LAGOM

THE NEW JAVA/SCALA MICROSERVICES FRAMEWORK

JWORKS

POWERED BY ORDINA

ABOUT ME

Yannick De Turck
Senior Java Developer
Ordina Belgium
@YannickDeTurck
https://github.com/YannickDeTurck

BLOGPOSTS LAGOM

Lagom: First Impressions and Initial Comparison to Spring Cloud

Lagom 1.2: What's new?

TOPICS

Introduction

Writing microservices

ES & CQRS

Cassandra Persistence

Persistent Read-Side

Publish-Subscribe & Messagebroker Support

Demo: Lagom Shop

INTRODUCTION

MEET LAGOM

- Lightbend's microservices framework
- Focus on reactiveness
- MVP version released on March 2016
- Java & Scala API

DESIGN PHILOSOPHY

- Opinionated
- Message-Driven and Asynchronous
- Streaming first-class concept
- Distributed persistent patterns using ES and CQRS
- Embraces Domain-Driven Design

ARCHITECTURE AND TECHNOLOGIES

- Scala & Java
- Play Framework
- Akka Cluster & Akka Persistence
- sbt & Maven
- Cassandra & RDBs
- Guice & Macwire
- Kafka
- Zookeeper
- ConductR

DEVELOPER PRODUCTIVITY

- Start up with `$runAll`
- Hot code reloading
- Intra-service communication is managed for you
- Infrastructure is set up for you

WRITING MICROSERVICES

PROJECT STRUCTURE

```
helloworld-api           → Microservice API submodule
 └ src/main/scala        → Java source code interfaces with model objects
helloworld-impl          → Microservice implementation submodule
 └ logs                  → Logs of the microservice
 └ src/main/scala        → Java source code implementation of the API submodule
 └ src/main/resources    → Contains the microservice application config
 └ src/test/scala        → Java source code unit tests
logs                     → Logs of the Lagom system
project                  → Sbt configuration files
 └ build.properties      → Marker for sbt project
 └ plugins.sbt           → Sbt plugins including the declaration for Lagom itself
.gitignore               → Gitignore file
build.sbt                → Application build script
```

API INTERFACE

```scala
trait HelloService extends Service {
  def hello(id: String): ServiceCall[NotUsed, String]
  def useGreeting: ServiceCall[GreetingMessage, GreetingResponse]

  override final def descriptor = {
    import Service._
    named("hello")
      .withCalls(
        pathCall("/api/hello/:id", hello _),
        pathCall("/api/hello", useGreeting _)
      )
      .withAutoAcl(true)
  }
}
```

API INTERFACE

```scala
trait HelloService extends Service {
  ...
}

case class GreetingMessage(name: String, message: String)
object GreetingMessage {
  implicit val format: Format[GreetingMessage] = Json.format[GreetingMessage]
}

case class GreetingResponse(greeting: String)
object GreetingResponse {
  implicit val format: Format[GreetingResponse] = Json.format[GreetingResponse]
}
```

API IMPLEMENTATION

```scala
class HelloServiceImpl(implicit ec: ExecutionContext) extends HelloService {
  override def hello(id: String) = ServiceCall { _ =>
    Future(s"Hello, $id")
  }

  override def useGreeting = ServiceCall { request =>
    Future(GreetingResponse(s"${request.message}, ${request.name}"))
  }
}
```

APPLICATION

Wires your code together

```
abstract class HelloApplication(context: LagomApplicationContext)
  extends LagomApplication(context)
    with AhcWSComponents {
  override lazy val lagomServer = serverFor[HelloService](wire[HelloServiceImpl])
}
```

LOADER

Application loader for our application

```
class HelloLoader extends LagomApplicationLoader {
  override def load(context: LagomApplicationContext): LagomApplication =
    new HelloApplication(context) {
      override def serviceLocator: ServiceLocator = NoServiceLocator
    }

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

  override def describeServices = List(
    readDescriptor[HelloService]
  )
}
```

LOADER

Tell Play about your application loader in `application.config`

```
play.crypto.secret = whatever
play.application.loader = com.example.hello.impl.HelloLoader
```

REGISTERING THE MICROSERVICE

build.sbt

```
organization in ThisBuild := "com.example"
version in ThisBuild := "1.0-SNAPSHOT"
scalaVersion in ThisBuild := "2.11.8"

lazy val `hello` = (project in file("."))
  .aggregate(`hello-api`, `hello-impl`)

lazy val `hello-api` = (project in file("hello-api"))
  .settings(
    libraryDependencies ++= Seq(
      lagomScaladslApi
    )
  )

lazy val `hello-impl` = (project in file("hello-impl"))
  .enablePlugins(LagomScala)
  .settings(
    libraryDependencies ++= Seq(
      lagomScaladslTestKit,
      macwire,
      scalaTest
```

TESTING THE MICROSERVICE

```
$ curl localhost:9000/api/hello/Yannick
Hello, Yannick

$ curl -H "Content-Type: application/json" -X POST -d \
    '{"message":"Good evening","name":"Yannick"}' http://localhost:9000/api/hello
{"greeting":"Good evening, Yannick"}
```

TESTING THE MICROSERVICE

First the test setup

```
class HelloServiceSpec extends AsyncWordSpec with Matchers with BeforeAndAfterAll {
  private val server = ServiceTest.startServer(
    ServiceTest.defaultSetup
      .withCassandra(true)
  ) { ctx =>
    new HelloApplication(ctx) with LocalServiceLocator
  }

  val client = server.serviceClient.implement[HelloService]

  override protected def afterAll() = server.stop()
  ...
}
```

