O'REILLY®

Compliments of Lightbend

Developing Reactive Microservices

Enterprise Implementation in Java

Markus Eisele

Developing Reactive Microservices

Enterprise Implementation in Java

Markus Eisele

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Developing Reactive Microservices
by Markus Eisele

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Foster
Production Editor: Shiny Kalapurakkel
Copyeditor: Christina Edwards
Proofreader: Charles Roumeliotis
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

May 2016: First Edition

Revision History for the First Edition
2016-05-09: First Release

978-1-491-96236-7

[LSI]

Table of Contents

Foreword

"Everyone" is talking about microservices. It is reaching the peak of inflated expectations, and—as with all hyped technologies—it is easy to dismiss as just one of our industry's latest fads, one that will die out quicker than it emerged and be soon forgotten. You can already hear some old-timers say that it is not bringing anything new to the table, that it is just SOA (or god forbid, CORBA) all over again, just common sense rebranded—"been there, done that, move on." These individuals are right—and they are wrong.

Right because the goals of microservices are the same ones that we have pursued in software engineering for decades: isolation, decoupling, composition, integration, maintainability, extensibility, time-to-market, resilience, and scalability.

And wrong because the world of the software engineer is vastly different now than just 10 to 15 years ago. Now we are faced with challenges that are new and scary to most developers, but we have also been given the means to address them.

Today, multicore processors, cloud computing, and mobile devices are the norm, which means that all new systems are distributed systems from day one—a completely different and more challenging world to operate in, a world with lots of new and interesting opportunities. This shift has forced our industry to rethink some of its old "best" practices around system architecture and design.

One example of this is the recent interest in reactive systems and architecture. These systems are defined by the core traits of Responsiveness, Resilience, and Elasticity powered by Asynchronous Message Passing.

Another change is the departure from monolithic architectures toward systems that are decomposed into manageable, discrete, and autonomous services, and scaled individually; these systems can fail, be rolled out, and upgraded in isolation, a design we now call microservices-based.

Traditional architectures, tools, and products simply won't cut it anymore. We can't make the proverbial horse faster—we need cars for where we are going. Layering a new shiny tool like microservices on top of our existing software stack and platform, which was made for monoliths, and then expecting us to not have to change the way we think or learn anything new will only set us up for failure.

The good news is that today we have lots of great tools, frameworks, platforms, and architectural patterns that can help us do the right thing, make the right decisions, and set us up for success.

In this practical report, written for the curious Java developer, Markus shows you how you can take the hard-won knowledge of reactive systems (standing on the shoulders of giants like Jim Gray, Pat Helland, Joe Armstrong, and Robert Virding) and make it a solid foundation for your next microservices-based system.

Along the way you will learn how to create autonomous services that can be rolled out and upgraded in isolation, replicated, and migrated at runtime; self-healed; persisted in a scalable and resilient way; integrated with external and legacy systems; communicated with other services over efficient and evolvable asynchronous protocols; and scaled elastically on demand. And you will learn all this through the lens of Lagom, the reactive microservices framework built on Akka and the Play Framework—a most efficient, pragmatic, and fun way of slaying the monolith.

I hope you enjoy the ride. I know I did.

—Jonas Bonér, CTO at Lightbend

CHAPTER 1

Introduction

If I had asked people what they wanted, they would have said faster horses.

—Henry Ford (July 30, 1863 - April 7, 1947), founder of the Ford Motor Company

With microservices taking the software industry by storm, traditional enterprises are forced to rethink what they've been doing for almost two decades. It's not the first time technology has shocked the well-oiled machine of software architecture to its core. We've seen design paradigms change over time and project management methodologies evolve. Old hands might see this as another wave that will gently find its way to the shore of daily business. But this time it looks like the influence is far bigger than anything we've seen before. And the interesting part is that microservices aren't new.

Talking about compartmentalization and introducing modules belongs to the core skills of architects. Our industry has learned how to couple services and build them around organizational capabilities. The really new part in microservices-based architectures is the way truly independent services are distributed and connected back together. Building an individual service is easy. Building a system out of many is the real challenge because it introduces us to the problem space of distributed systems. This is a major difference from classical, centralized infrastructures. As a result, there are very few concepts from the old world that still fit into a modern architecture.

