Reactive Platform from scratch, based on Scala.

IoT example based on Lagom Microservices.

accenture digital

David Urdiales Nieto & Rafael Hidalgo

Big Data Team Lead

Cádiz, 26th October 2017

Agenda

1.- Lagom Overview

2.- Lagom Development Environment

3.- Service API

4.- Persistence API

5.- Message Broker API

6.- Workshop Demo

1.- Lagom Overview

- Lagom Framework
- Language
- Components
- Features

Lagom Framework

- Microservices framework for a system of microservices
- Service location, communication protocols, and other issues are handled by Lagom transparently.
- Event sourcing and CQRS (Command Query Responsibility Segregation) for persistence.
- Fully integrated development environment

Language

- Core of framework written in Scala
- Developments are in Java, Scala(since v1.3.1)

Components

- Java8(& Scala)
- SBT
- Jackson
- Cassandra
- Play framework
- Akka
- SLF4J & Logback

Features

- Service API
- Persistence API
- Message Broker API
- Development Environment
- Production Environment

2.- Lagom Development Environment

- Service Gateway
- Service Locator
- Embedded Cassandra Server
- Embedded Kafka Server

Service Gateway

- Service Gateway is embedded in Lagom's development environment,
- By default, the service gateway runs on port: 9000
- http://localhost:9000

https://www.lagomframework.com/documentation/1.3.x/scala/ServiceLocator.html

Service Locator

- A **Service Locator** is embedded in Lagom's development environment, allowing services to discover and communicate with each others.
- By **default**, the service locator runs on **port: 8000**
- **demo --> port: 10000**
- http://localhost:10000

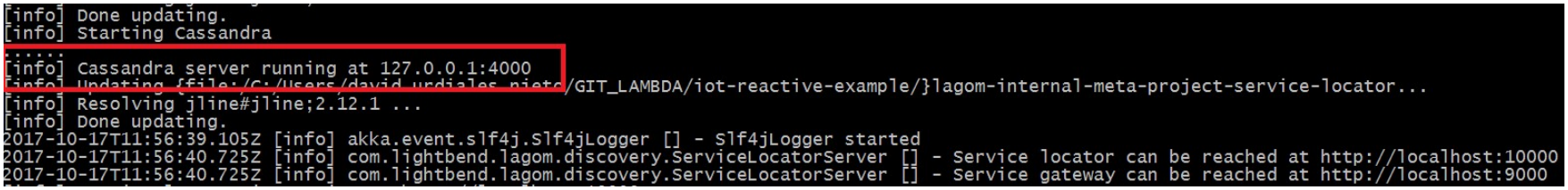
build.sbt

```
1   // Project settings
2   organization in ThisBuild := "com.example"
3   version in ThisBuild := "0.1.0-SNAPSHOT"
4
5   // Build Settings
6   scalaVersion in ThisBuild := "2.11.8"
7
8   // Runtime properties
9   lagomServiceLocatorPort in ThisBuild := 10000
10
```

https://www.lagomframework.com/documentation/1.3.x/scala/ServiceLocator.html

Cassandra Server

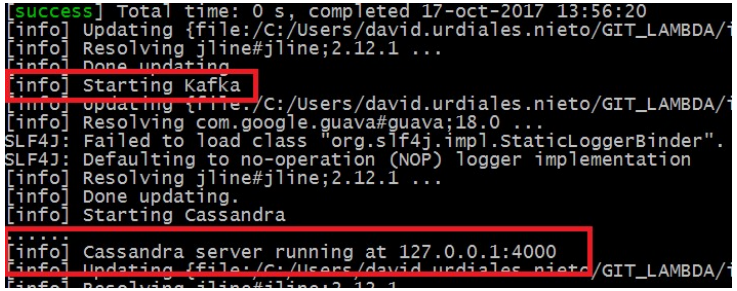
- By default, Lagom services needing to persist data use Cassandra as database.
- An Embedded Cassandra server is used in the development environment, so that you don't have to install it.
- By default, the Cassandra server is started on port: 4000