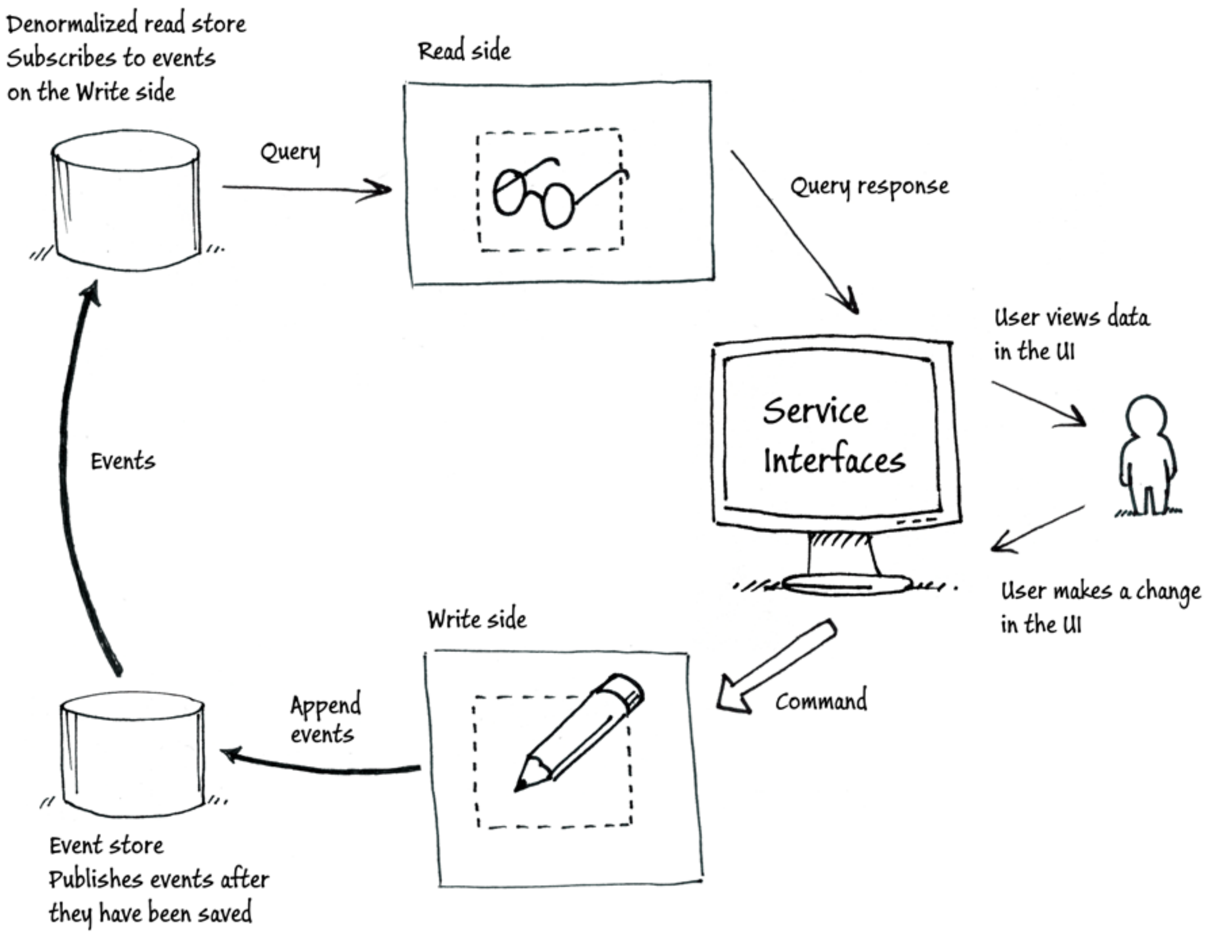
TESTING THE MICROSERVICE

Testing our service

```
class HelloServiceSpec extends AsyncWordSpec with Matchers with BeforeAndAfterAll {
  ...
  "Hello service" should {
    "say hello" in {
      client.hello("Yannick").invoke().map { answer =>
        answer should ===("Hello, Yannick")
      }
    }
    "allow responding with a custom message" in {
      for {
        response <- client.useGreeting.invoke(GreetingMessage("Yannick", "Good evening"))
      } yield {
        response.greeting should ===("Good evening, Yannick")
      }
    }
  }
}
```

ES & CQRS

ES AND CQRS


Denormalized read store
Subscribes to events
on the Write side
Query
Read side
Query response
User views data
in the UI
Service
Interfaces
User makes a change
in the UI
Events
Write side
Command
Append
events
Event store
Publishes events after
they have been saved

EVENT SOURCING

- Capture all changes as domain events
- Handlers process these events

EVENT SOURCING BENEFITS

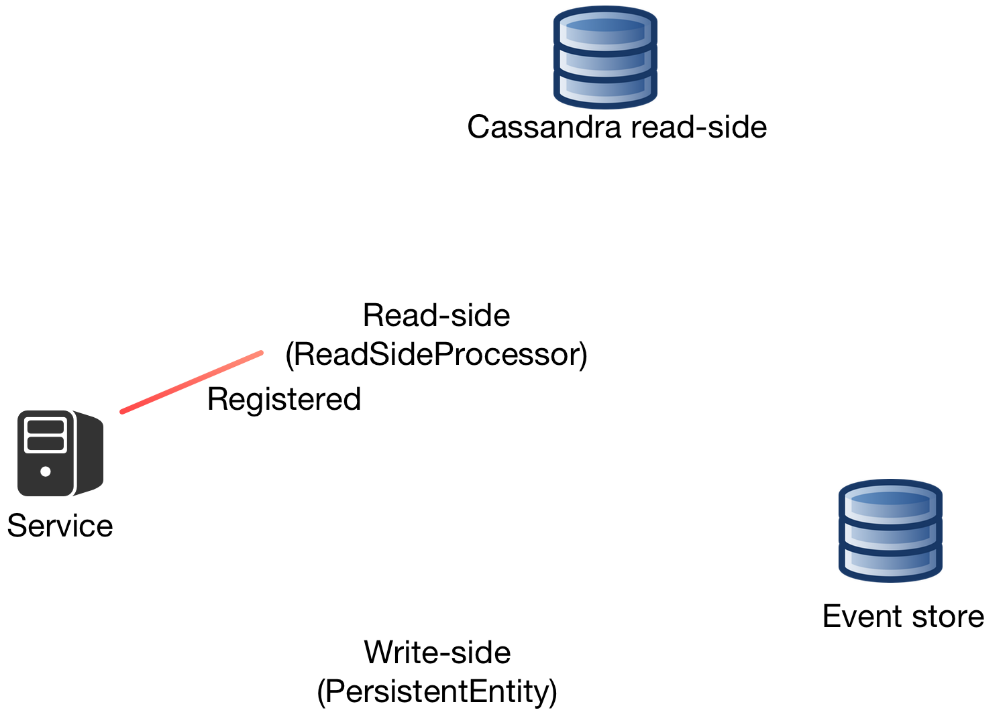
- All events are stored in an event store
- No object-relational impedance mismatch
- Built-in audit mechanism and historical tracing
- Performance, simplification and scalability
- Testability
- Debugging by replaying the event log