Today's Challenges for Enterprises

In the past, enterprise developers had to think in terms of specifications and build their implementations inside application server containers without caring too much about their individual life cycle. Creating standardized components for every application layer (e.g., UI, Business, Data, and Integration) while accessing components across them was mostly just an injected instance away.

Connecting to other systems via messaging, connectors, or web services in a point-to-point fashion and exposing system logic to centralized infrastructures was considered best practice. It was just too easy to quickly build out a fully functional and transactional system without having to think about the hard parts like scaling and distributing those applications. Whatever we built with a classic Java EE or Spring platform was a "majestic monolith" at best.

While there was nothing wrong with most of them technically, those applications can't scale beyond the limits of what the base platform allows for in terms of clustering or even distributed caching. And this is no longer a reasonable choice for many of today's business requirements.

With the growing demand for real- and near-time data originating from mobile and other Internet-connected devices, the amount of requests hitting today's middleware infrastructures goes beyond what's manageable for operations and affordable for management. In short, digital business is disrupting traditional business models and driving application leaders to quickly modernize their application architecture and infrastructure strategies. The logical step now is to switch thinking from collaboration between objects in one system to a collaboration of individually scaling systems. There is no other way to scale with the growing demands of modern enterprise systems.

Why Java EE Is Not an Option

Traditional application servers offer a lot of features, but they don't provide what a distributed system needs. Using standard platform APIs and application servers can only be a viable approach if you scale both an application server and database for each deployed service and invest heavily to use asynchronous communication as much as possible. And this approach would still put you back into the 1990s with CORBA, J2EE, and distributed objects. What's more, those runtimes are resource intensive and don't start up or restart fast enough to compensate for failing instances. If that's not enough, you're still going to miss many parts of the so-called "outer architecture" like service discovery, orchestration, configuration, and monitoring.

Aren't Microservices Just SOA?

Many might think that service-oriented architecture (SOA) dressed up in new clothes is the perfect acronym for microservices. However, the answer to this question is twofold: yes, because the thoughts behind isolation, composition, integration, and discrete and autonomous services are the same; and no, because the fundamental ideas of SOA were often misunderstood and misused, resulting in complicated systems where an enterprise service bus (ESB) was used to hook up multiple monoliths communicating over complicated, inefficient, and inflexible protocols. This means the requirement for SOA is stronger than ever. And this might have been the biggest problem for SOA-based applications. They simply tried to apply a new technology stack without redesigning and re-architecturing the existing application portfolio.

DevOps and Methodologies

> Model culture after open source organizations: meritocracy, shared consciousness, transparency, network, platforms.
>
> —Christian Posta, *Red Hat*

Let me issue a warning here: this book focuses on the implementation parts, rather than the organizational aspects, of reactive microservices. But you won't succeed with a microservices architecture if you forget about them. While you can read a lot about how early adopters like Netflix structured their teams for speed instead of efficiency, the needs for enterprise-grade software are different. Teams are usually bigger and the software to be produced is more complex and involves a lot more legacy code. Nevertheless, there are good approaches to structuring enterprise-size development teams around business capabilities and in small units while retaining the relevant steering mechanisms.

The many organizational aspects are summarized in a great presentation by Fred George. The most important individual principle from this presentation is: “When you build it, you own it.” From development to testing to production.

The Pyramid of Modern Enterprise Java Development

There are other surrounding innovations that are creating new opportunities and platform approaches for traditional enterprises. Our industry is learning how everything fits into the bigger picture of distributed systems by embracing all the individual parts and architecting the modern enterprise.

The pyramid in Figure 1-1 was introduced in my first book and breaking it down into individual parts and technologies from an implementation perspective is a natural next step.