https://www.lagomframework.com/documentation/1.3.x/scala/CassandraServer.html

Kafka Server

- By default, **Lagom services** that **need to share information between each others** use **Kafka as a message broker**.
- **In a microservice architecture**, usage of a message broker ensures that the services are not strongly coupled with each other. **Embedded a Kafka server** in the dev. environment
- By **default**, the **Kafka server** is started on **port: 9092**
- Zookeeper port: 2181



https://www.lagomframework.com/documentation/1.3.x/scala/KafkaServer.html

3.- Service API

- Interface as descriptor. Service Descriptor
- Implementing Service
- Dependency Injection. Macwire
- Testing Services

Service Descriptor

- Interface as descriptor. Service Descriptor
- Defined API of exposed Service.

```
override def descriptor: Descriptor = {
  import Service._
  named("logger").withCalls(
    restCall(Method.POST, "/measure", addMeasure _),
    pathCall("/measure/:id", getMeasures _),
    pathCall("/measure/latest/:id", getLatestMeasure _)
  ).withTopics(
    topic[AddMeasure](DataLoggerService.AddMeasureTopic, publishMeasure)
  ).withAutoAcl(true)
}
```

https://www.lagomframework.com/documentation/1.3.x/scala/ServiceDescriptors.html

Implementing Service

- Services are implemented by providing an implementation of the service descriptor trait, implementing each call specified by that descriptor.
- Implementation of the DataLoggerService descriptor

```
class DataLoggerServiceImpl(persistentEntityRegistry: PersistentEntityRegistry,
                            cassandraReadSide: CassandraReadSide,
                            readSide: ReadSide,
                            session: CassandraSession)
                           (implicit ctx: ExecutionContext) extends DataLoggerService {
  /** Just a simple logger. **/
  private final val logger: Logger = LoggerFactory.getLogger(classOf[DataLoggerServiceImpl])
```

https://www.lagomframework.com/documentation/1.3.x/scala/ServiceImplementation.html

Dependency Injection -1

- The simplest way to build the **Application cake** and then **wire** your code inside it is by creating an abstract class that extends **LagomApplication**
- **Macwire** to wire the dependencies

```
abstract class DataLoggerApplication(context: LagomApplicationContext)
  extends LagomApplication(context)
    with AhcWSComponents
    with CassandraPersistenceComponents {

  override lazy val lagomServer = serverFor[DataLoggerService](wire[DataLoggerServiceImpl])
  override lazy val jsonSerializerRegistry = MeasureSerializerRegistry
```

build.sbt

```
10
11  // Dependencies
12  val macwire = "com.softwaremill.macwire" %% "macros" % "2.2.5" % "provided"
```

https://www.lagomframework.com/documentation/1.3.x/scala/DependencyInjection.html

Dependency Injection -2

- Created the application cake, now write an application loader. LagomApplicationLoader
- Methods that must be implemented: load and loadDevmode

```
class DataLoggerLoader extends LagomApplicationLoader {

  override def load(context: LagomApplicationContext) =
    new DataLoggerApplication(context) with LagomKafkaComponents {
      override def serviceLocator = ServiceLocator.NoServiceLocator
    }

  override def loadDevMode(context: LagomApplicationContext): LagomApplication =
    new DataLoggerApplication(context) with LagomDevModeComponents with LagomKafkaComponents

  override def describeService = Some(readDescriptor[DataLoggerService])
}
```

Testing Services

- Lagom testkit.

```
val scalaTest = "org.scalatest" %% "scalatest" % "3.0.1" % Test
```

```
// Define datalogger-impl project build
lazy val `datalogger-impl` = (project in file("datalogger-impl"))
  .enablePlugins(LagomScala)
  .settings(
    libraryDependencies ++= Seq(
      lagomScaladslPersistenceCassandra,
      lagomScaladslKafkaBroker,
      lagomScaladslPubSub,
      lagomScaladslTestKit,
      macwire,
      scalaTest
    )
```

- Support for functional tests.
- ServiceTest.