COMMAND QUERY RESPONSIBILITY SEGREGATION

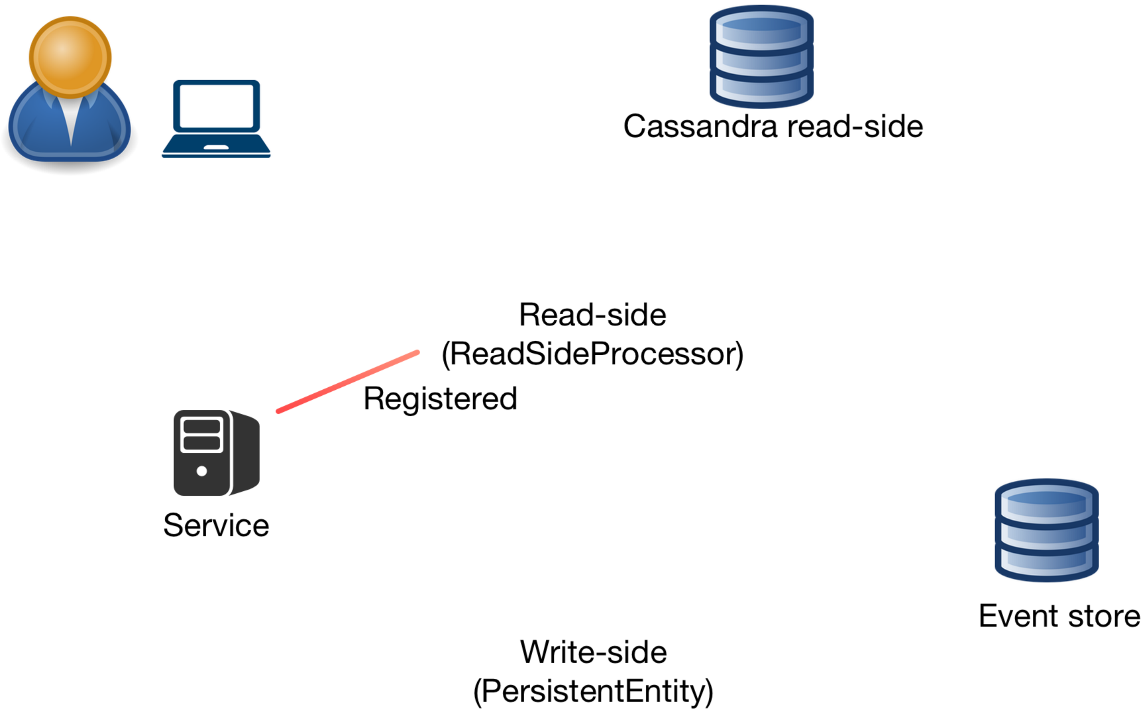
- Separation of write- and read-side
- Scalability
- Different models for write- and read-side
- Eventual consistency