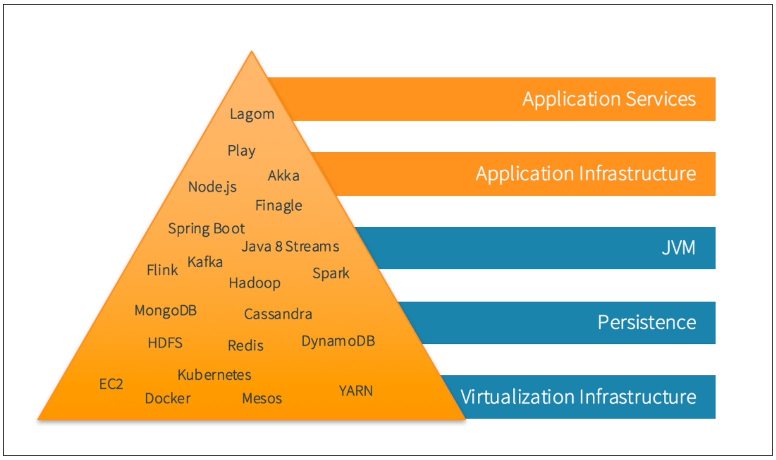


Figure 1-1. Pyramid of modern enterprise Java development refined

Virtualization Infrastructure

Virtualization and infrastructures have been major trends in software development, from specialized appliances and software as a service (SaaS) offerings to virtualized datacenters. In fact, most of the applications we use—and cloud computing as we know it today—would not have been possible without the server utilization and cost savings that resulted from virtualization.

But now new cloud architectures are reimagining the entire data center. Virtualization as we know it is reaching the limits of what is possible for scaling and orchestration of individual applications. With today's applications looking to exploit smaller runtimes and individual services, the need for a complete virtualized operating system (OS) is decreasing. What originated with the Internet giants like Google and Facebook quickly caught the attention of major enterprise customers who are now looking to adopt containers and orchestration. And the trust put forward into cloud-based solutions using those technologies is only increasing.

Both virtual machines and containers are means of isolating applications from hardware. However, unlike virtual machines—which virtualize the underlying hardware and contain an OS along with the application stack—containers virtualize only the OS and contain only the application. As a result, containers have a very small footprint and can be launched in mere seconds. A physical machine can accommodate four to eight times more containers than VMs. With the transformation of data centers, and the switch over to more lightweight container operation systems, our industry is fully adopting public, private, and hybrid cloud infrastructures.

Persistence

The traditional method of data access in monoliths (e.g., Java database access—JDBC) doesn't scale well enough in highly distributed applications because JDBC operations block the socket input/output (IO) and, further on, blocks the thread they run on. The key concept of immutability plays a very important role in microservices. They are thread-safe and you don't run into synchronization issues. And while they can't be changed, there is no problem parallelizing work without conflicting access. This is extremely helpful in representing commands, messages, and states. It also encourages developers to implement distributed systems with an event-sourced architecture.

Event sourcing (ES) and command query responsibility segregation (CQRS) are frequently mentioned together. Both of them are explained in Chapter 3. Although neither one necessarily implies the other, they do complement each other. The main conceptual difference for ES architectures is that changes are captured as immutable facts of things that have happened. For example: "the flight was booked by Markus." All events are stored and the current state can be derived from the events. The advantages are plenty:

- There is no need for O/R mapping. Events are logged and part of the domain model.
- With every change being captured as an event, the current state can easily be replayed and audited because both operate on the same data.
- Persistence works without updates or deletes, the most expensive data manipulation operations.

With these approaches in mind, there is no longer a need for traditional relational database management systems (RDBMS). The persistence of modern applications allows for the embrace of NoSQL-based data stores. A NoSQL (originally referring to "non-SQL" or "nonrelational") database provides a mechanism for storage and retrieval of data that is modeled differently than the tabular relations used in relational databases.

Combining these technologies with the JVM allows developers to build a "fast data" architecture. The emphasis on immutability improves robustness, and data pipelines are naturally modeled and implemented using collections (like lists and maps) with composable operations. The phrase "fast data" captures the range of new systems and approaches, which balance various tradeoffs to deliver timely, cost-efficient data processing, as well as higher developer productivity.

Java Virtual Machine (JVM)

Handling streams of data, especially "live" data whose volume is not predetermined, requires special care in an asynchronous system. The most prominent issue is that resource consumption needs to be controlled so that a fast data source does not overwhelm the stream destination. Asynchronicity is the better way to enable the parallel use of computing resources on collaborating network hosts or multiple CPU cores within a single machine.

