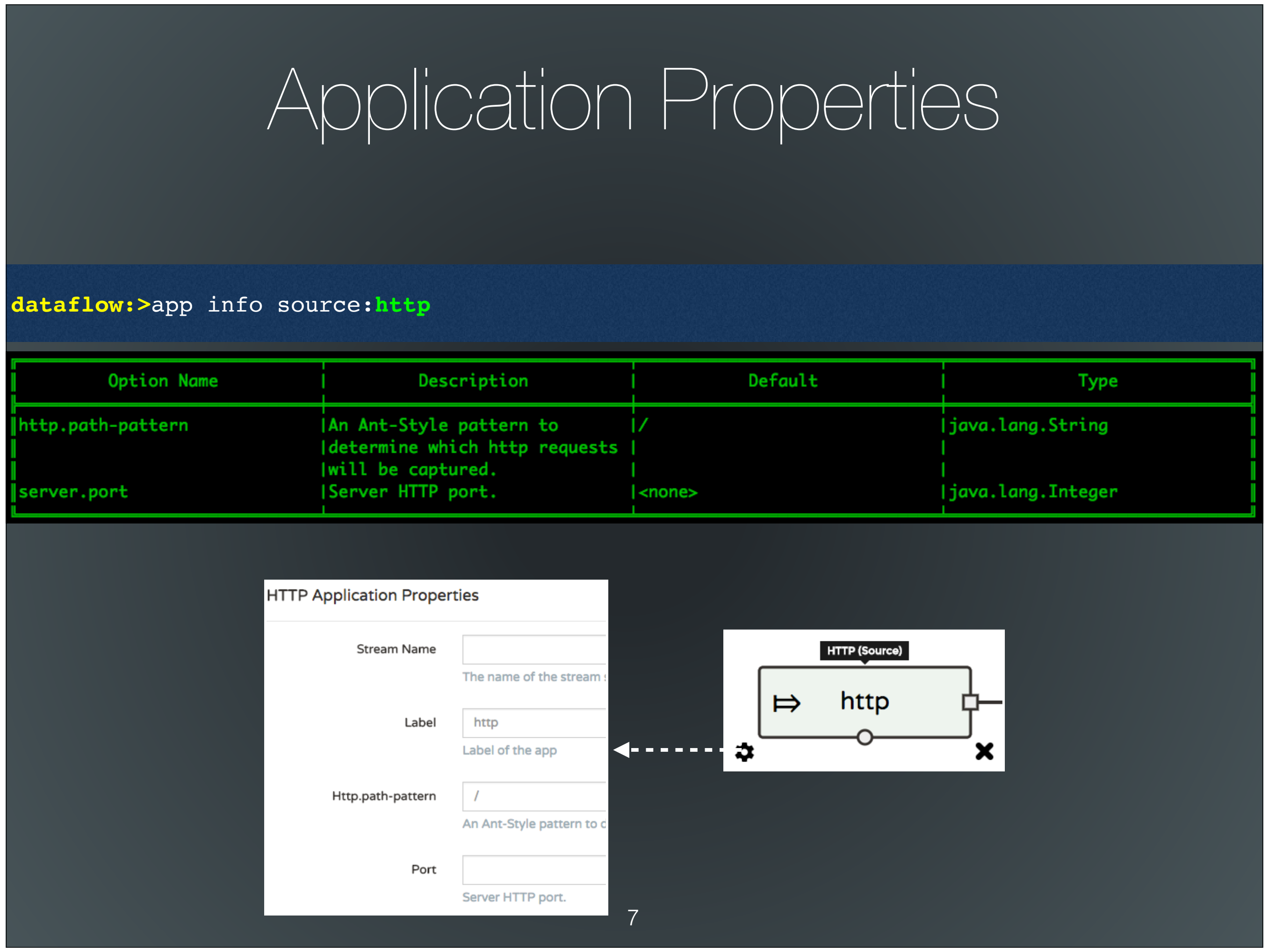
We call it TAPing the stream. Again, the DSL provides easy to use syntaxes to TAP a copy of data.

1) stream ***foo*** - Data from http port=9001 written to S3.
2) stream ***bar*** - TAPs data from the producer on ***foo*** stream and the copy of data is then sent to “hdfs” after some cleansing through “filter” processor.
3) stream ***baz*** - TAPs data from the producer on ***foo*** stream and the copy of data is now sent to “cassandra" after some transformation through “transform” processor.

