# Reactive Microservices With Lagom and Java

BY **MARKUS EISELE**

CONTENTS



This Refcard helps you with your first steps in Lagom. Lagom is a framework that helps you to build reactive microservices.

Most microservices frameworks focus on helping you build fragile, single instance microservices - which, by definition, aren't scalable or resilient. Lagom helps you build microservices as systems — Reactive systems, to be precise — so that your microservices are elastic and resilient from the start and don't require extra plumbing around them.

Building Reactive Systems can be hard, but Lagom abstracts the complexities away. Akka and Play do the heavy lifting underneath and developers can focus on a simpler event-driven programming model on top, while benefitting from a message-driven system under the hood. Lagom provides an opinionated framework that acts like guide rails to speed you along the journey. Lagom tools and APIs simplify development and deployment of a system that includes microservices.

Reactive Microservices Architecture: Design Principles for Distributed Systems, by Jonas Bonér, describes the base principles behind modern systems and how they should be built.

"As we detangle our systems, we shift the power from central governing bodies to smaller teams who can seize opportunities rapidly and stay nimble because they understand the software within **well-defined** boundaries that they control." *(Jonas Bonér)*

## REACTIVE MICROSERVICES REQUIREMENTS

Looking at microservices-based architectures, you quickly realize that they have various requirements that need to be met. This includes isolation and service autonomy, but ultimately leads to the traits that are defined by The Reactive Manifesto (reactivemanifesto.org). Lagom is asynchronous by default — its APIs make inter-service communication via streaming a first-class concept. All Lagom APIs use the asynchronous IO capabilities of Akka Stream for asynchronous streaming; the Java API uses `JDK8 CompletionStage` for asynchronous computation. With built-in support for Event Sourcing (ES) with Command Query Responsibility Segregation (CQRS), Lagom favors an event-sourced architecture for data persistence. Persistent Entity is Lagom's implementation of event sourcing.

The Lagom framework includes libraries and a development environment that support you from development to deployment:

- During development, a single command builds your project and starts all your services and the supporting Lagom infrastructure. It hot-reloads when you modify code. The development environment allows you to bring up a new service or join an existing Lagom development team in just minutes.
- You can create microservices using Java or Scala. Lagom offers an especially seamless experience for communication between microservices. Service location, communication protocols, and other issues are handled by Lagom transparently, maximizing convenience and productivity. Lagom supports Event sourcing and CQRS (Command Query Responsibility Segregation) for persistence.
- You can deploy your microservices on your platform of choice.

## DESIGNING YOUR MICROSERVICES SYSTEM

First, identify a need for a simple microservice that can consume asynchronous messages. It needn't be complex or even provide a lot of value. The simplicity reduces the risk associated with deployment and can provide a quick win. Next, at the architectural level, pull out a core service that can be compartmentalized. Divide it into a system of microservices.

*Continued on next page*

# Making Reactive Microservices easy.

Leverage a decade of hard-won knowledge building microservices with Akka and Play. Lagom, as a lightweight layer on top of Akka and Play, preserves all of their advantages, proven and hardened over time, so it's extremely easy to do the right thing.

- Hot-reload environment
- Best practices guide development
- Natively asynchronous
- Single responsibility made simple
- Container-ready service

## Learn how to build Reactive Microservices with Lagom (**free eBook**)

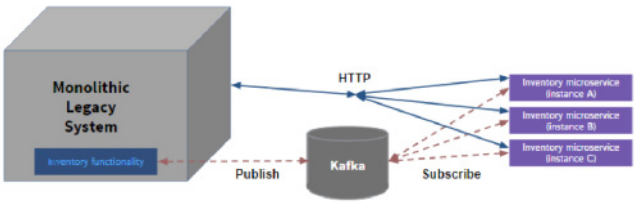

**DOWNLOAD EBOOK**

Still chugging along with a monolithic enterprise system that's difficult to scale and maintain, and even harder to understand? In this concise report, Lightbend CTO Jonas Bonér explains why microservice-based architecture that consists of small, independent services is far more flexible than the traditional all-in-one systems that continue to dominate today's enterprise landscape.

"Java finally gets microservices tools." - InfoWorld

lagom Lightbend

When you attack the problem a piece at a time, you and your team will learn as you go and will become increasingly effective. Employing approaches such as Domain-driven Design (DDD) can help your organization deal with the complexity inherent in enterprise systems. DDD encourages breaking large models into Bounded Contexts. Each Bounded Context defines a boundary that applies to a particular team, addresses specific usage, and includes the data schema and physical elements necessary to materialize the system for that context. Bounded Contexts allow small teams to focus on one context at a time and work in parallel.




