Lightbend Lagom Microservices “Just Right”

•••

Duncan DeVore - @ironfish
Henrik Engström - @h3nk3
Philly JUG - July 27, 2016

Lightbend

Lagom - [lah-gome]

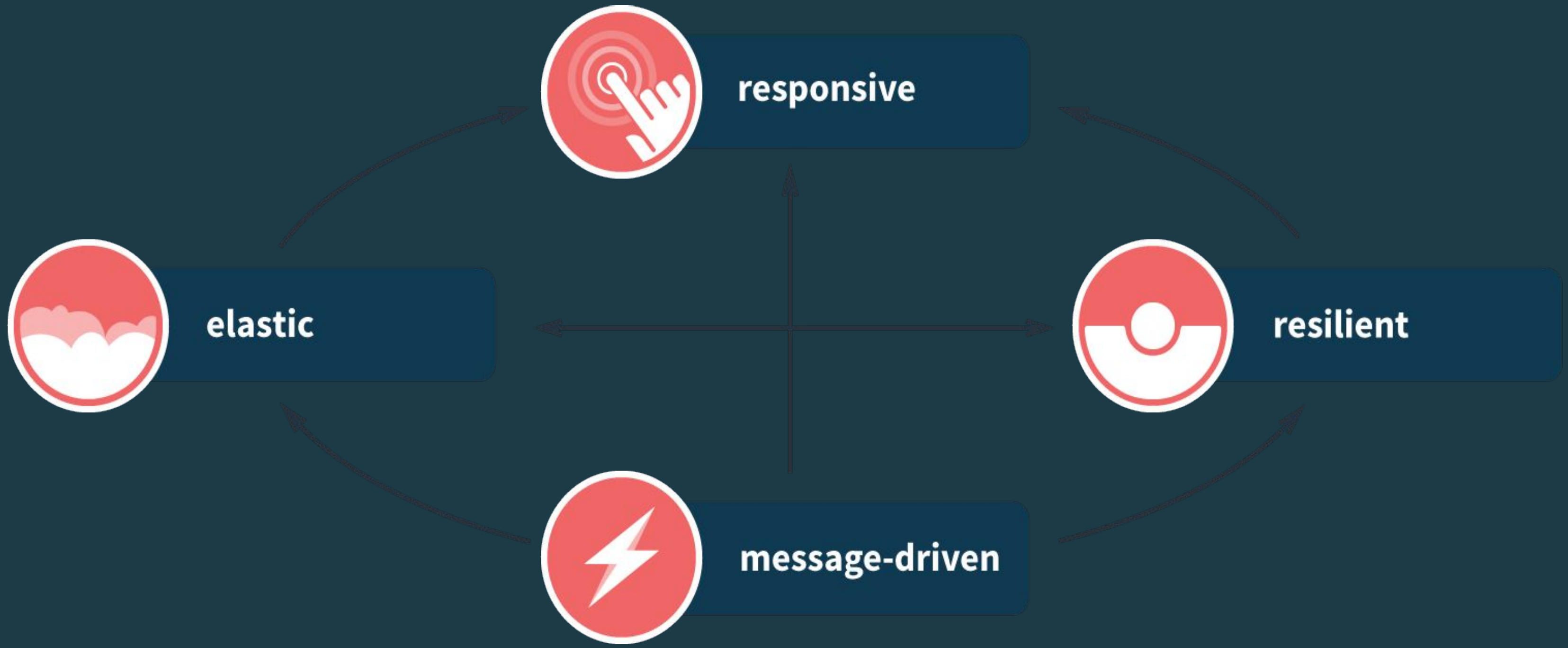
Adequate, sufficient, just right

A great explanation of Lagom: https://www.youtube.com/embed/1tFrRUgFrX4

Agenda

- Overview Reactive
- Why Lagom?
- Lagom Walkthrough
  - Development Environment
  - Service API
  - Persistence API
- Running in Production

REACTIVE OVERVIEW



Lightbend

Why Lagom?