ES AND CQRS IN LAGOM

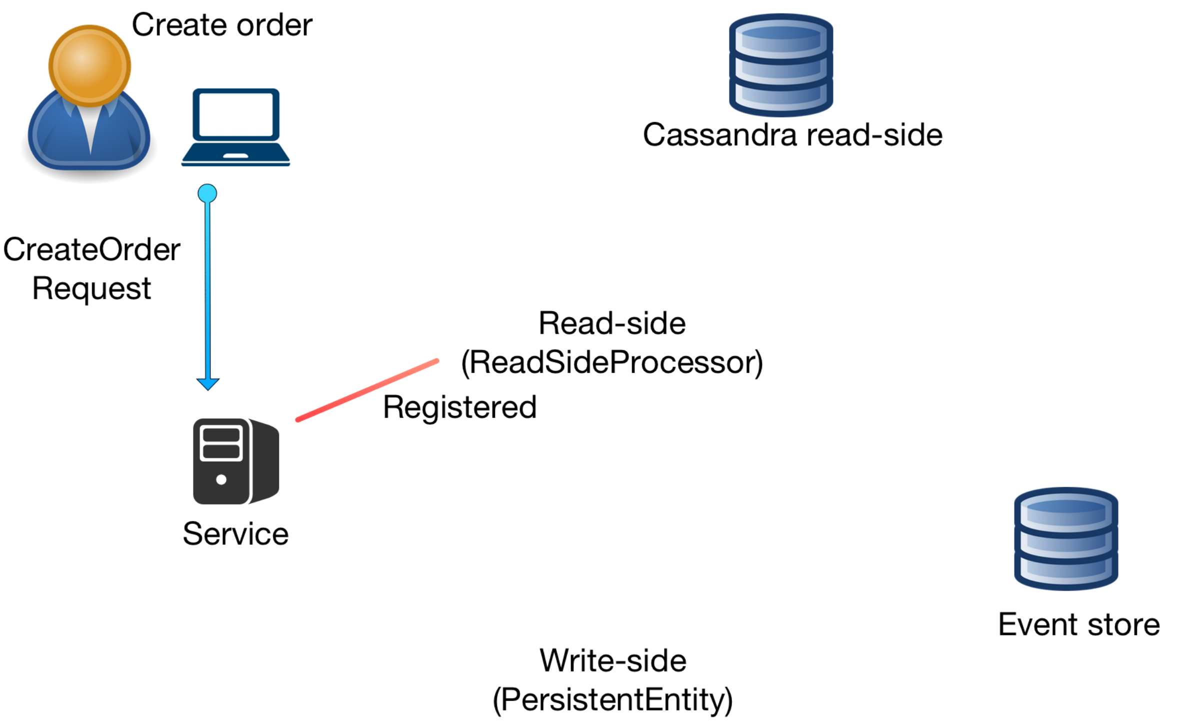
ES AND CQRS IN LAGOM



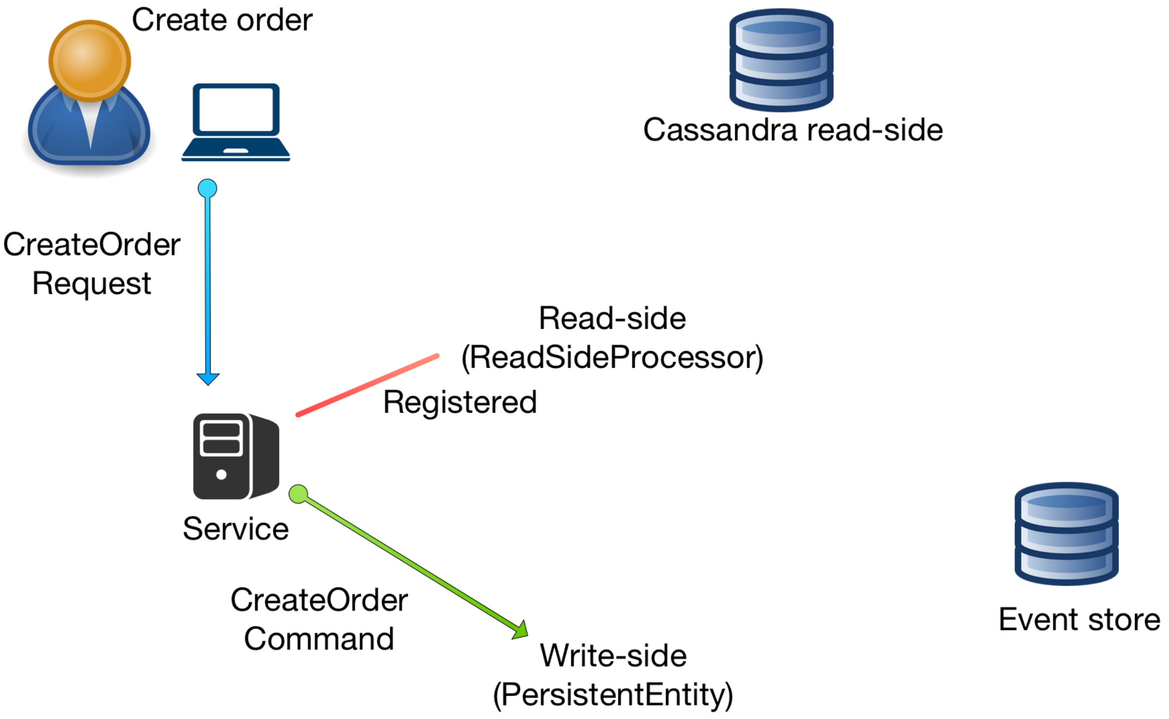
ES AND CQRS IN LAGOM



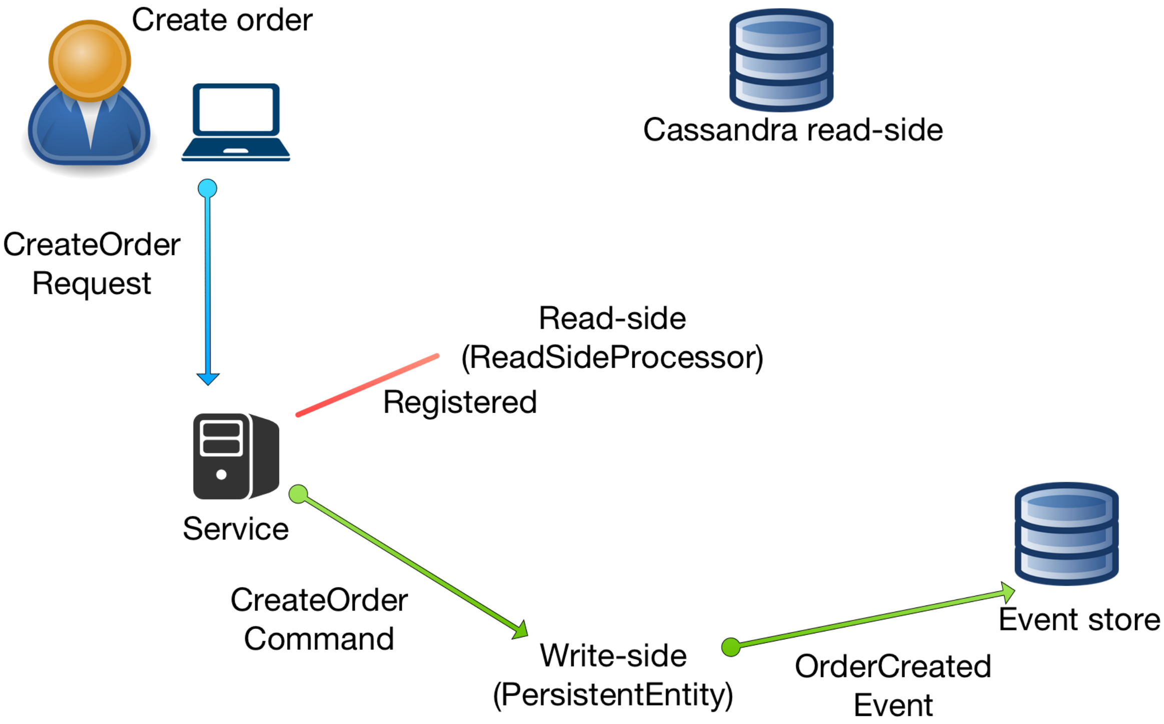
ES AND CQRS IN LAGOM



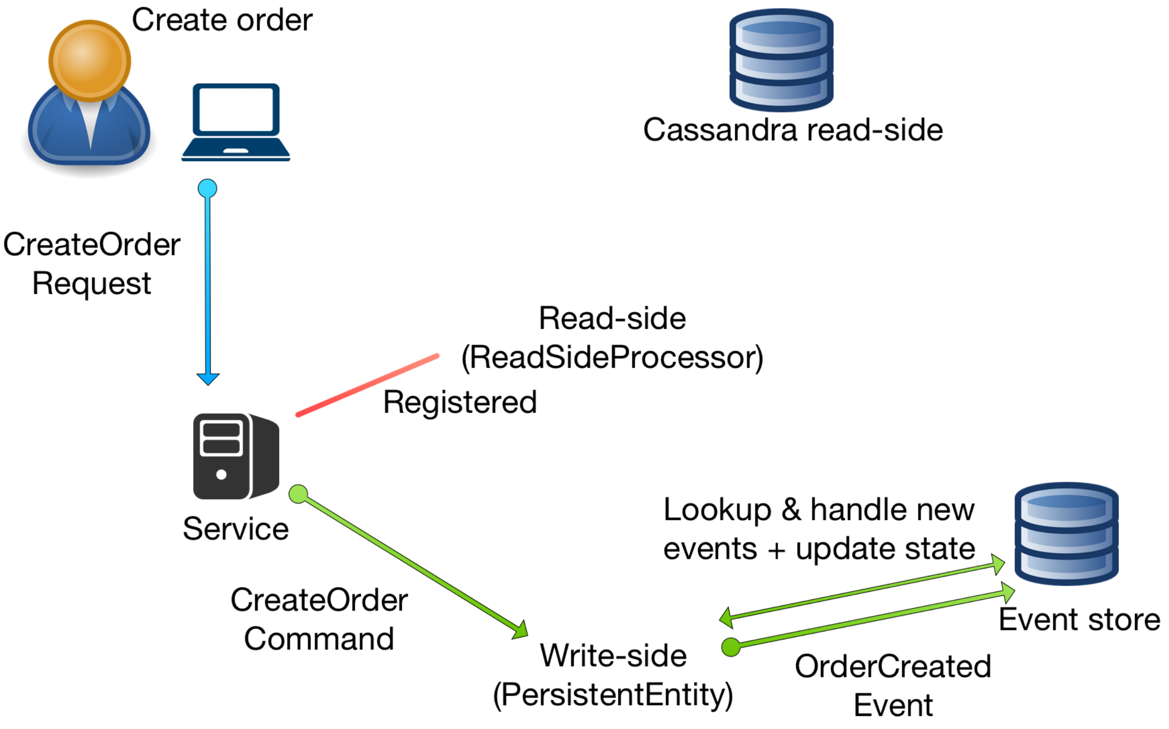
ES AND CQRS IN LAGOM



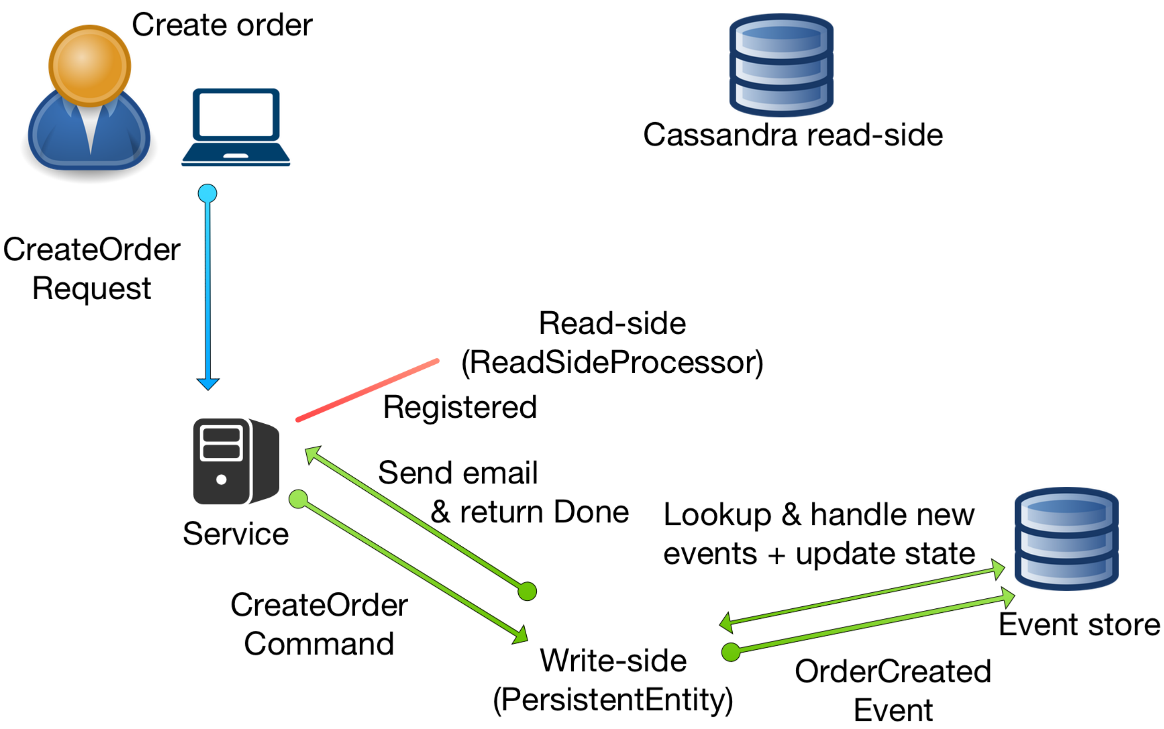
ES AND CQRS IN LAGOM



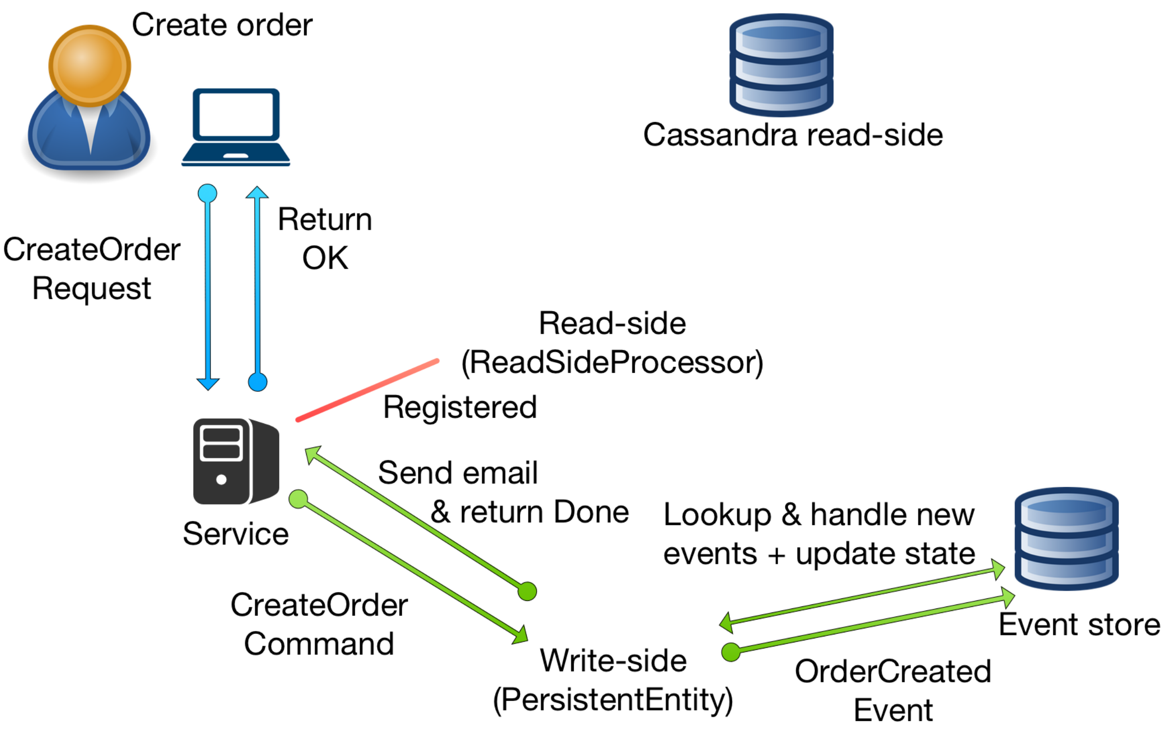
ES AND CQRS IN LAGOM



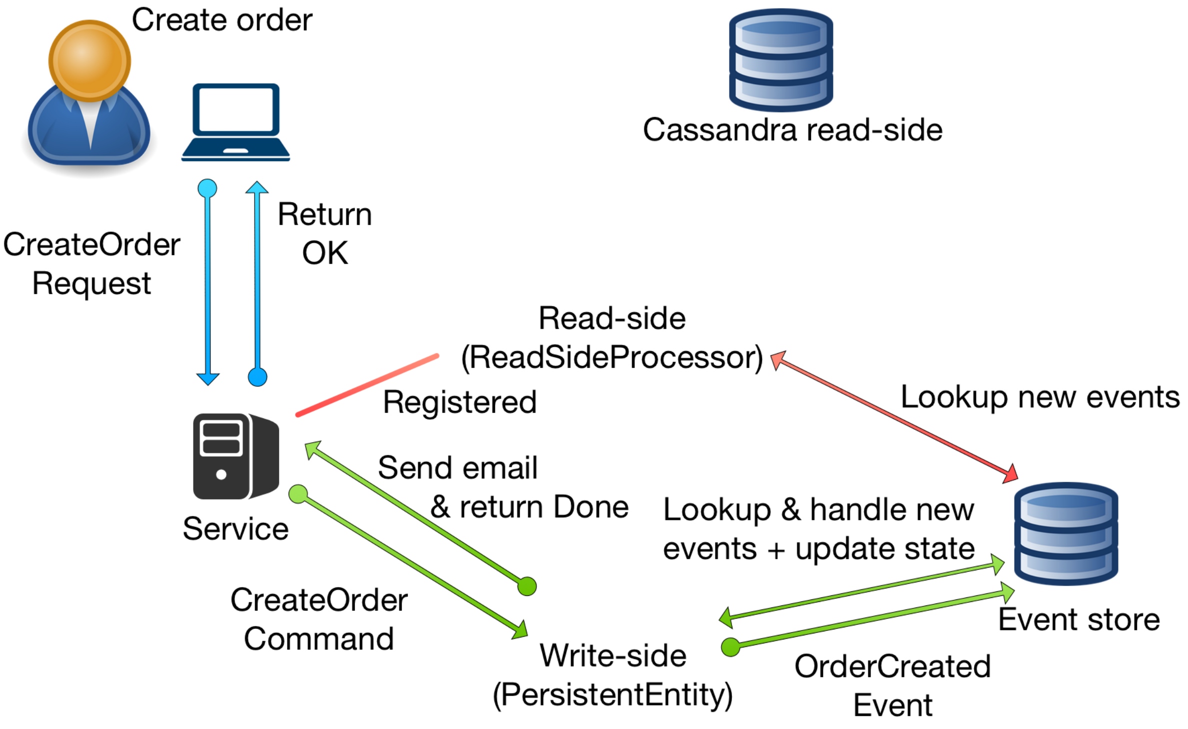
ES AND CQRS IN LAGOM



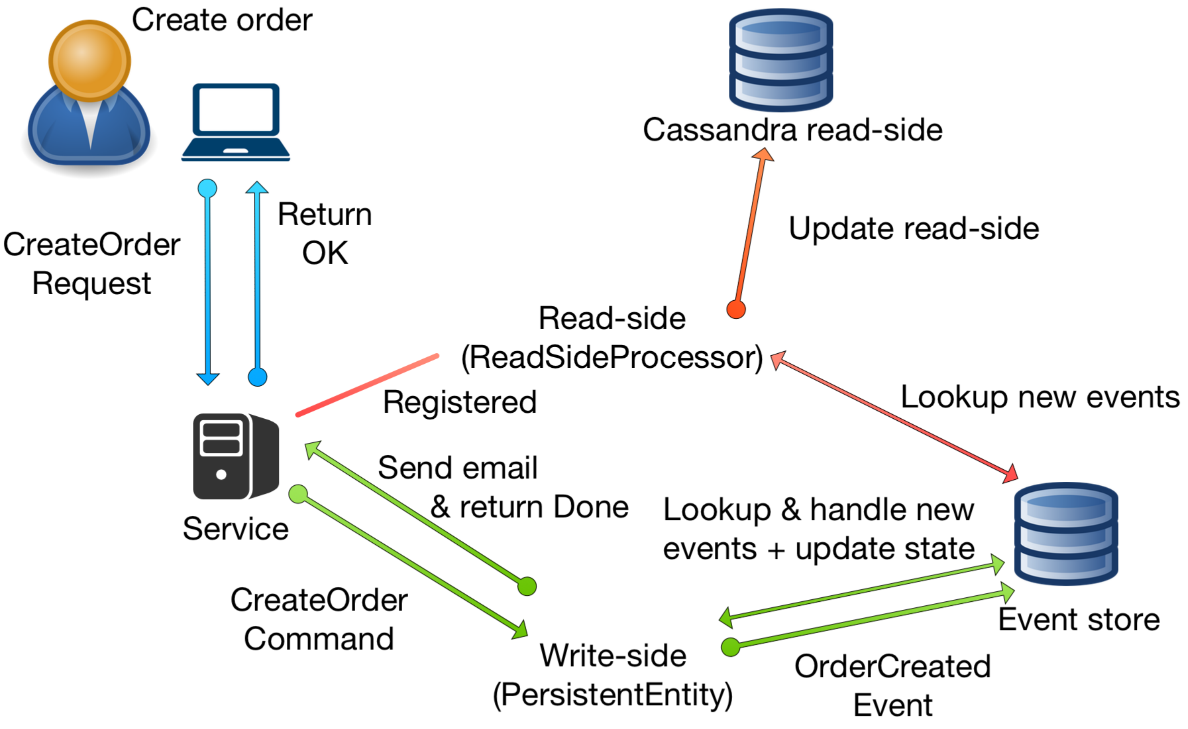
ES AND CQRS IN LAGOM



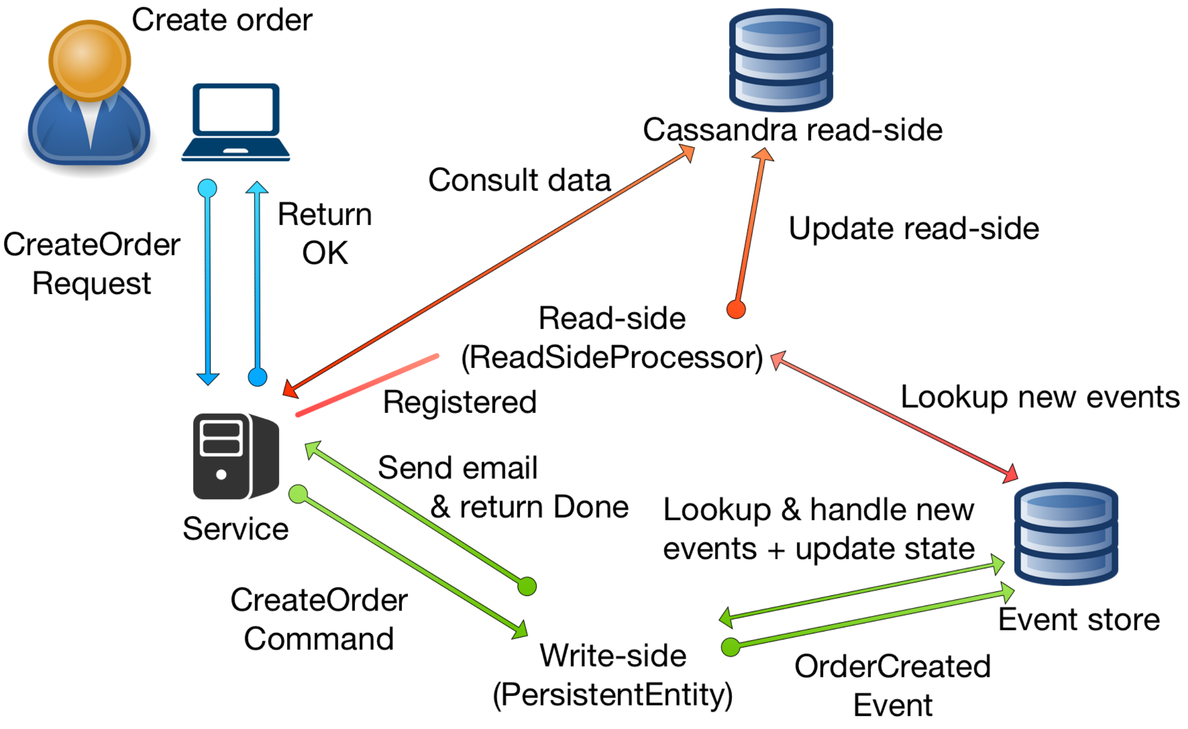
ES AND CQRS IN LAGOM



ES AND CQRS IN LAGOM



ES AND CQRS IN LAGOM



CQRS AND ES INTRODUCTION

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

ES AND CQRS - PERSISTENTENTITY

```
class HelloEntity extends PersistentEntity {
  override type Command = HelloCommand[_]
  override type Event = HelloEvent
  override type State = HelloState
  override def initialState: HelloState = HelloState("Hello", LocalDateTime.now.toString)

  override def behavior: Behavior = {
    case HelloState(message, _) => Actions().onCommand[UseGreetingMessage, Done] {
      case (UseGreetingMessage(newMessage), ctx, state) =>
        ctx.thenPersist(GreetingMessageChanged(newMessage)) { _ =>