## SERVICES AND COMMUNICATION

Whether you're building a new system from scratch or dissecting a monolith into microservices, answers to the following questions will help you make good choices.

- Does this service do only one thing?
- Is this service autonomous?
- Does this service own its own data?

You should arrive at services that are isolated and autonomous. Such services communicate with each other (inter-service) by sending messages over a network. To achieve performance and resilience, you will often run multiple instances of the same service, typically on different nodes, and such intra-service communication also goes over the network. In addition, third-party and legacy systems might also consume or provide information for your microservice system.

## INTER-SERVICE COMMUNICATION

While similar, inter- and intra-service communication have very different needs, and you need multiple implementation options. **Inter-service communication** must use loosely-coupled protocols and message formats to maintain isolation and autonomy, while **intra-service communication** can take advantage of mechanisms that have less overhead and better performance.

Service calls, either synchronous or asynchronous (streaming), allow services to communicate with each other using published APIs and standard protocols (HTTP and WebSockets). Lagom services are described by an interface, known as a service descriptor. This interface not only defines how the service is invoked and implemented, it also defines the metadata that describes how the interface is mapped down onto an underlying transport protocol. Generally, the service descriptor, its implementation, and consumption should remain agnostic to what transport is being used, whether that's REST, websockets, or some other transport.

```
import com.lightbend.lagom.javadsl.api.*;

import static com.lightbend.lagom.javadsl.api.Service.*;
public interface HelloService extends Service {
    ServiceCall<String, String> sayHello();
    ServiceCall<NotUsed, PSequence<Item>> getItems(long
    orderId, int pageNo,
    int pageSize);
    default Descriptor descriptor() {
        return named("hello").withCalls(
                call(this::sayHello),
                pathCall("/order/:orderId/
                 items?pageNo&pageSize",
                this::getItems)
        );
    }
}
```

When you use `call`, `namedCall`, or `pathCall`, Lagom will make a best effort attempt to map it down to REST in a semantic fashion, so that means if there is a request message, it will use POST, if there's none, it will use GET. Every service call in Lagom has a request message type and a response message type. When the request or response message isn't used, `akka.NotUsed` can be used in their place. Request and response message types fall into two categories: strict and streamed. A strict message is a single message that can be represented by a simple Java object. The message will be buffered into memory, and then parsed, for example, as JSON. The above service calls use strict messages.

A streamed message is a message of the type `Source`. `Source` is an Akka streams API that allows asynchronous streaming and handling of messages.

```
ServiceCall<String, Source<String, ?>> tick(int
interval);
default Descriptor descriptor() {
    return named("clock").withCalls(
        pathCall("/tick/:interval", this::tick)
    );
}
```

This service call has a strict request type and a streamed response type. An implementation of this might return a Source that sends the input tick message String at the specified interval.

Services are implemented by providing an implementation of the service descriptor interface, implementing each call specified by that descriptor.

```
import com.lightbend.lagom.javadsl.api.*;
import akka.NotUsed;
import static java.util.concurrent.CompletableFuture.
completedFuture;

public class HelloServiceImpl implements HelloService {

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

The `sayHello()` method is implemented using a lambda. An important thing to realize here is that the invocation of `sayHello()` itself does not execute the call, it only returns the call to be executed. The advantage here is that when it comes to composing the call with other cross cutting concerns, such as authentication, this can easily be done using ordinary function-based composition.

Having provided an implementation of the service, we can now register it with the Lagom framework. Lagom is built on top of the Play Framework, and so uses Play's Guice-based dependency injection support to register components. To register a service, you'll need to implement a Guice module. This is done by creating a class called Module in the root package.

```
import com.google.inject.AbstractModule;
import com.lightbend.lagom.javadsl.server.
ServiceGuiceSupport;

public class Module extends AbstractModule implements
ServiceGuiceSupport {

    protected void configure() {
        bindService(HelloService.class, HelloServiceImpl.
        class);
    }
}
```

Working with streamed messages requires the use of Akka streams. The tick service call is going to return a Source that sends messages at the specified interval. Akka streams has a helpful constructor for such a stream:

```
public ServerServiceCall<String, Source<String, ?>>
tick(int intervalMs) {
  return tickMessage -> {
    FiniteDuration interval = FiniteDuration.
    create(intervalMs, TimeUnit.MILLISECONDS);
    return completedFuture(Source.tick(interval,
    interval, tickMessage));
  };
}
```

The first two arguments are the delay before messages should be sent, and the interval at which they should be sent. The third argument is the message that should be sent on each tick. Calling this service call with an interval of 1000 and a request message of tick will result in a stream being returned that sent a `tick` message every second.

If you want to read from the Request header or add something to the Response header, you can use `ServerServiceCall`. If you're implementing the service call directly, you can simply change the return type to be `HeaderServiceCall`.