- State-of-the-art technologies in an opinionated way
- Building Microservices is hard!
- Developer experience matters
  - No brittle script to run your services
  - Inter-service communication just works
  - Services are automatically reloaded on code change
- Takes you through to production deployment

Under the hood

- sbt build tool (developer environment)
- Play 2.5
- Akka 2.4 (clustering, streams, persistence)
- Cassandra (default data store)
- Jackson (JSON serialization)
- Guice (DI)

Anatomy of a Lagom project

Each service definition is split into two sbt projects: api & impl

```
hello-world-system          → project root
  └ helloworld-api          → helloworld api project
  └ helloworld-impl         → helloworld implementation
  └ project                 → sbt configuration files
    └ plugins.sbt           → sbt plugins
  └ build.sbt               → the project build file
```

Service API

Lightbend

Service Definition

```java
// this source is placed in your api project
package hello.api;
import com.lightbend.lagom.javadsl.api.*;
import static com.lightbend.lagom.javadsl.api.Service.*;
public interface HelloService extends Service {
    ServiceCall<String, String> sayHello();
    default Descriptor descriptor() {
        return named("helloservice").withCalls(
            namedCall("hello", this::sayHello)
        );
    }
}
```

Lightbend

ServiceCall explained

```
interface ServiceCall<Request, Response> {
    CompletionStage<Response> invoke(Request request);
}
```

- ServiceCall contains two types:
  - Request: type of incoming request message (e.g. String)
  - Response: type of outgoing response message (e.g. String)
- CompletionStage: a promise of a value in the future
- JSON is the default serialization format for request/response messages
- There are two kinds of request/response messages: Strict and Streamed

Strict Messages

```java
ServiceCall<String, String> sayHello();


default Descriptor descriptor() {
    return named("helloservice").withCalls(
        namedCall("hello", this::sayHello)
    );
}
```

Strict messages are *fully* buffered into memory

Streamed Messages

```
ServiceCall<String, Source<String, ?>> tick(int interval);


default Descriptor descriptor() {
    return named("clock").withCalls(
        pathCall("/tick/:interval", this::tick)
    );
}
```

- A streamed message is of type **Source** (an Akka streams API)
- Back-pressured, asynchronous handling of messages
- WebSocket is the selected transport protocol

Service Implementation

Remember this service definition?

```
// this source is placed in your api project
public interface HelloService extends Service {
    ServiceCall<String, String> sayHello();

    default Descriptor descriptor() {
        return named("helloservice").withCalls(
            namedCall("hello", this::sayHello)
        );
    }
}
```

Here is the **Service Implementation**

```java
// this source is placed in your implementation project
package hello.impl;
import com.lightbend.lagom.javadsl.api.*;
import static java.util.concurrent.CompletableFuture.completedFuture;
import hello.api.HelloService;

public class HelloServiceImpl implements HelloService {
    public ServiceCall<String, String> sayHello() {
        return name -> completedFuture("Hello " + name);
    }
}
```

Register Service Implementation

```java
// this source is placed in your implementation project
package hello.impl;
import com.google.inject.AbstractModule;
import com.lightbend.lagom.javadsl.server.ServiceGuiceSupport;
import hello.api.HelloService;
public class HelloModule extends AbstractModule implements
        ServiceGuiceSupport {
    protected void configure() {
        bindServices(serviceBinding(HelloService.class,
            HelloServiceImpl.class));
    }
}
```

Lightbend

Register Service Implementation - part II

```
// Instruct Lagom to load this module by adding it to
// the application.conf file:

play.modules.enabled += hello.impl.HelloModule
```

Lightbend

Demo Time

Persistence API

Lightbend

Principles

- Each service *owns* its data
  - Only the service has direct access to the DB
- We advocate the use of Event Sourcing (ES) and CQRS
  - ES: Capture *all* state's changes as events
  - CQRS: separate models for write and read

Event Sourcing/CQRS:

Command Query Responsibility Segregation

*"CQRS is simply the creation of **two objects** where there was previously only one. The separation occurs based upon whether the methods are a command or a query (the same definition that is used by Meyer in Command and Query Separation: a command is any method **(object)** that mutates state and a query is any method **(object)** that returns a value)"*