  The service is running in a server for test.

```
class DataLoggerServiceSpec extends AsyncWordSpec with Matchers with BeforeAndAfterAll {

  lazy val measurement: AddMeasure = AddMeasure("1234-fgh", DateTime.now, List(Measure("foo", 0.0123)))

  lazy val server = ServiceTest.startServer(ServiceTest.defaultSetup.withCassandra(true)) { ctx =>
    new DataLoggerApplication(ctx) with LocalServiceLocator with TestTopicComponents {

    }
  }
  lazy val client = server.serviceClient.implement[DataLoggerService]
  implicit lazy val system = server.actorSystem
  implicit lazy val mat = server.materializer
```

https://www.lagomframework.com/documentation/1.3.x/scala/TestingServices.html

4.- Persistence API

- Persistent Entity. Event Sourcing
- Persistent ReadSide. CQRS
- Cassandra Read-Side Support

Persistent Entity

- The three abstract type members that the concrete PersistentEntity subclass must define: Command, Event, State
- PersistentEntity Interaction by sending command messages.
- The state of an entity is persistent using Event Sourcing.

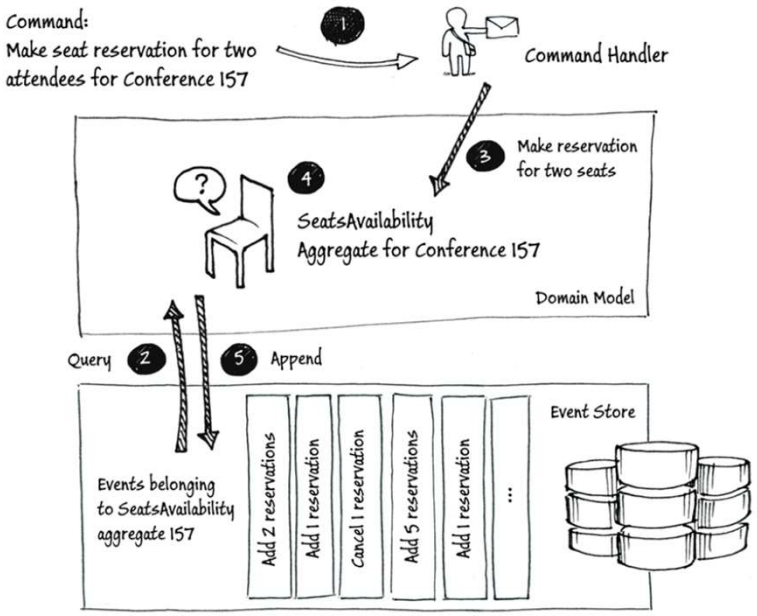
```
final class MeasureEntity extends PersistentEntity {
  override type Command = MeasureCommand
  override type Event = MeasureEvent
  override type State = MeasureState

  override def initialState = MeasureState.empty

  override def behavior: Behavior = Actions()
    .onCommand[AddMeasure, Done] {
    case (addMeasure, ctx, state) => {
      ctx.thenPersist(AddMeasureEvent(addMeasure.asInstanceOf[AddMeasure])) { _ =>
        ctx.reply(Done)
      }
    }
  }.onEvent {
    case (AddMeasureEvent(measure), state) => MeasureState(Some(measure), logged = true)
```

https://www.lagomframework.com/documentation/1.3.x/scala/PersistentEntity.html

Event Sourcing



https://msdn.microsoft.com/en-us/library/jj591559.aspx

Event sourcing model:

- Treats the **database** as an append-only log of serialized events.

1.- **Command** to reserve seats. Handled by command handler.

2.- Aggregate instance is populated by querying for all of the events that belong to **SeatsAvailability** aggregate 157.

3.- Command handler **invokes** the **business method**.

4.- The SeatsAvailability Aggregate performs its domain logic.