Below you’d see the flo representation of this complex topology. It is easy to read and understand.



Demo: TAPs



Application Properties

```
dataflow:>app info source:http
╔══════════════════════════════╤══════════════════════════════╤══════════════════════════════╤══════════════════════════════╗
║         Option Name          │         Description          │           Default            │             Type             ║
╠══════════════════════════════╪══════════════════════════════╪══════════════════════════════╪══════════════════════════════╣
║http.path-pattern             │An Ant-Style pattern to       │/                             │java.lang.String              ║
║                              │determine which http requests │                              │                              ║
║                              │will be captured.             │                              │                              ║
║server.port                   │Server HTTP port.             │<none>                        │java.lang.Integer             ║
╚══════════════════════════════╧══════════════════════════════╧══════════════════════════════╧══════════════════════════════╝
```

HTTP Application Properties

Stream Name	The name of the stream
Label	http — Label of the app
Http.path-pattern	/ — An Ant-Style pattern to
Port	Server HTTP port.

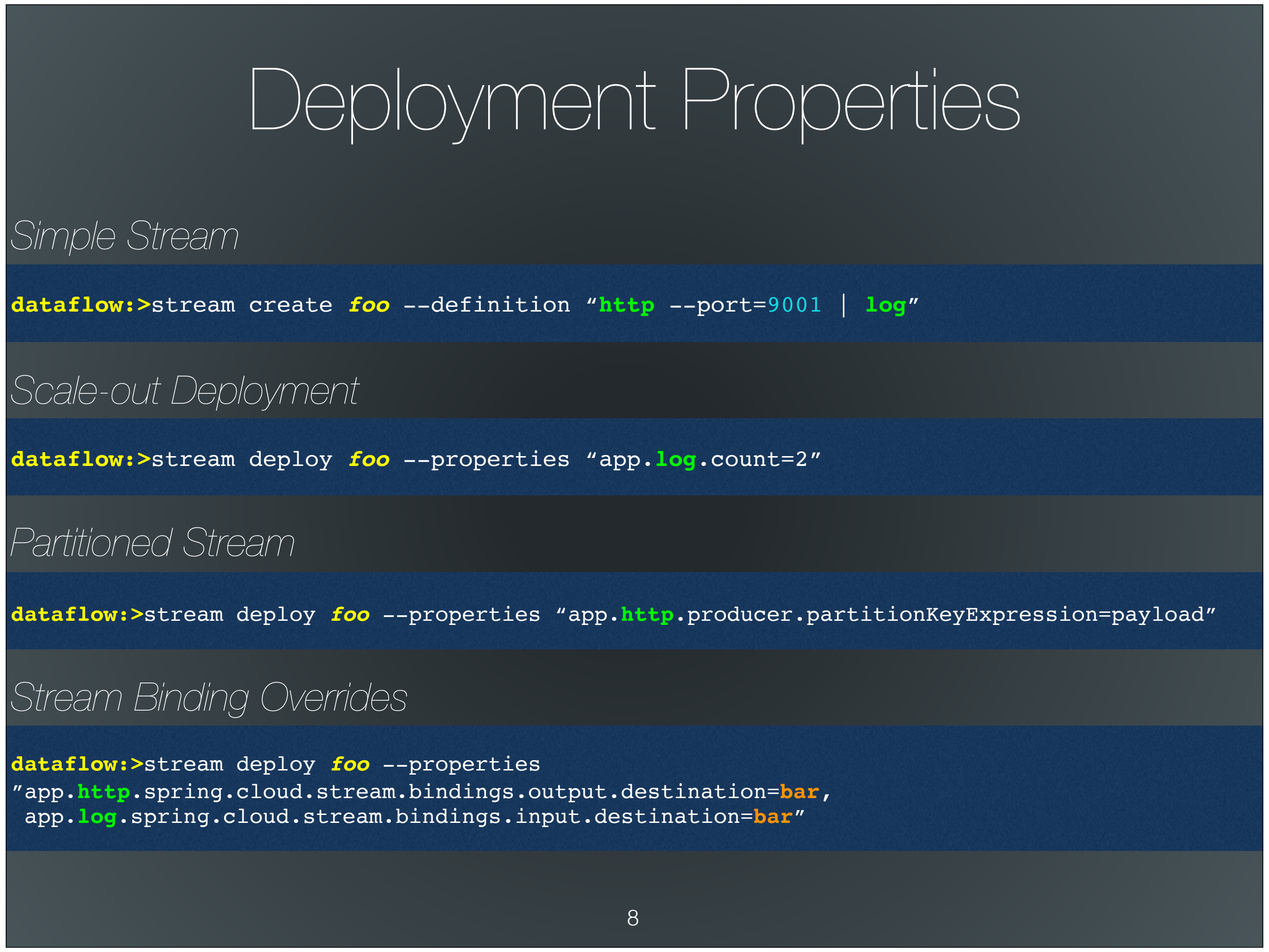
Fundamentally, everything in the pipeline is a Spring Boot application. All Boot goodies apply to these applications, too.