The second key to handling this data is about being able to scale your application to deal with a lot of concurrency. This can best be achieved with a nonblocking, streams-based programming approach. It can't be done effectively with thread pools and blocking (OS threads) implementations. The components and implementations that meet these requirements must follow the Reactive Manifesto, which means they are:

- Scalable up and down with demand
- Resilient against failures that are inevitable in large distributed systems
- Responsive to service requests even if failures limit the ability to deliver services
- Driven by messages or events from the world around them

Aims and Scope

This report walks you through the creation of a sample reactive microservices-based system. The example is based on Lagom, a new framework that helps Java developers to easily follow the described requirements for building distributed, reactive systems. As an Apache-licensed, open source project, it is freely available for download, and you can try out the example yourself or play with others provided in the project's GitHub repository.

Going forward, Chapter 2 provides an overview of the reactive programming model and basic requirements for developing reactive microservices. Chapter 3 looks at creating base services, exposing endpoints, and then connecting them with a simple, Web-based user interface. Chapter 4 deals with the application's persistence, while Chapter 5 focuses on first pointers for using integration technologies to start a successful migration away from legacy systems.

CHAPTER 2

Reactive Microservices and Basic Principles

Traditional application architectures and platforms are obsolete.

—Anne Thomas, *Gartner's Modernizing Application Architecture and Infrastructure Primer for 2016*

With classical application platforms no longer an option for today's business requirements, Java developers have little to no choice but to switch to new technologies and paradigms. However, very few of these technologies are built to make an easy transition from traditional Java and Java EE development paradigms. The need to write applications that must be uncompromisingly robust, reliable, available, scalable, secure, and self-healing resulted in the creation of techniques and tools to address these new requirements, such as Akka, Vert.x, and Netty.

The key to implementing reactive microservices in Java-based systems is to find a very opinionated way to build, run, and scale applications on the JVM. Microservices need to be asynchronous by default. Relying on synchronous REST creates coupling that affects productivity, scalability, and availability. These are requirements only a new microservices framework can solve based on the principles of asynchronous message passing, event-based persistence through event sourcing, and CQRS out-of-the-box. All of these requirements are hard to compare with classic middleware platforms as they embrace a new way of programming based on the Reactive Manifesto.

Reactive principles are not new. They have been proven and hardened for more than 40 years, going back to the seminal work by Carl Hewitt and his invention of the Actor Model, Jim Gray and Pat Helland at Tandem Systems, and Joe Armstrong and Robert Virding and their work on Erlang. These people were ahead of their time, but now the world has caught up with their innovative thinking and we depend on their discoveries and work more than ever.

Microservices in a Reactive World

Up until now, the usual way to describe reactive applications has been to use a mix of technical and industry buzzwords, such as asynchronous, nonblocking, real-time, highly available, loosely coupled, scalable, fault-tolerant, concurrent, reactive, message-driven, push instead of pull, distributed, low latency, and high throughput.

The Reactive Manifesto brings all of these characteristics together and defines them through four high-level traits: responsive, resilient, elastic, and message-driven (Figure 2-1). Even if it looks like this describes an entirely new architectural pattern, the core principles have long been known in industries that require real-time IT systems, such as financial trading. If you think about systems composed out of individual services, you will realize how closely the reactive world is related to microservices. Let's explore the four main characteristics of a reactive microservices system closer.