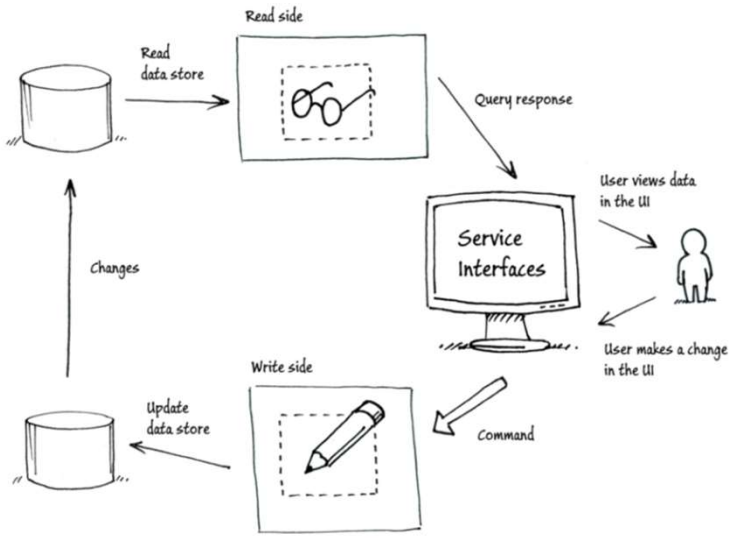
5.- **The system** appends the event that records making **two new reservations**.

Persistent Read-Side

- **Persistent Entities:** for holding the state of individual entities, but not for serving queries that span more than one entity.
- Create **another view** of the data that is tailored to the queries that the service provides.
- Separation of the **write-side** and the **read-side** of the persistent data: CQRS
- ReadSideProcessor: responsible for handling the events produced by the persistent entity, and tracking which events it has handled. This is done using **offsets**.

https://www.lagomframework.com/documentation/1.3.x/scala/ReadSide.html

CQRS (Command Query Responsability Segregation)



https://msdn.microsoft.com/en-us/library/jj591573.aspx

With the CQRS pattern:

- The write-side entities focus on the updating commands
- ...and optimize the read-side for different types of queries.
- Better scalability. Read-side can be scaled out to many nodes independently of the write-side.
- The object or objects of the read side contain only query methods.
- The objects on the write side contain only command methods.