Specifically, the properties are a powerful feature in Boot; with that, we could override the application behavior at runtime.

There are multiple ways one can enable such properties and likewise, there are plenty of ways to override them in Boot. Refer to Boot’s ref. guide for more details: https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

Building upon this, SCDF and the ecosystem of projects provide similar support, too. From SCDF perspective, the DSL lends easy to use method of overriding application properties.

In this example, you are seeing all the overridable properties for “http-source” application. The same can be overridden when you create pipelines via Flo.



Deployment Properties

Simple Stream

```
dataflow:>stream create foo --definition “http --port=9001 | log”
```

Scale-out Deployment

```
dataflow:>stream deploy foo --properties “app.log.count=2”
```

Partitioned Stream

```
dataflow:>stream deploy foo --properties “app.http.producer.partitionKeyExpression=payload”
```

Stream Binding Overrides

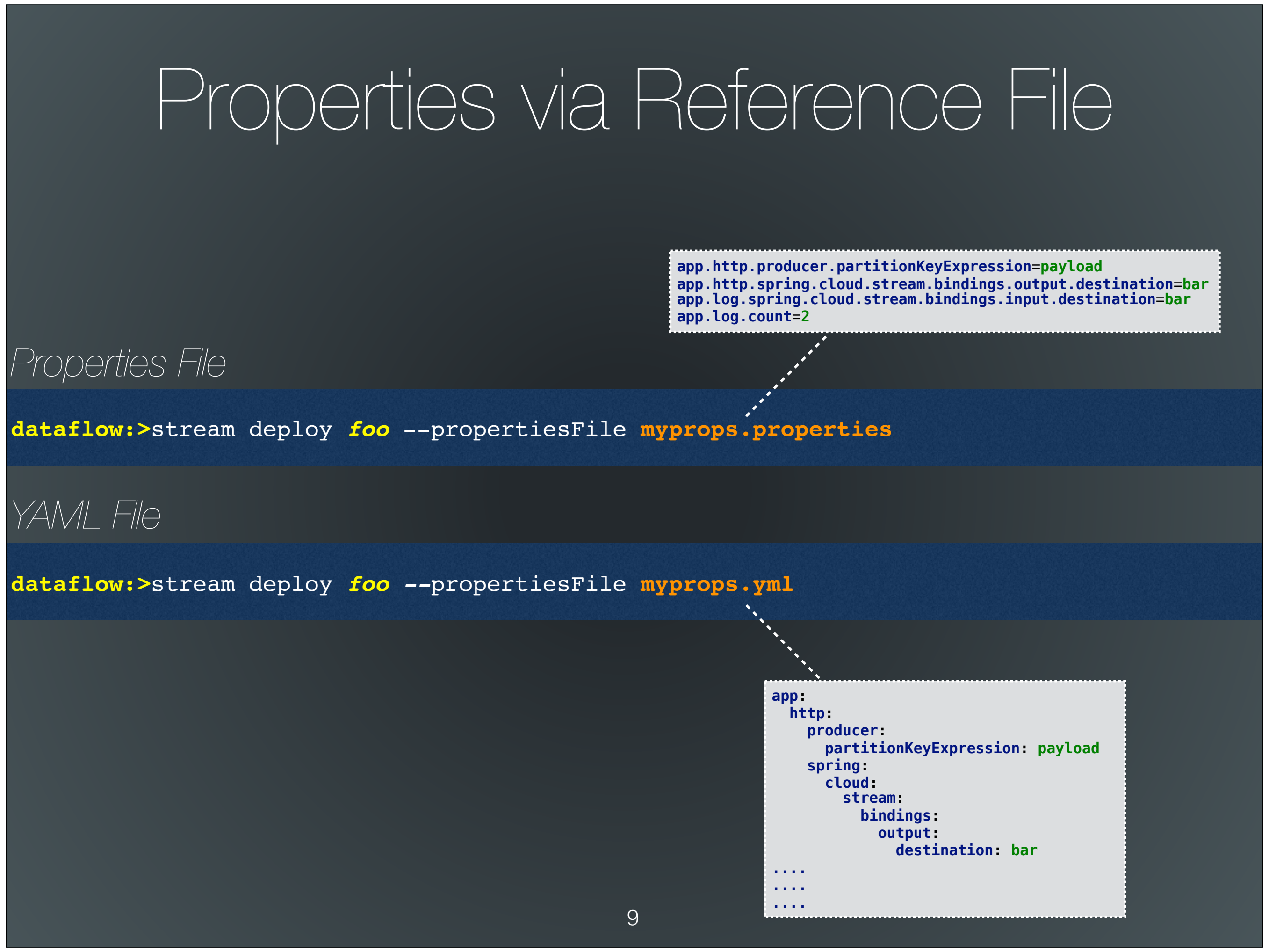
```
dataflow:>stream deploy foo --properties
”app.http.spring.cloud.stream.bindings.output.destination=bar,
 app.log.spring.cloud.stream.bindings.input.destination=bar”
```