```
public HeaderServiceCall<String, String> sayHello() {
  return (requestHeader, name) -> {
    String user = requestHeader.principal()
        .map(Principal::getName).orElse("No one");
    String response = user + " wants to say hello to " +
    name;
    ResponseHeader responseHeader = ResponseHeader.OK
        .withHeader("Server", "Hello service");
    return completedFuture(Pair.create(responseHeader,
    response));
  };
}
```

Publishing messages to a broker, such as Apache Kafka, decouples communication even further. Lagom's Message Broker API provides at-least-once semantics and uses Kafka. If a new instance starts publishing information, its messages are added to events previously emitted. If a new instance subscribes to a topic, they will receive all events, past, present, and future. Topics are strongly typed; hence both the subscriber and producer can know in advance what the expected data flowing through will be.

To publish data to a topic a service needs to declare the topic in its service descriptor.

```
import com.lightbend.lagom.javadsl.api.*;
import com.lightbend.lagom.javadsl.api.broker.Topic;

import static com.lightbend.lagom.javadsl.api.Service.*;

public interface HelloService extends Service {
  String GREETINGS_TOPIC = "greetings";
  @Override
  default Descriptor descriptor() {
    return named("helloservice").withCalls(
        pathCall("/api/hello/:id",  this::hello),
        pathCall("/api/hello/:id", this::useGreeting)
      )
      // here we declare the topic(s) this service will
       publish to
      .publishing(
        topic(GREETINGS_TOPIC, this::greetingsTopic)
      )
      .withAutoAcl(true);
  }
  // The topic handle
  Topic<GreetingMessage> greetingsTopic();

  ServiceCall<NotUsed, String> hello(String id);
  ServiceCall<GreetingMessage, Done> useGreeting(String
  id);
}
```

The syntax for declaring a topic is like the one used already to define services' endpoints. The `Descriptor.publishing` method accepts a sequence of topic calls; each topic call can be defined via the `Service.topic` static method. The latter takes a topic name, and a reference to a method that returns a Topic instance. Data flowing through a topic is serialized to JSON by default. It is possible to use a different serialization format by passing a different message serializer for each topic defined in a service descriptor.

The primary source of messages that Lagom is designed to produce is persistent entity events. Rather than publishing events in an ad-hoc fashion in response to things that happen, it is better to take the stream of events from your persistent entities and adapt that to a stream of messages sent to the message broker. This way, you can ensure that events are processed at least once by both publishers and consumers, which allows you to guarantee a very strong level of consistency throughout your system.

Lagom's `TopicProducer` helper provides two methods for publishing a persistent entity's event stream, `singleStreamWithOffset` for use with non-sharded read-side event streams, and `taggedStreamWithOffset` for use with sharded read-side event streams. Both methods take a callback which takes the last offset that the topic producer published, and allows resumption of the event stream from that offset via the `PersistentEntityRegistry.eventStream` method for obtaining a read-side stream.

Here's an example of publishing a single, non-sharded event stream:

```
public Topic<GreetingMessage> greetingsTopic() {
    return TopicProducer.singleStreamWithOffset(offset ->
{
        return persistentEntityRegistry
                .eventStream(HelloEventTag.INSTANCE,
                offset)
                .map(this::convertEvent);
    });
}
```

To subscribe to a topic, a service just needs to call `Topic.subscribe()` on the topic of interest. For instance, if a service wants to collect all greeting messages published by the earlier `HelloService` all you should do is to @Inject the `HelloService` and subscribe to the greetings topic.