Cassandra Read-Side Support

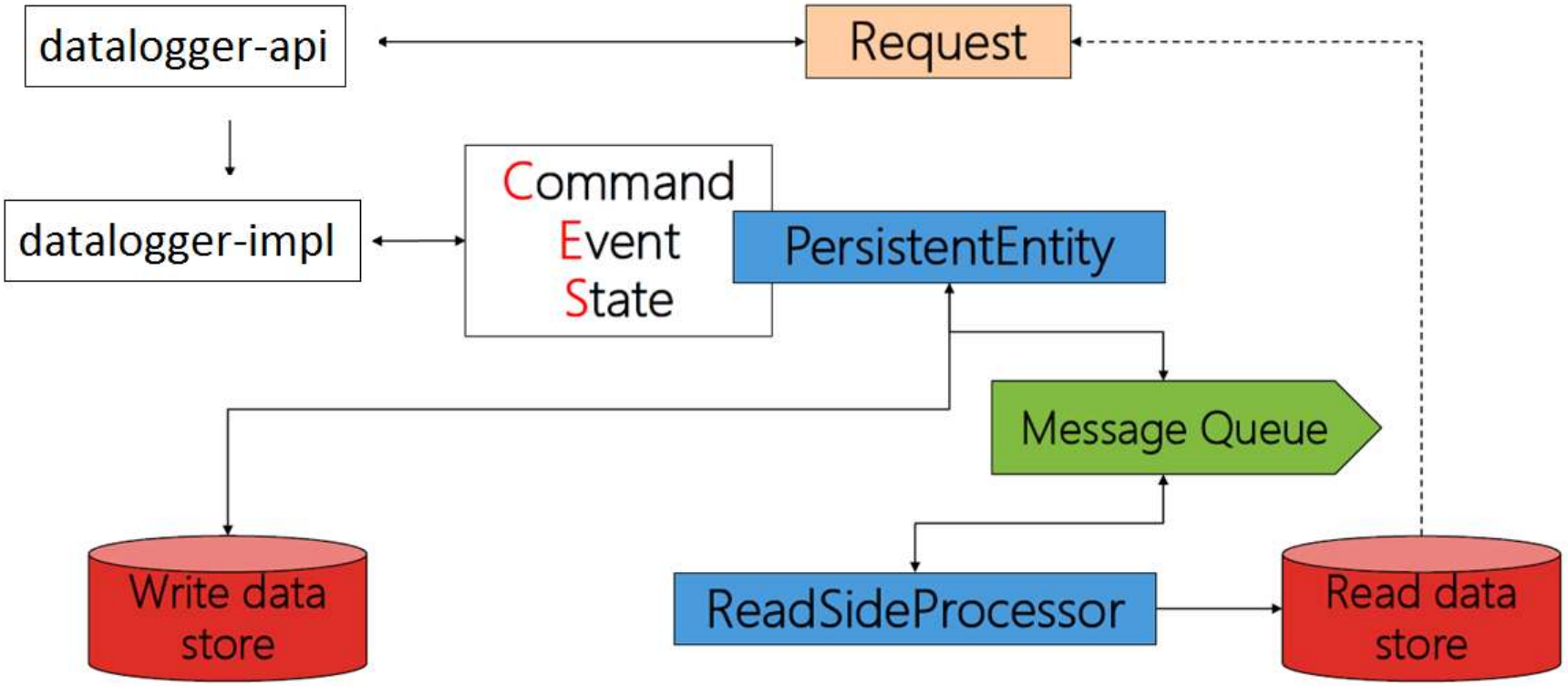
- CassandraSession & CassandraReadSide
- All methods in CassandraSession are non-blocking and they return a Future or a Source.
- Statements are expressed in Cassandra Query Language (CQL)

```
class MeasureEventProcessor(session: CassandraSession, readSide: CassandraReadSide)
                           (implicit ctx: ExecutionContext) extends ReadSideProcessor[MeasureEvent] {

  /** *
    * Statement used to select all measure from some metrics into a timerange
  def selectMeasuresFromTimeRangeStatement(id: String, name: String, init: String, end: String): String =
    s"""SELECT id , name, tstamp, value, last_updated
       |FROM $MeasureTable
       |WHERE id= '$id' AND name = '$name' AND tstamp >= '$init' AND tstamp <= '$end'
     """.stripMargin
```

https://www.lagomframework.com/documentation/1.3.x/scala/ReadSideCassandra.html

Service & Persistence Workflow



5.- Message Broker API

- Declaring a Topic
- Implementing a Topic. Pushing Data

Declaring a Topic

- The syntax for declaring a topic is similar to the one used already to define services' endpoints.
- The Descriptor.withTopics method accepts a sequence of topic calls, each topic call can be defined via the topic method on the Service object.

```
override def descriptor: Descriptor = {
  import Service._
  named("logger").withCalls(
    restCall(Method.POST, "/measure", addMeasure _),
    pathCall("/measure/:id", getMeasures _),
    pathCall("/measure/latest/:id", getLatestMeasure _)
  ).withTopics(
    topic[AddMeasure](DataLoggerService.AddMeasureTopic, publishMeasure)
  ).withAutoAcl(true)
}
```

https://www.lagomframework.com/documentation/1.3.x/scala/MessageBrokerApi.html

Implementing a Topic. Pushing Data

- Stream of events from persistent entities, need to be adapted to a stream of messages to be sent to the message broker.
- TopicProducer provides two methods for publishing a persistent entities event stream. singleStreamWithOffset (non sharded), taggedStreamWithOffset (sharded)
- PersistentEntityRegistry.eventStream method for obtaining a read-side stream.