We reviewed application properties.

When deploying the stream, properties that control the deployment of the apps into the target platform are known as deployment properties. SCDF provides easy method to override them as well.

1) Imagine a simple stream “http | log”
2) Now, if you’d want to deploy 2-instances of log-sink, you’d do that via “app.log.count=2”
3) If you’d want to supply a partitioning method for the producer (http-source), you’d do that via: “app.http.producer.partitionKeyExpression=payload”
4) If you’d want to override the default destinations (generated by SCDF = ***<server-name>.<stream-name>.<app-name>***) for your producer and consumer applications to connect, you’d do that via ”app.http.spring.cloud.stream.bindings.output.destination=bar,app.log.spring.cloud.stream.bindings.input.destination=bar”

http://docs.spring.io/spring-cloud-stream/docs/Chelsea.M1/reference/htmlsingle/#_configuration_options



Properties via Reference File

```
app.http.producer.partitionKeyExpression=payload
app.http.spring.cloud.stream.bindings.output.destination=bar
app.log.spring.cloud.stream.bindings.input.destination=bar
app.log.count=2
```

Properties File

```
dataflow:>stream deploy foo --propertiesFile myprops.properties
```

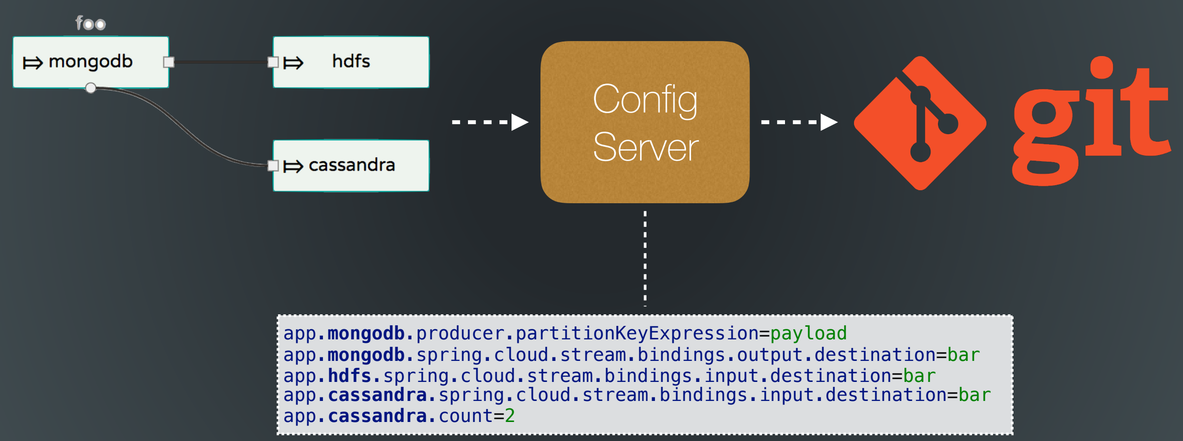
YAML File

```
dataflow:>stream deploy foo --propertiesFile myprops.yml
```

```
app:
  http:
    producer:
      partitionKeyExpression: payload
    spring:
      cloud:
        stream:
          bindings:
            output:
              destination: bar
....
....
....
```

Properties can be provided as plain “.properties” or as “.yml” artifacts.