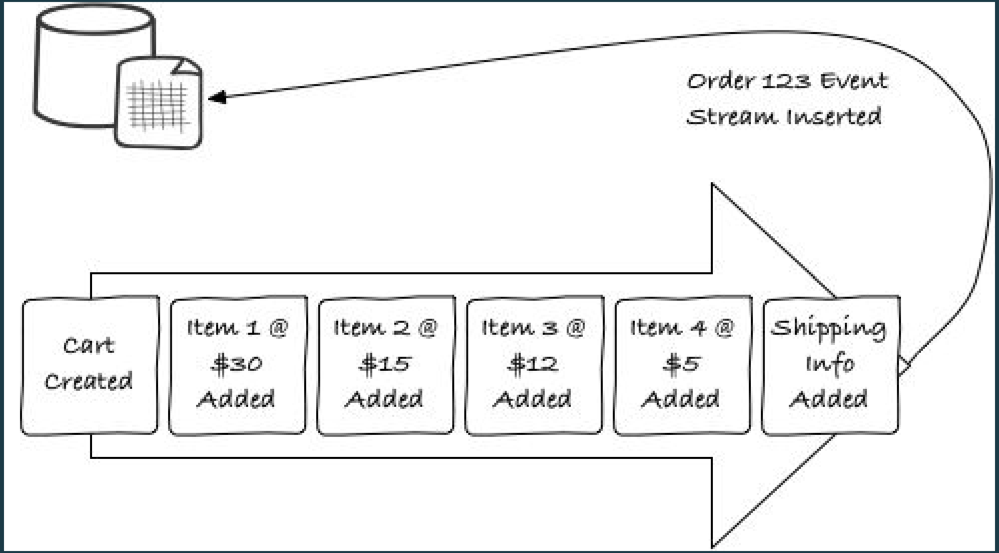
- Greg Young

Event Sourcing/CQRS:

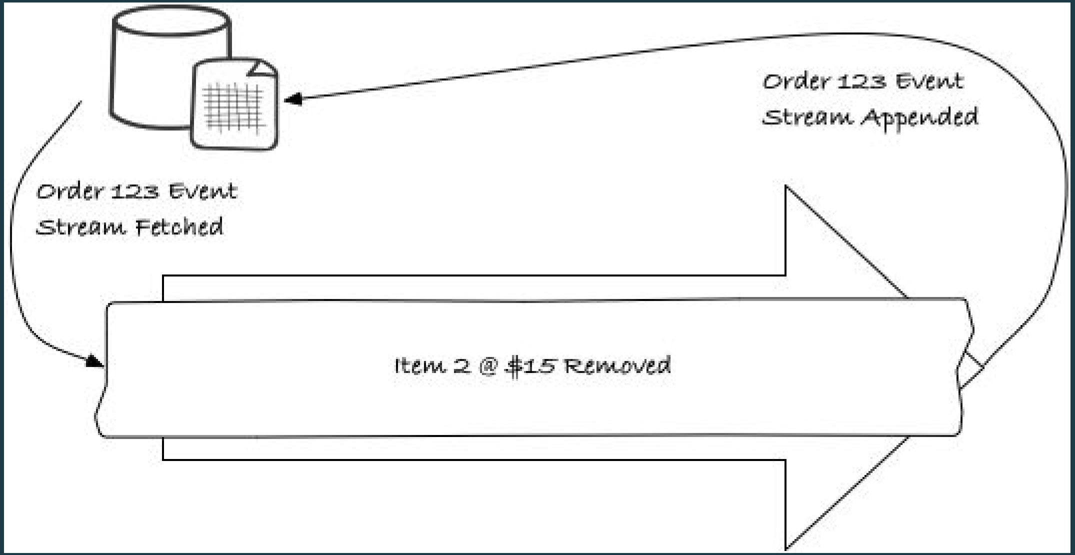
Historical behavior is captured

- Behavioral by nature
- Convert valid commands into 1/n events
- Current state is not persisted
- Current state is derived
- Append only store

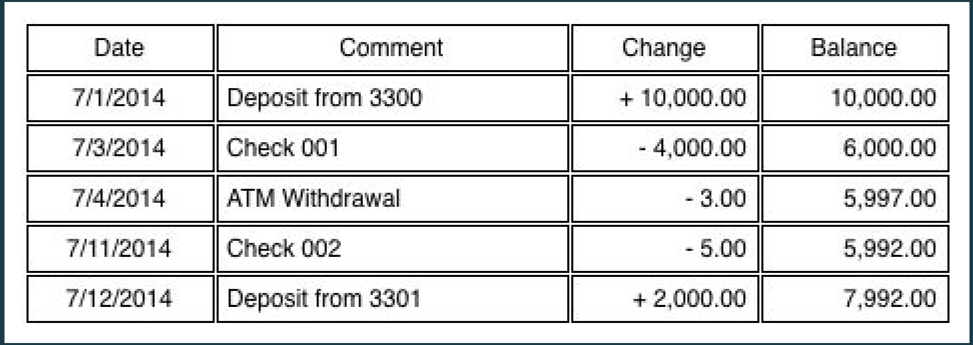
Event Sourcing/CQRS:



Event Sourcing/CQRS:


Order 123 Event Stream Fetched
Order 123 Event Stream Appended
Item 2 @ $15 Removed

Event Sourcing/CQRS:



Date	Comment	Change	Balance
7/1/2014	Deposit from 3300	+ 10,000.00	10,000.00
7/3/2014	Check 001	- 4,000.00	6,000.00
7/4/2014	ATM Withdrawal	- 3.00	5,997.00
7/11/2014	Check 002	- 5.00	5,992.00
7/12/2014	Deposit from 3301	+ 2,000.00	7,992.00

Event Sourcing/CQRS:

- Create your own *Command* and *Event* classes
- Subclass **PersistentEntity**
    - Define *Command* and *Event* handlers
    - Can be accessed from anywhere in the cluster
    - (corresponds to an *Aggregate Root* in DDD)

Benefits of Event Sourcing/CQRS

- Allows you to time travel
- Audit log
- Future business opportunities
- No need for ORM
- Implicit read/write optimization
- No database migration script, ever!
- Performance & Scalability
- Testability & Debuggability

Persistence Example

Let’s implement the *add a friend* functionality in chirper

Event Sourcing: Example

1. Create a **AddFriend** command class
2. Create a **FriendAdded** event class
3. Define a **FriendEntity** holding the state (i.e., who are the friends of a specific user)
    a. Create a *command handler* for the **AddFriend** command
    b. Create an *event handler* for the **FriendAdded** event

Event Sourcing: Example cont’d

1. Create a **AddFriend** command class
2. Create a **FriendAdded** event class
3. Define a **FriendEntity** holding the state of what are friends of a given user
4. Create a *command handler* for the **AddFriend** command
5. Create an *event handler* for the **FriendAdded** event

Event Sourcing: Example cont’d

```java
public interface FriendCommand extends Jsonable {
    // other commands . . .
    @SuppressWarnings("serial")
    @Immutable
    @JsonDeserialize
    public final class AddFriend implements FriendCommand,PersistentEntity.ReplyType<Done> {
        public final String friendUserId;

        @JsonCreator
        public AddFriend(String friendUserId) {
            this.friendUserId = Preconditions.checkNotNull(friendUserId, "friendUserId");
        }
        // equals, equalTo, hashCode, toString love . . .
    }
}
```

Event Sourcing: Example cont’d

1. Create a **AddFriend** command class
2. Create a **FriendAdded** event class
3. Define a **FriendEntity** holding the state of what are friends of a given user
4. Create a *command handler* for the **AddFriend** command
5. Create an *event handler* for the **FriendAdded** event

Event Sourcing: Example cont'd

```java
public interface FriendEvent extends Jsonable, AggregateEvent<FriendEvent> {
    // other commands . . .
    @SuppressWarnings("serial")
    @Immutable
    @JsonDeserialize
    public class FriendAdded implements FriendEvent {
        //...
        @JsonCreator
        public FriendAdded(String userId, String friendId, Optional<Instant> timestamp) {
            this.userId = Preconditions.checkNotNull(userId, "userId");
            this.friendId = Preconditions.checkNotNull(friendId, "friendId");
            this.timestamp = timestamp.orElseGet(() -> Instant.now());
        }
        // equals, equalTo, hashCode, toString love . . .
    }
}
```

Lightbend

Event Sourcing: Example cont’d

1. Create a **AddFriend** command class
2. Create a **FriendAdded** event class
3. Define a **FriendEntity** holding the state of what are friends of a given user
4. Create a *command handler* for the **AddFriend** command
5. Create an *event handler* for the **FriendAdded** event

Event Sourcing: Example cont’d

```
public class FriendEntity extends PersistentEntity<FriendCommand, FriendEvent, FriendState> {

    @Override
    public Behavior initialBehavior(Optional<FriendState> snapshotState) {
        BehaviorBuilder b = newBehaviorBuilder(snapshotState.orElse(
            new FriendState(Optional.empty())));

        // define more command and event handlers

        return b.build();
    }
}
```