```
helloService.greetingsTopic()
    .subscribe() // <-- you get back a Subscriber
    instance
    .atLeastOnce(Flow.fromFunction((GreetingMessage
    message) -> {
        return doSomethingWithTheMessage(message);
    }));
```

When calling `Topic.subscribe()`, you will get back a Subscriber instance. In the above code snippet, we have subscribed to the greetings topic using at-least-once delivery semantics. That means each message published to the greetings topic is received at least once. The subscriber also offers an `atMostOnceSource` that gives you at-most-once delivery semantics. If in doubt, default to using at-least-once delivery semantics.

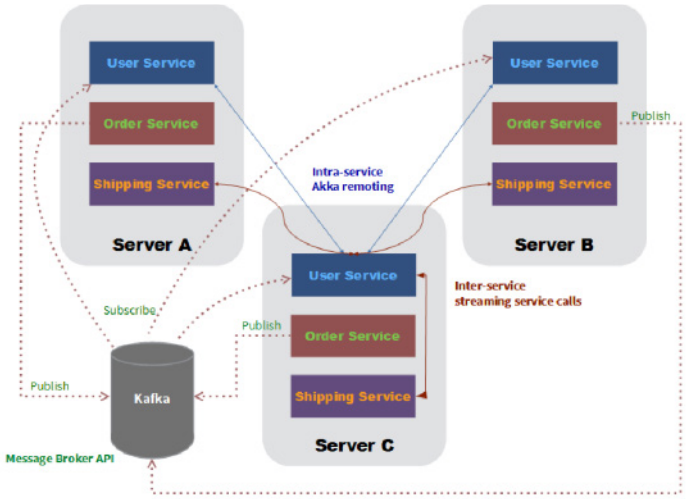
Finally, subscribers are grouped together via `Subscriber.withGroupId`. A subscriber group allows many nodes in your cluster to consume a message stream while ensuring that each message is only handled once by each node in your cluster. Without subscriber groups, all your nodes for a service would get every message in the stream, leading to their processing being duplicated. By default, Lagom will use a group id that has the same name as the service consuming the topic.

## INTRA-SERVICE COMMUNICATION

Nodes of a single service (collectively called a cluster) require less decoupling. For this reason, **intra-service communication** can take advantage of mechanisms that have less overhead and better performance.

- Akka remoting
- Distributed publish-subscribe
- Event streaming

The diagram below illustrates each of these types of inter- and intra-service communication in a Lagom system distributed across three servers. In this example, the Order service publishes to one or more Kafka topics, while the User service subscribes to consume the information. The User service communicates with other User service instances (cluster members) using Akka remoting. The Shipping service and User service exchange information by streaming it in service calls.




## PERSISTENCE, CQRS, AND EVENT SOURCING

Each microservice should own its data. There must be no sharing of databases across different services, since that would result in a too-tight coupling between the services, ultimately making the database the bottleneck and coupling point of your application. In this way, each microservice operates within a clear boundary. To achieve this in Lagom, the persistence module promotes the use of Event Sourcing (ES) and Command Query Responsibility Segregation (CQRS). Event sourcing is the practice of capturing all changes as

domain events, which are immutable facts of things that have happened. Event Sourcing is used for an Aggregate Root, such as a customer with a given customer identifier. Lagom introduces the `PersistentEntity` as an API to interact with ES. The persistent entity is also the transaction boundary. Invariants can be maintained within one entity but not across several entities. Lagom persists the event stream in the database. Event stream processors, other services, or clients read and — optionally, act on — stored events. Lagom supports persistent read-side processors and message broker topic subscribers. To recreate the current state of an entity when it is started, the events are replayed.

If you are familiar with JPA it is worth noting that a `PersistentEntity` can be used for similar things as a JPA `@Entity`, but several aspects are rather different. For example, a JPA `@Entity` is loaded from the database from wherever it is needed, i.e. there may be many Java object instances with the same entity identifier. In contrast, there is only one instance of `PersistentEntity` with a given identifier. With JPA you typically only store current state and the history of how the state was reached is not captured. You interact with a `PersistentEntity` by sending command messages to it. The entities are automatically distributed across the nodes in the cluster of the service. Each entity runs only at one place, and messages can be sent to the entity without requiring the sender to know the location of the entity. An entity is kept alive, holding its current state in memory, as long as it is used. When it has not been used for a while, it will automatically be passivated to free up resources. When an entity is started, it replays the stored events to restore the current state. This can be either the full history of changes or started from a snapshot, which will reduce recovery times.

Lagom supports the following databases:

- Cassandra
- PostgreSQL
- MySQL
- Oracle
- H2

Cassandra is fully supported and integrated into the development environment, and there is no need to install, manage, or configure it. For instructions on configuring your project to use Cassandra, see Using Cassandra for Persistent Entities. If you want to use one of the relational databases listed above, see Using a Relational Database for Persistent Entities on how to configure your project. A simple stub of a `PersistentEntity` looks like:

```
import com.lightbend.lagom.javadsl.persistence.
PersistentEntity;
public class Post
  extends PersistentEntity<BlogCommand, BlogEvent,
  BlogState> {
  @Override
  public Behavior initialBehavior(Optional<BlogState>
  snapshotState) {
    BehaviorBuilder b = newBehaviorBuilder(
        snapshotState.orElse(BlogState.EMPTY));
    // TODO define command and event handlers
    return b.build();
  }
}
```

The functions that process incoming commands are registered in the Behavior using `setCommandHandler` of the `BehaviorBuilder`. You should define one command handler for each command class that the entity can receive.