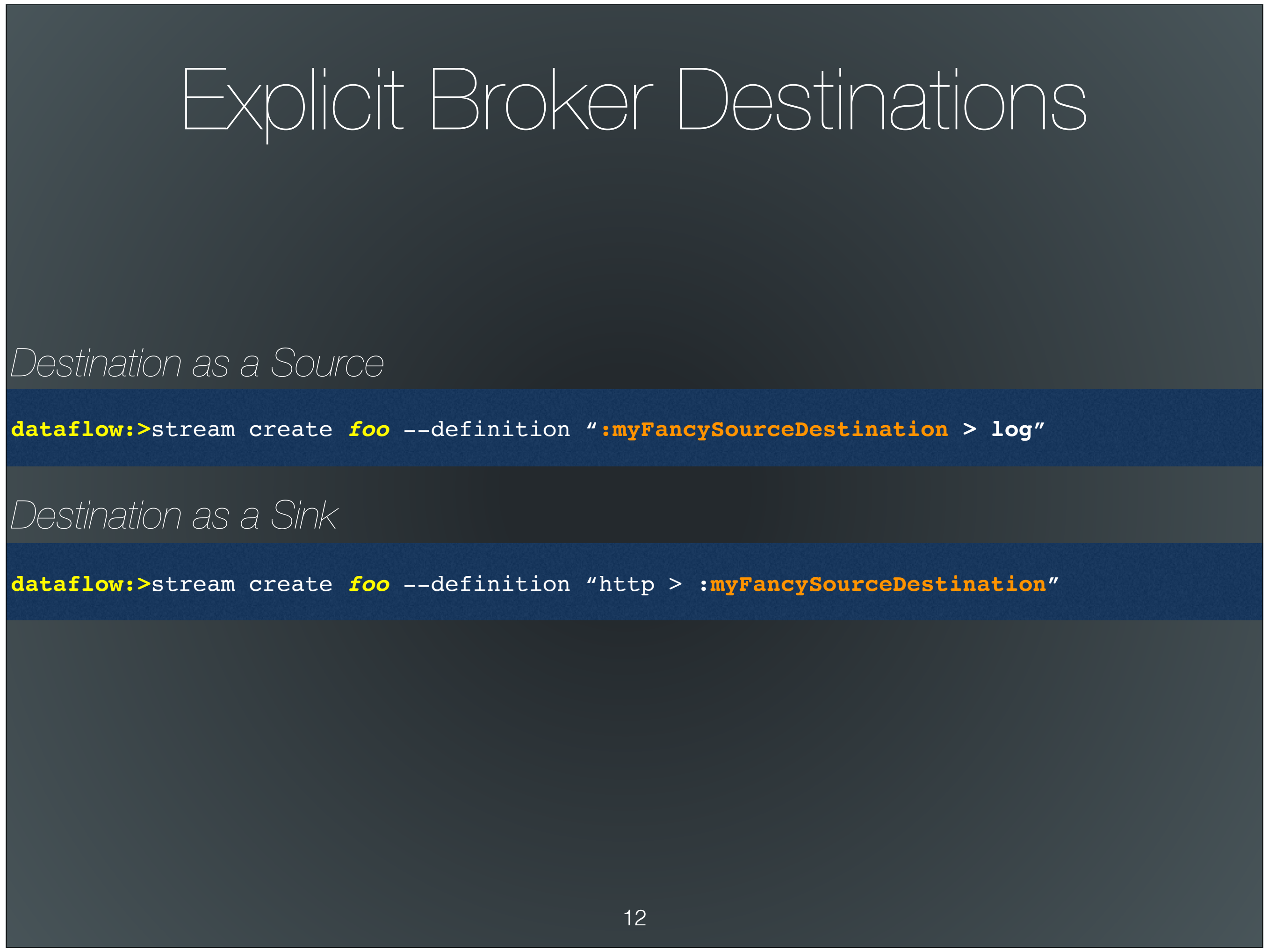
Properties via Reference File



```
app.mongodb.producer.partitionKeyExpression=payload
app.mongodb.spring.cloud.stream.bindings.output.destination=bar
app.hdfs.spring.cloud.stream.bindings.input.destination=bar
app.cassandra.spring.cloud.stream.bindings.input.destination=bar
app.cassandra.count=2
```