          ctx.reply(Done)
        }
    }.onReadOnlyCommand[Hello, String] {
      case (Hello(name), ctx, state) =>
        ctx.reply(s"$message, $name!")
    }.onEvent {
      case (GreetingMessageChanged(newMessage), state) =>
        HelloState(newMessage, LocalDateTime.now().toString)
    }
  }
}
```

ES AND CQRS - STATE

```scala
case class HelloState(message: String, timestamp: String)

object HelloState {
  implicit val format: Format[HelloState] = Json.format
}
```

ES AND CQRS - COMMAND

```
sealed trait HelloCommand[R] extends ReplyType[R]
case class UseGreetingMessage(message: String) extends HelloCommand[Done]

object UseGreetingMessage {
  implicit val format: Format[UseGreetingMessage] = Json.format
}

case class Hello(name: String) extends HelloCommand[String]

object Hello {
  implicit val format: Format[Hello] = Json.format
}
```

ES AND CQRS - EVENT

```scala
sealed trait HelloEvent extends AggregateEvent[HelloEvent] {
  override def aggregateTag = HelloEvent.Tag
}

object HelloEvent {
  val Tag = AggregateEventTag[HelloEvent]
}

case class GreetingMessageChanged(message: String) extends HelloEvent

object GreetingMessageChanged {
  implicit val format: Format[GreetingMessageChanged] = Json.format
}
```

ES AND CQRS - SERVICEIMPL

```