```
b.setCommandHandler(AddPost.class, (AddPost cmd,
CommandContext<AddPostDone> ctx) -> {
  final PostAdded postAdded = new PostAdded(entityId(),
  cmd.getContent());
  return ctx.thenPersist(postAdded, (PostAdded evt) ->
    // After persist is done additional side effects can
    be performed
    ctx.reply(new AddPostDone(entityId())));
});
```

A command handler returns a Persist directive that defines what event or events, if any, to persist. Each command must define what type of message to use as reply to the command by implementing the `PersistentEntity.ReplyType` interface.

```
final class AddPost implements BlogCommand,
PersistentEntity.ReplyType<AddPostDone> {
// …
}
```

When an event has been persisted successfully, the current state is updated by applying the event to the current state. The functions for updating the state are registered with the `setEventHandler` method of the `BehaviorBuilder`. You should define one event handler for each event class that the entity can persist. Event handlers are used both when persisting new events and when replaying events.

```
b.setEventHandler(PostAdded.class, evt ->
  new BlogState(Optional.of(evt.getContent()),
false));
```

The event handlers are typically only updating the state, but they may also change the behavior of the entity in the sense that new functions for processing commands and events may be defined. Snapshotting helps to reduce the time needed to recreate the `PersitentEntity` when it is started.

## CREATING YOUR FIRST LAGOM APPLICATION

Everything you need to get started is JDK (Java Development Kit) 8 and Maven (3.3 or higher). Maven downloads dependencies and creates the project structure for you. It can take from a few seconds to a few minutes to complete. After confirming the prerequisites, open a console or command window and follow these steps:

1. Create a new directory for your project.
2. Change into the new directory and enter the following (all on one line):

```
mvn archetype:generate -DarchetypeGroupId=com.
lightbend.lagom -DarchetypeArtifactId=maven-
archetype-lagom-java -DarchetypeVersion=1.3.5
```

3. Maven prompts you for:
   **groupId** — Typically something like com.example.

   **artifactId** — Becomes the top-level folder name, for example, my-first-project.

   **version** — The version for your project, press Enter to accept the default.

   **package** — Defaults to the same value as the groupId
4. Enter Y to accept the values. Maven creates the project.
5. Change into the top-level project folder and run it:

```
mvn lagom:runAll
```

   The `runAll` command takes a bit of time. It starts Hello World microservices and registers them in a service directory. It also starts a Cassandra server and a web server.
6. When you see the message, `Services started, ...`, verify that the services are indeed up and running by invoking the hello service endpoint from any HTTP client, such as a browser: http://localhost:9000/api/hello/World

   The request returns the message `Hello, World!`.

## LAGOM IN PRODUCTION

Lagom doesn't prescribe any particular production environment, but out-of-the-box support is provided for Lightbend Enterprise Suite. Lagom sbt support leverages the sbt-native-packager to produce archives of various types. By default, zip archives can be produced, but you can also produce tar.gz, MSI, Debian, RPM, Docker and more. If using Maven, there are many plugins for Maven to produce artifacts for various platforms.

Running a package requires the provision of a service locator implementation i.e. something that provides the ability for your service to be able to lookup the location of another dynamically at runtime. At a technical level, you provide an implementation of a `ServiceLocator`. Learn more in the official Lagom documentation.

## FURTHER LEARNING

This Refcard barely scratched the surface of what is possible with Lagom and how to start building your own microservices based systems. Make sure to dive deeper into the official Lagom Documentation and look at the available examples on GitHub.

## ABOUT THE AUTHOR

**MARKUS EISELE** is a Java Champion, former Java EE Expert Group member, founder of JavaLand, reputed speaker at Java conferences around the world, and a very well-known figure in the Enterprise Java world. He works for Lightbend.

You've known and seen him at different conferences and Java User Groups meetups or read his blogs or are following his social media presence. While talking about middleware for many years you'll continue to hear him talk about enterprise grade Java going forward. Focussed on education about the latest trends in building enterprise systems in a reactive way with Java.

You can find him on Twitter @myfear.



DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com