Alternatively, you can also use config-server backed by a centralized GIT to resolve properties more dynamically via “profiles” at the runtime.

Demo: All Things Properties



Explicit Broker Destinations

Destination as a Source

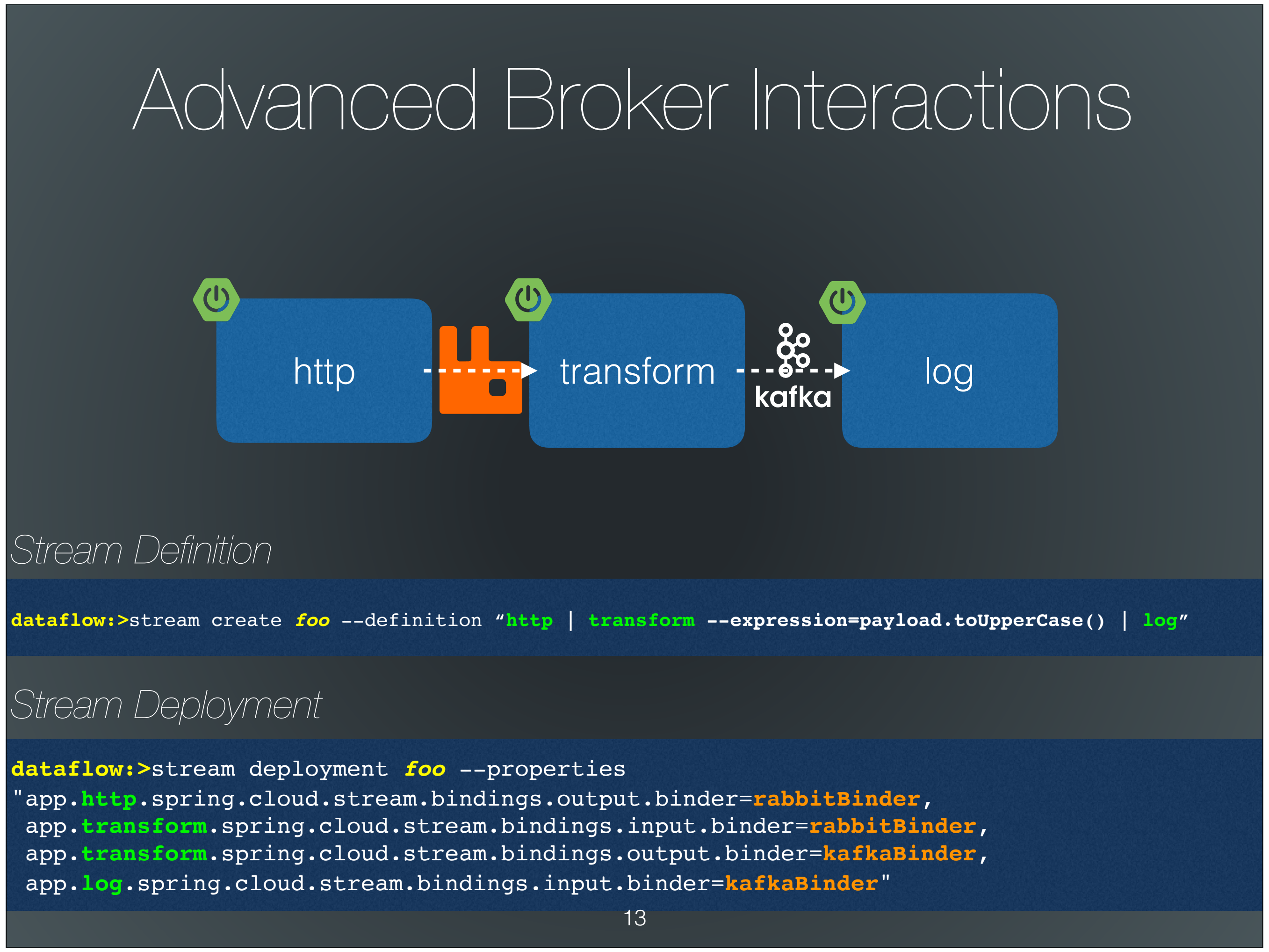
```
dataflow:>stream create foo --definition “:myFancySourceDestination > log”
```