Lightbend

Event Sourcing: Example cont’d

```
@SuppressWarnings("serial")
@Immutable
@JsonDeserialize
public final class FriendState implements Jsonable {
    public final Optional<User> user;

    @JsonCreator
    public FriendState(Optional<User> user) { this.user = Preconditions.checkNotNull(user, "user"); }

    public FriendState addFriend(String friendUserId) {
        if (!user.isPresent())
            throw new IllegalStateException("friend can't be added before user is created");
        PSequence<String> newFriends = user.get().friends.plus(friendUserId);
        return new FriendState(Optional.of(new User(user.get().userId, user.get().name, Optional.of(newFriends))));
    }
    // equals, equalTo, hashCode, toString love . . .
}
```

Event Sourcing: Example cont’d

1. Create a **AddFriend** command class
2. Create a **FriendAdded** event class
3. Define a **FriendEntity** holding the state of what are friends of a given user
4. Create a *command handler* for the **AddFriend** command
5. Create an *event handler* for the **FriendAdded** event

Event Sourcing: Example cont’d

```java
public class FriendEntity extends PersistentEntity<FriendCommand, FriendEvent, FriendState> {
    @Override
    public Behavior initialBehavior(Optional<FriendState> snapshotState) {
        // CommandHandler
        BehaviorBuilder b = newBehaviorBuilder(snapshotState.orElse(new FriendState(Optional.empty())));
        // Command handlers are invoked for incoming messages (commands).
        // A command handler must "return" the events to be persisted (if any).
        b.setCommandHandler(AddFriend.class, (cmd, ctx) -> {
            if (!state().user.isPresent()) {
                ctx.invalidCommand("User " + entityId() + " is not  created");
                return ctx.done();
            } else if (state().user.get().friends.contains(cmd.friendUserId)) {
                ctx.reply(Done.getInstance());
                return ctx.done();
            } else {
                return ctx.thenPersist(new FriendAdded(getUserId(), cmd.friendUserId), evt -> ctx.reply(Done.getInstance()));
            }
        });
        // more command/event handlers
    }
}
```

Lightbend

Event Sourcing: Example cont'd

1. Create a **AddFriend** command class
2. Create a **FriendAdded** event class
3. Define a **FriendEntity** holding the state of what are friends of a given user
4. Create a *command handler* for the **AddFriend** command
5. Create an *event handler* for the **FriendAdded** event

Event Sourcing: Example cont’d

```
b.setEventHandler(FriendAdded.class, evt -> state().addFriend(evt.friendId));
```

No side-effects in the event handler!

Event Sourcing: Example cont'd

```java
public interface FriendService extends Service {
    // other service calls . . .

    ServiceCall<FriendId, NotUsed> addFriend(String userId);

    @Override
    default Descriptor descriptor() {
        return named("friendservice").withCalls(
            pathCall("/api/users/:userId", this::getUser),
            namedCall("/api/users", this::createUser),
            pathCall("/api/users/:userId/friends", this::addFriend),
            pathCall("/api/users/:userId/followers", this::getFollowers)
        );
    }
}
```

Lightbend

Event Sourcing: Example cont'd

```java
public class FriendServiceImpl implements FriendService {
    private final PersistentEntityRegistry persistentEntities;
    private final CassandraSession db;

    @Inject
    public FriendServiceImpl(PersistentEntityRegistry persistentEntities, CassandraReadSide readSide, CassandraSession db) {
        this.persistentEntities = persistentEntities;
        this.db = db;
        // at service startup we must register the needed entities
        persistentEntities.register(FriendEntity.class);
        readSide.register(FriendEventProcessor.class);
    }

    @Override
    public ServiceCall<FriendId, NotUsed> addFriend(String userId) {
        return request -> { return friendEntityRef(userId).ask(new AddFriend(request.friendId)).thenApply(ack -> NotUsed.getInstance()); };
    }

    private PersistentEntityRef<FriendCommand> friendEntityRef(String userId) {
        PersistentEntityRef<FriendCommand> ref = persistentEntities.refFor(FriendEntity.class, userId);
        return ref;
    }
}
```

Event Sourcing/CQRS:

- Tightly integrated with Cassandra
- Create the query tables:
  - Subclass **CassandraReadSideProcessor**
  - Consumes events produced by the **PersistentEntity** and updates tables in Cassandra optimized for queries
- Retrieving data: Cassandra Query Language
  - e.g., `SELECT id, title FROM postsummary`

Running in Production

- sbt-native packager is used to produce zip, MSI, RPM, Docker
- Lightbend ConductR* (our container orchestration tool)
- Lightbend Reactive Platform*
    - Split Brain Resolver (for Akka cluster)
    - Lightbend Monitoring

**Requires a Lightbend subscription (but it is free to use during development)*

Current[Lagom]

- Current version is 1.0.0 (released yesterday!)
- Java API, but no Scala API yet
  - We are working on the Scala API
  - But using Scala with the Java API works quite well! https://github.com/dotta/activator-lagom-scala-chirper

Future[Lagom]

- Maven support
- Message broker integration (e.g. Kafka)
- Scala API
- Support for other cluster orchestration tools
  - Want Kubernetes support? Contribute! https://github.com/huntc/kubernetes-lib
- Support for writing integration tests
- Swagger integration

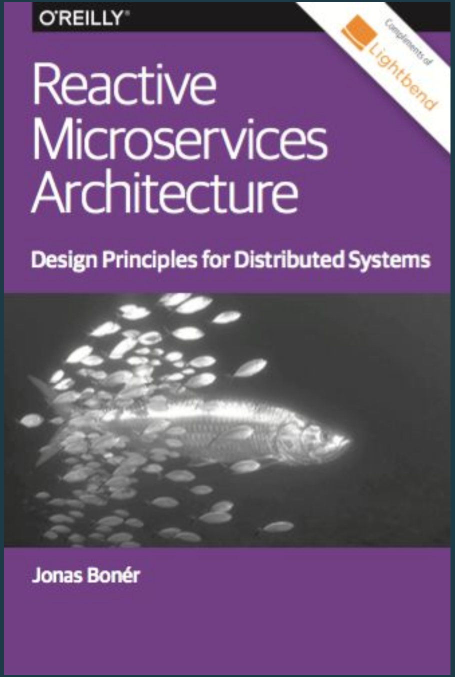
Lightbend

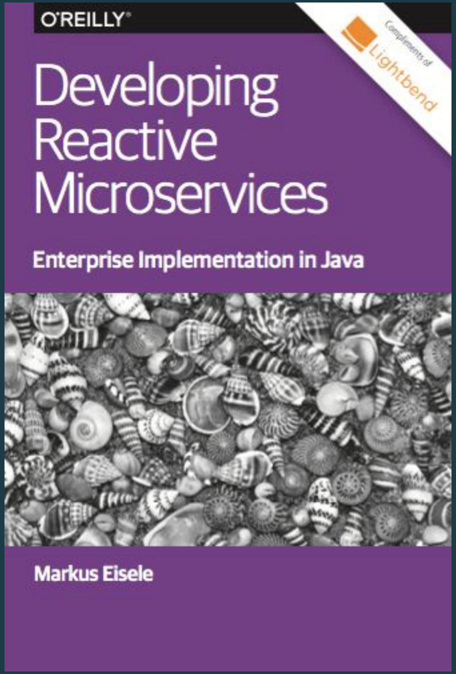
Next: Seq[Step]

- Try Lagom yourself
  - https://lightbend.com/lagom
- Lagom on Github
  - https://github.com/lagom/lagom
- Read Jonas Bonér's *free* ebook *Reactive Services Architecture*
  - https://lightbend.com/reactive-microservices-architecture
- Great presentation by Greg Young on why you should use ES
  - https://www.youtube.com/watch?v=JHGkaShoyNs

Upgrade your grey matter

Two free O'Reilly eBooks by Lightbend


O'REILLY
Complements of Lightbend
Reactive Microservices Architecture
Design Principles for Distributed Systems
Jonas Bonér


O'REILLY
Complements of Lightbend
Developing Reactive Microservices
Enterprise Implementation in Java
Markus Eisele

http://bit.ly/ReactiveMicroservice http://bit.ly/DevelopReactiveMicroservice

Q&A

Some commonly asked questions...

- CRUD vs ES
- Microservices Design - Push vs Pull
- How does this compare to Serverless/FaaS?

Lightbend

Thank you for listening!

@h3nk3
@ironfish
@lagom
@lightbend

Lightbend