Adam Bellemare - Building Event-Driven Microservices

15. Testing Event-Driven Microservices

General Testing Principles
- 👍 great things about testing event-driven microservices is that they're very modula
- Functional testing
- Nonfunctional testing

Unit-Testing Topology Functions

Stateless Functions

```
myInputStream
    .filter(myFilterFunction)
    .map(myMapFunction)
    .to(outputStream)
```

myMapFunction and myFilterFunction are independent functions, neither of which keeps state

Stateful Functions

```
public Long addValueToAggregation(String key, Long eventValue) {
    //The data store needs to be made available to the unit-test environment
    Long storedValue = datastore.getOrElse(key, 0L);
    //Sum the values and load them back into the state store
    Long sum = storedValue + eventValue;
    datastore.upsert(key, sum);
    return sum;
}
```

Two ways of implementing data store
- 👍 mocking the endpoint — good for high-performance unit testing
- locally available version of the data store

Testing the Topology
- Topology is a single, large, complex function with many moving parts
  - Which event is produced
  - When produced
- Apache Spark
  - StreamingSuiteBase
  - spark-fast-tests
  - built-in MemoryStream class
- Apache Flink and Apache Beam — provide their own topology testing
- Kafka Streams — TopologyTestDriver
- 👍 testing frameworks do not require to create an event broker to hold input events

Testing Schema Evolution and Compatibility — Pull in the schemas from the schema registry and perform evolutionary rule checking

Integration Testing of Event-Driven Microservices

Local

Requires
- event broker
- schema registry
- any microservice-specific data stores
- the microservice itself
- any required processing framework (e.g. heavyweight or SaaS)
- also can use
  - containerization
  - logging
  - container management system

Can test
- intermittent failures
- out-of-order events
- loss of network access
- functionality of horizontal scaling (particularly where copartitioning and state are concerned)
- event-driven and request-response logic at the same time, in the same workflows

2 main ways of creating test env
- Within the Runtime of Your Test Code
- External to Your Test Code

Integrate Hosted Services Using Mocking and Simulator Options
- Event brokers
  - Google's PubSub, Kinesis — 👍 have emulators
  - Microsoft Azure's Event Hubs — 👎 does not have emulator — however, it allows to use Kafka but not for all features
- FaaS
  - Google Cloud functions, Amazon's lambda, MS Azure functions — 👍 can be tested locally
  - Whisk, OpenFaaS, and Kubeless — also provide mechanisms for local testing

Integrate Remote Services That Have No Local Options — ⚠️ The remote environments must be provisioned for each developer

Remote

Temporary Integration env
- Populating with events from production — ⚠️ But must account for any security!
- Populating with events from a curated testing source
- Creating mock events using schemas

Shared env
- With the testing data that represents a subset of production data, or carefully crafted testing data
- 🔨 a common strategy to employ when investment in tooling is low

⚠️ Production env
- 👍 use its own designated output event streams and state stores — such that it doesn't affect the existing production systems

Hybrid

Choosing Your Full-Remote Integration Testing Strategy — ⚠️ Invest into supportive tooling!