Destination as a Sink

```
dataflow:>stream create foo --definition “http > :myFancySourceDestination”
```

You can connect directly to a queue/topic and the DSL for doing that is what you’re seeing in this slide.

You don’t need an explicit application as source/sink is the point. This method is more flexible and it eliminates the requirement to code, test and build producers from scratch.



Advanced Broker Interactions

Stream Definition

```
dataflow:>stream create foo --definition “http | transform --expression=payload.toUpperCase() | log”
```

Stream Deployment

```
dataflow:>stream deployment foo --properties
"app.http.spring.cloud.stream.bindings.output.binder=rabbitBinder,
 app.transform.spring.cloud.stream.bindings.input.binder=rabbitBinder,
 app.transform.spring.cloud.stream.bindings.output.binder=kafkaBinder,
 app.log.spring.cloud.stream.bindings.input.binder=kafkaBinder"
```

GLOBAL configurations provided at the SERVER level:

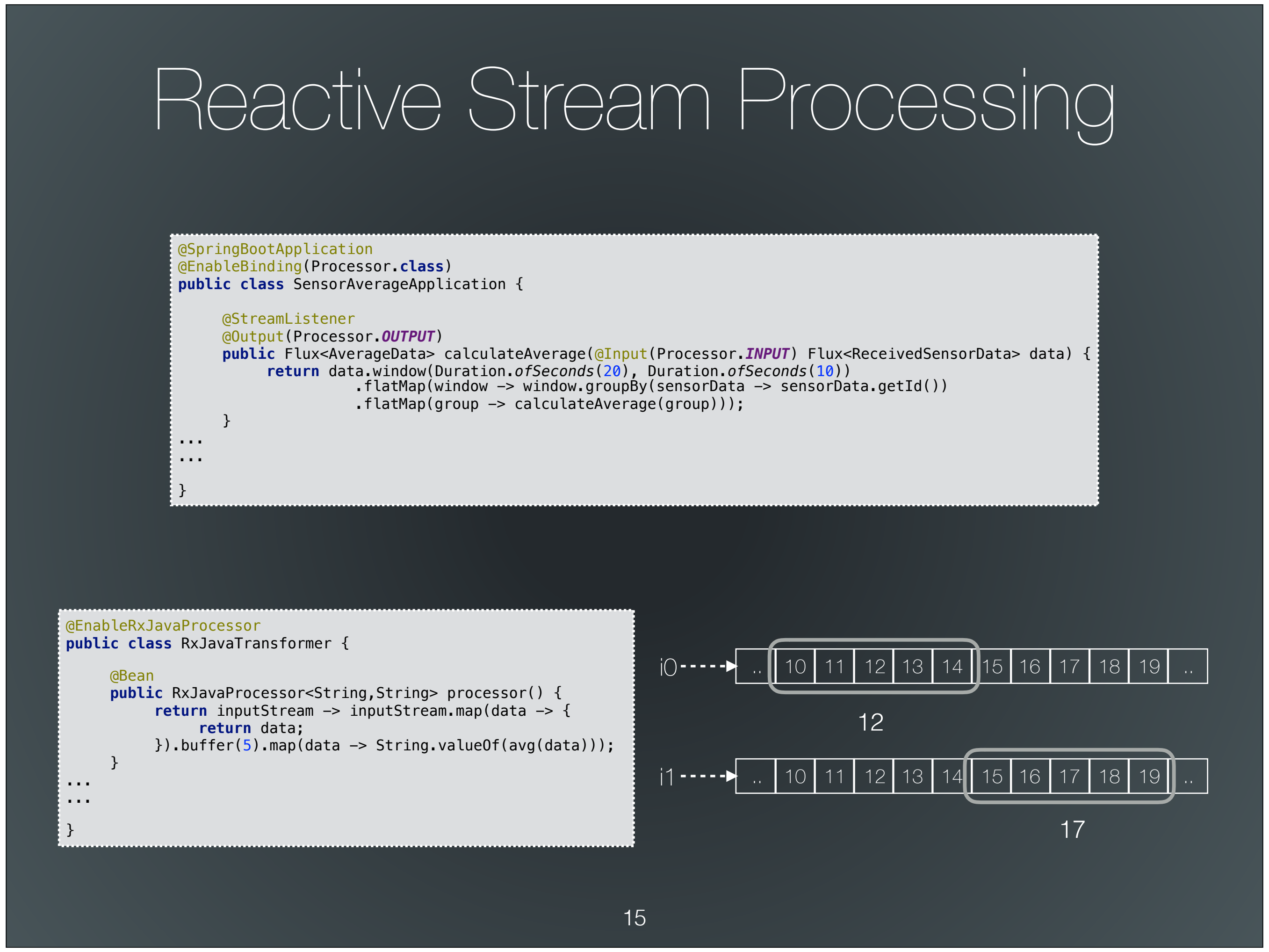
```
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.rabbitBinder.type=rabbit
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.rabbitBinder.environment.spring.rabbitmq.host=awesome1.dev
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.rabbitBinder.environment.spring.rabbitmq.port=90000
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.kafkaBinder.type=kafka
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.kafkaBinder.environment.spring.cloud.stream.kafka.binder.brokers=awesome2.dev:9092
spring.cloud.dataflow.applicationProperties.stream.spring.cloud.stream.binders.kafkaBinder.environment.spring.cloud.stream.kafka.binder.zkNodes=awesome2.dev:2181
```