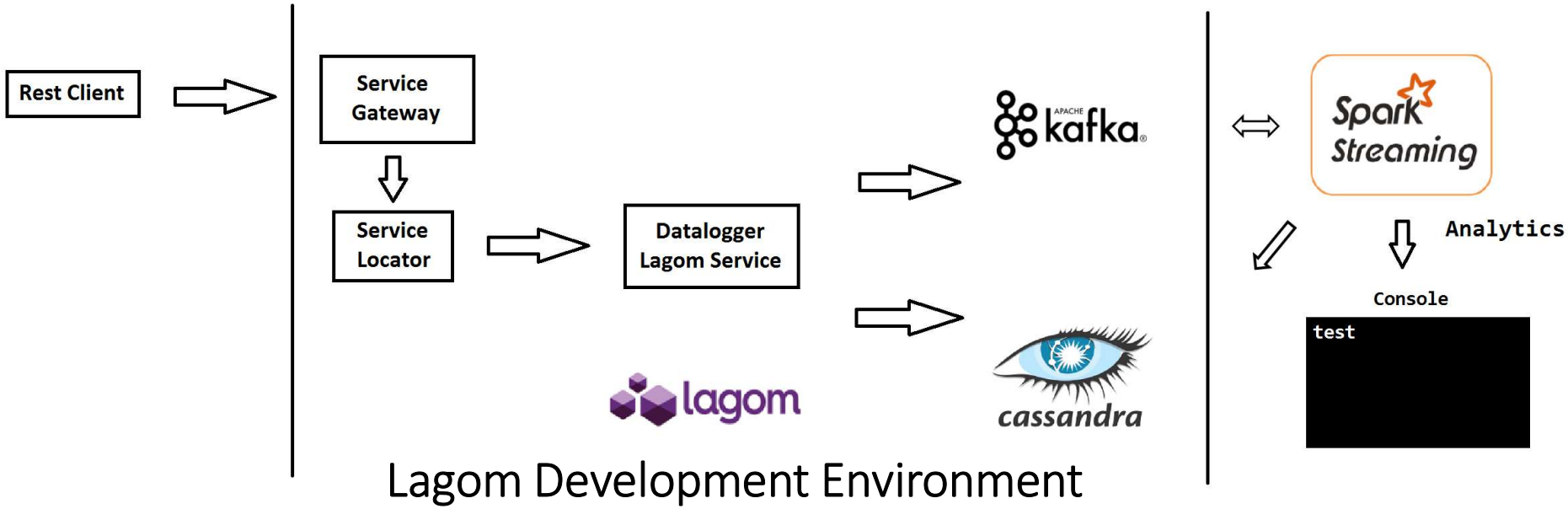
```
override def publishMeasure: Topic[AddMeasure] = TopicProducer
  .taggedStreamWithOffset(MeasureEvent.Tag.allTags.toList) { (tag, fromOffset) =>
    persistentEntityRegistry.eventStream(tag, fromOffset) map { event =>
      logger.info(s"Handling event on topic: $event")
      event.event match {
        case AddMeasureEvent(measure) => (measure, event.offset)
      }
    }
  }
```