class HelloServiceImpl(persistentEntityRegistry: PersistentEntityRegistry)(implicit ec: Exec
  override def hello(id: String) = ServiceCall { _ =>
    val ref = persistentEntityRegistry.refFor[HelloEntity](id)
    ref.ask(Hello(id))
  }

  override def useGreeting(id: String) = ServiceCall { request =>
    val ref = persistentEntityRegistry.refFor[HelloEntity](id)
    ref.ask(UseGreetingMessage(request.message))
  }
}
```

CASSANDRA PERSISTENCE

CONFIGURATION - KEYSPACE

- Tables are stored in Cassandra keyspaces
- Service should use a unique keyspace name to avoid conflicts
- Lagom has three internal components
  - **Journal**: stores serialized events
  - **Snapshot store**: stores snapshots of the state
  - **Offset store**: used for Cassandra Read-Side support to keep track of the most recent event handled by each read-side processor

CASSANDRA

- Developer: embedded Cassandra gets started by Lagom
  - It's also possible to use an external Cassandra
- Production: dynamically locatable Cassandra server for resiliency

CONFIGURATION - KEYSPACE

`application.conf`

```
cassandra-journal.keyspace = my_service_journal
cassandra-snapshot-store.keyspace = my_service_snapshot
lagom.persistence.read-side.cassandra.keyspace = my_service_read_side
```