It is easy to build complex pipelines that interact with multiple binder implementations.

1) Imagine a requirement that is strictly only *requires* RabbitMQ for some processing - perhaps something to do with Rabbit’s powerful security features.
2) At the same time, a team would like to use Kafka for some high throughput / low-latency downstream processing.

Both of these requirements are legit and in SCDF, we provide easy options to plug different binding implementation between producers and consumers. This visual represents that.

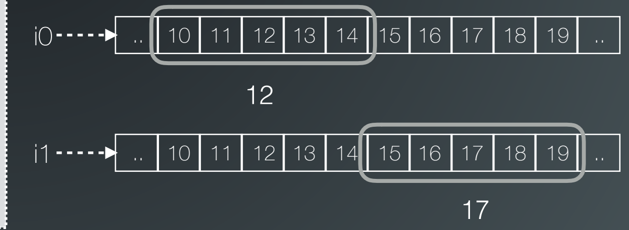
Demo: All Things Destination



Reactive Stream Processing

```java
@SpringBootApplication
@EnableBinding(Processor.class)
public class SensorAverageApplication {

     @StreamListener
     @Output(Processor.OUTPUT)
     public Flux<AverageData> calculateAverage(@Input(Processor.INPUT) Flux<ReceivedSensorData> data) {
          return data.window(Duration.ofSeconds(20), Duration.ofSeconds(10))
                     .flatMap(window -> window.groupBy(sensorData -> sensorData.getId())
                     .flatMap(group -> calculateAverage(group)));
     }
...
...

}
```

```java
@EnableRxJavaProcessor
public class RxJavaTransformer {

     @Bean
     public RxJavaProcessor<String,String> processor() {
          return inputStream -> inputStream.map(data -> {
               return data;
          }).buffer(5).map(data -> String.valueOf(avg(data)));
     }
...
...

}
```



Spring Cloud Stream also supports the use of reactive APIs where incoming and outgoing data is handled as continuous data flows.

Instead of specifying how each individual message should be handled, you can use operators that describe functional transformations from inbound to outbound data flows.

Project Reactor and RxJava APIs are more natively supported at the programming model.

With this, you can build complex computation business logic in your applications. Examples include “moving-average”, “time-windowing”, “aggregations” — all this can be easily done in the application itself. They can be registered in SCDF to build complex streaming topologies using the DSL/Flo.

A moving-average example is shown in this slide. For every 5-entries coming from the upstream application, the average is calculated and the outcome is sent to downstream app for processing/persistence.

http://docs.spring.io/spring-cloud-stream/docs/Chelsea.M1/reference/htmlsingle/#_reactive_programming_support

Demo: Moving Average