https://www.lagomframework.com/documentation/1.3.x/scala/MessageBrokerApi.html

6.- Workshop Demo

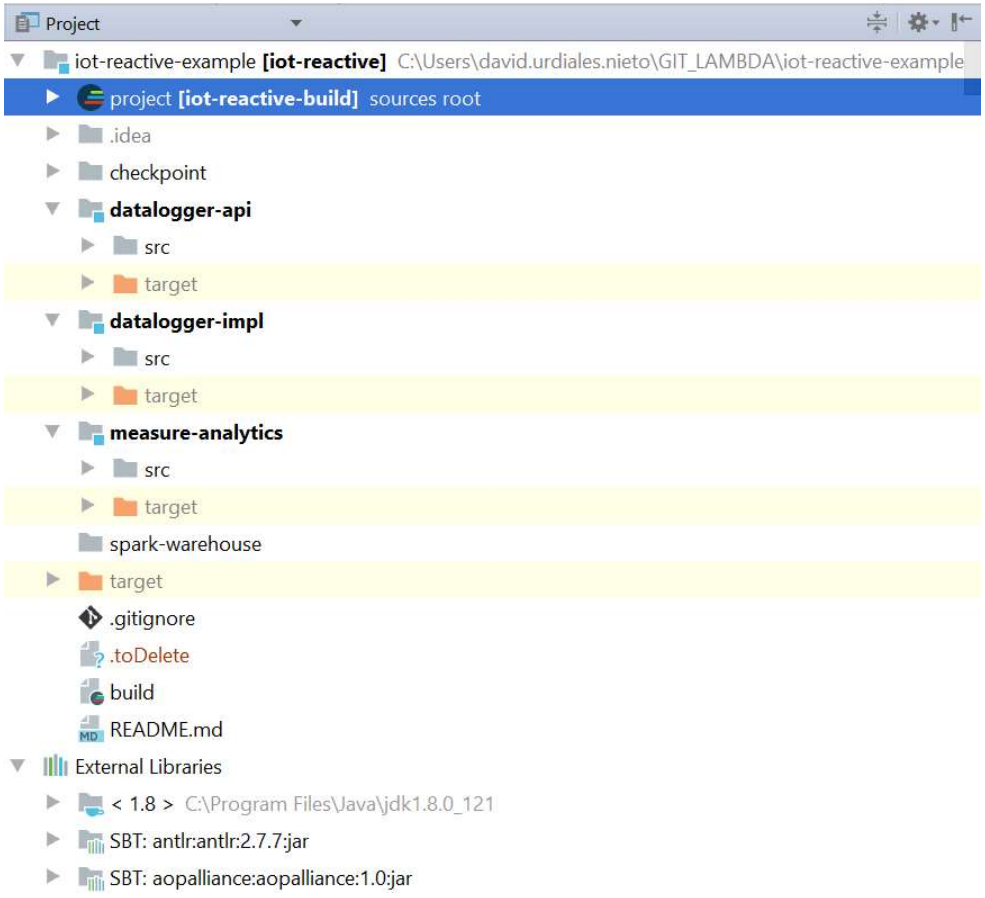
- IoT Example
- Project Structure
- Running Dev Environment

IoT Example



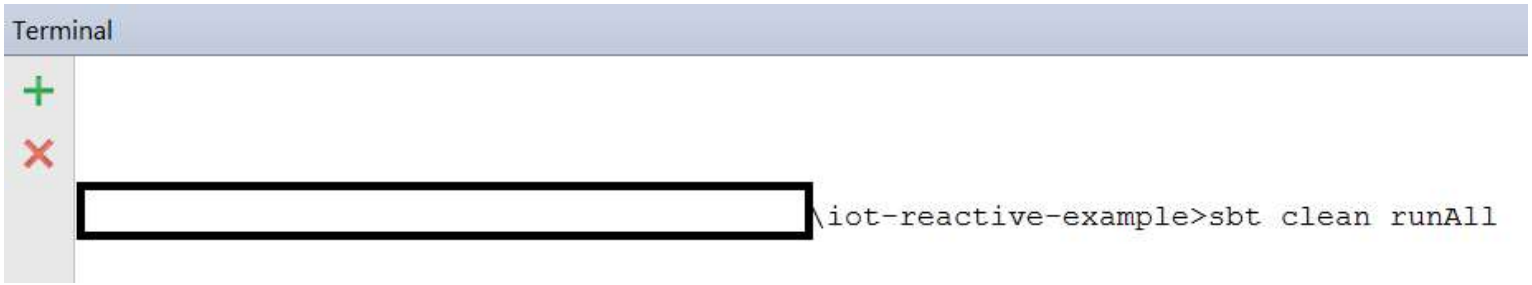
GitHub: https://github.com/durdiales/iot-reactive-example

Project Structure



Running Dev Environment

- One command to start all > sbt clean runAll
- Hot reload code
- IDE Integration



References

- https://www.lagomframework.com
- https://martinfowler.com/articles/microservices.html
- https://docs.microsoft.com/enus/azure/architecture/patterns/cqrs
- https://docs.microsoft.com/enus/azure/architecture/patterns/event-sourcing

Thanks!

Q& A

Thanks to:

José A. Zumaquero Torres

We worked together for this workshop.

David Urdiales Nieto
david.urdiales.nieto@accenture.com