CONFIGURATION - KEYSPACE

`application.conf`

```
my-service.cassandra.keyspace = my_service

cassandra-journal.keyspace = ${my-service.cassandra.keyspace}
cassandra-snapshot-store.keyspace = ${my-service.cassandra.keyspace}
lagom.persistence.read-side.cassandra.keyspace = ${my-service.cassandra.keyspace}
```

PERSISTENT READ-SIDE

READ-SIDE DESIGN

- Read-side can be implemented using any datastore
- Similar to the traditional approach of persistence
- One primary rule: only updated in response to receiving events from persistent entities
- Implemented in Lagom via a `ReadSideProcessor`

READSIDEPROCESSOR

- Consume events produced by persistent entities and update the database table
- Tracks events it has handled using offsets
  - Offset tracking is done automatically for you
- To consume events from a read-side, the events need to be tagged
  - Events sharing a tag can be consumed as a sequential, ordered stream of events
- Sharding your read-side event processing load is possible

READSIDEPROCESSOR - TAGGING

HelloEntity.scala

```scala
sealed trait HelloEvent extends AggregateEvent[HelloEvent] {
  override def aggregateTag = HelloEvent.Tag
}

object HelloEvent {
  val NumShards = 20
  val Tag = AggregateEventTag.sharded[HelloEvent](NumShards)
}
```

READSIDEPROCESSOR - BUILD HANDLER

ItemEventProcessor.scala

```scala
private[impl] class ItemEventProcessor(session: CassandraSession, readSide: CassandraReadSid
                                       (implicit ec: ExecutionContext)
  extends ReadSideProcessor[ItemEvent] {

  private var insertItemStatement: PreparedStatement = _

  override def buildHandler: ReadSideProcessor.ReadSideHandler[ItemEvent] = {
    readSide.builder[ItemEvent]("itemEventOffset")
      .setGlobalPrepare(createTables)
      .setPrepare(_ => prepareStatements())
      .setEventHandler[ItemCreated](e => {
        insertItem(e.event.item)
      })
      .build
  }

  override def aggregateTags: Set[AggregateEventTag[ItemEvent]] = ItemEvent.Tag.allTags
  ...
}
```

READSIDEPROCESSOR - CREATING TABLES

ItemEventProcessor.scala

```
...
  private def createTables() = {
    for {
      _ <- session.executeCreateTable(
        """
        CREATE TABLE IF NOT EXISTS item (
          id timeuuid PRIMARY KEY,
          title text,
          description text,
          price decimal
        )
      """)
    } yield Done
  }
  ...
```

READSIDEPROCESSOR - PREPARING STATEMENTS

ItemEventProcessor.scala

```
...
  private def prepareStatements() = {
    logger.info("Preparing statements...")
    for {
      insertItem <- session.prepare(
        """
        INSERT INTO item(
          id,
          title,
          description,
          price
        ) VALUES (?, ?, ?, ?)
      """)
    } yield {
      insertItemStatement = insertItem
      Done
    }
  }

  private def insertItem(item: Item) = {
    logger.info(s"Inserting $item...")
```

QUERYING THE READ-SIDE

ItemRepository.scala

```
private[impl] class ItemRepository(session: CassandraSession)(implicit ec: ExecutionContext)
  def selectAllItems: Future[Seq[Item]] = {
    session.selectAll(
      """
      SELECT * FROM item
    """).map(rows => rows.map(row => convertItem(row)))
  }

  def selectItem(id: UUID) = {
    session.selectOne("SELECT * FROM item WHERE id = ?", id)
  }

  private def convertItem(itemRow: Row): Item = {
    Item(
      itemRow.getUUID("id"),
      itemRow.getString("title"),
      itemRow.getString("description"),
      itemRow.getDecimal("price"))
  }
}
```

PUBLISH-SUBSCRIBE & MESSAGE BROKER SUPPORT

PUBLISH-SUBSCRIBE

- Well-known messaging pattern
- **Publisher**: publishes messages to topics
  - Without knowledge of which receivers there may be
- **Subscriber**: Subscribes and receives messages published to a topic
  - Without knowledge of which publishers there are
- Intra-service

PUBLISH-SUBSCRIBE - EXAMPLE

SensorService

```
trait SensorService extends Service {
  def registerTemperature(id: String): ServiceCall[Temperature, NotUsed]

  def temperatureStream(id: String): ServiceCall[NotUsed, Source[Temperature, NotUsed]]

  def descriptor = {
    import Service._

    named("/sensorservice").withCalls(
      pathCall("/device/:id/temperature", registerTemperature _),
      pathCall("/device/:id/temperature/stream", temperatureStream _)
    )
  }
}
```

PUBLISH-SUBSCRIBE - EXAMPLE

SensorServiceImpl

```scala
class SensorServiceImpl(pubSub: PubSubRegistry) extends SensorService {
  def registerTemperature(id: String) = ServiceCall { temperature =>
    val topic = pubSub.refFor(TopicId[Temperature](id))
    topic.publish(temperature)
    Future.successful(NotUsed.getInstance())
  }

  def temperatureStream(id: String) = ServiceCall { _ =>
    val topic = pubSub.refFor(TopicId[Temperature](id))
    Future.successful(topic.subscriber)
  }
}
```

MESSAGE BROKER SUPPORT

- One service to many other services
- Out of the box support for Apache Kafka
  - Comes with a Zookeeper
- `Topic` published by a service, consumed by other services after subscribing
- API has been designed to be independent of any backend
  - Support for other brokers may be added in the future

KAFKA - SERVICE

```
def itemEvents: Topic[ItemEvent]

override final def descriptor: Descriptor = {
  import Service._

  named("item").withCalls(
    ...
  ).withTopics(
    topic("item-ItemEvent", this.itemEvents)
  ).withAutoAcl(true)
}
```

KAFKA - SERVICEIMPL

```
override def itemEvents: Topic[api.ItemEvent] =
  TopicProducer.taggedStreamWithOffset(ItemEvent.Tag.allTags.toList) { (tag, offset) =>
    logger.info("Creating ItemEvent Topic...")
    registry.eventStream(tag, offset)
      .filter {
        _.event match {
          case x@(_: ItemCreated) => true
          case _ => false
        }
      }.mapAsync(1)(convertEvent)
  }

private def convertEvent(eventStreamElement: EventStreamElement[ItemEvent]): Future[(api.Ite
  eventStreamElement match {
    case EventStreamElement(id, ItemCreated(item), offset) =>
      Future.successful {
        (api.ItemCreated(item.id, item.title, item.description, item.price), offset)
      }
  }
}
```

KAFKA - USAGE

```
itemService.itemEvents.subscribe.withGroupId("dashboard-item-events")
  .atLeastOnce(Flow[ItemEvent].map(event => event.id.toString).collect { case x => x }
    .mapAsync(1)(id => {
      logger.info(s"Received item event $id")
      eventsCache += s"Inserted item $id\n"
      Future(Done.getInstance)
    }))
```

PUBLISH-SUBSCRIBE VS MESSAGE BROKER

- Messages may get lost vs `at-least-once` and `at-most-once` delivery semantics
- Single services cluster vs one service to many other services
- P-S: Subscriber will only receive messages after its subscription has been accepted
- MB: subscriber can consume all the messages since the last message it has consumed
    - Even if the subscriber was offline or down

DEMO: LAGOM SHOP

LAGOM SHOP

- Item Service: Create and lookup items
- Order Service: Create and lookup orders for items
- Play front-end

QUESTIONS?

Lagom Shop & Presentation:
https://github.com/yannickdeturck/lagom-shop-scala

Blogpost Lagom: First Impressions and Initial Comparison to Spring Cloud

Blogpost Lagom 1.2: What's new?

Podcast Lightbend Podcast Ep. 09: Andreas Evers test drives Lagom in comparison with Spring Cloud: http://bit.ly/25XVT8w

THANKS FOR WATCHING!