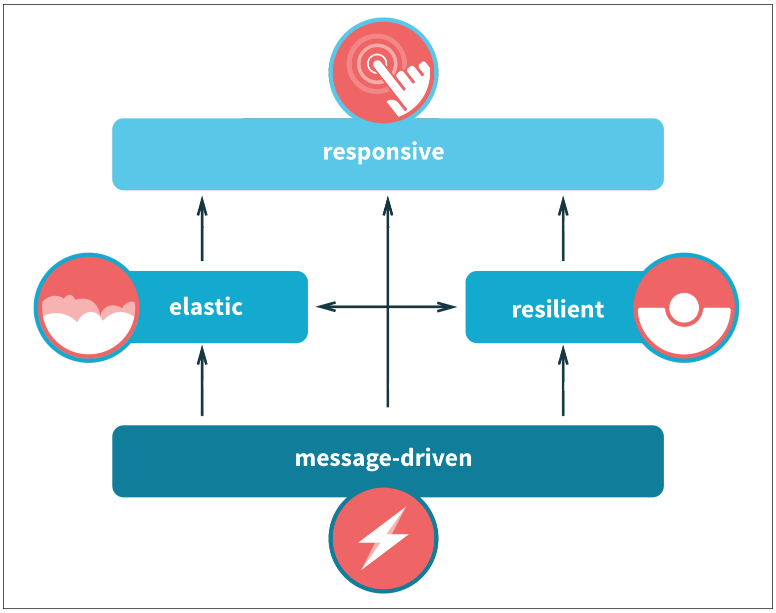


Figure 2-1. By means of an asynchronous, nonblocking, message-driven approach, highly resilient and elastic systems can be formed, resulting in a consistently responsive user experience

Responsive

The first and foremost quality of a service is that it must respond to requests it receives. The same holds true in those cases where services consume other services. They also expect responses in order to be able to continue to perform their work. A responsive application satisfies consumers' expectations in terms of availability and real-time responses. Responsiveness is measured in *latency*, which is the time between request and response.

Resilient

A very high-quality service performs its function without any downtime at all. But failures do happen and handling failure of an individual service in a gentle way without affecting the complete system is what the term *resiliency* describes. As a matter of fact, there is only one generic way to protect your system from failing as a whole when a part fails: distribute and compartmentalize.

Elastic

A successful service will need to be scalable both up and down in response to changes in the rate at which the service is used. With sudden traffic bursts hitting an application, it must be able to make use of increased hardware capacity when needed. This includes not only working on one machine but also how to facilitate the power of several physical nodes in a network spanning various locations transparently.

Message-Driven

The only way to fulfill all of the above requirements is to have loosely coupled services with explicit protocols communicating over messages. Also known as "share nothing" architecture, this removes scalability limits imposed by Amdahl's law. Components can remain inactive until a message arrives, freeing up resources while doing nothing. In order to realize this, nonblocking and asynchronous APIs must be provided that explicitly expose the system's underlying message structure.

The Reactive Programming Model for Java Developers

But how do developers implement all of the above requirements and not use blocking and synchronous APIs? Reactive programming is the answer. At the very core this is a programming paradigm based on asynchronous message passing. This is used to establish a boundary between components that ensures loose coupling, isolation, and location transparency, and provides the means to delegate errors as messages.

You can create data streams of anything. Streams are cheap in terms of memory consumption and quite ubiquitous. You can use anything as a stream: variables, user inputs, properties, caches, and data structures. And if you're thinking about StAX (streaming API for XML) right about now, I don't blame you. Working on a stream of data and generating output while the final element hasn't been received is exactly that kind of thinking—reacting to messages immediately when they arrive.

But reactive programming is closer to functional programming in general. A stream can be an input to another one. Even multiple streams can be used as inputs to another stream. You can *merge* two streams. You can *filter* a stream to get another one that has only those messages you are interested in. You can map data values from one stream to a new one. Java 8 introduced the Streams API, but this is a functional view over collections. Java 9 is slated to include the Reactive Streams specification.

Let's look at a simple example. Let's assume you have a bunch of receipts in your system and you need to find the ones you received for refueling your car ordered by receipt amount. Here's the complete example before Java 8:

```
List<Receipt> receipts = new Arraylist<>();
for(Receipt r: receipts){
  if(r.getType() == Receipt.FUEL){
    fuelTransactions.add(r);
  }
}
Collections.sort(fuelTransactions, new Comparator(){
  public int compare(Receipt r1, Receipt r2){
    return r2.getAmount().compareTo(r1.getAmount());
  }
});
List<Integer> byFuelAmount = new ArrayList<>();
for(Receipt r: fuelTransactions){
  byFuelAmount.add(r.getId());
}
```

That's a lot of code. And it is executed one step after another. When you work with the Streams API of Java 8 the same code looks like this:

```
List<Integer> byFuelAmount =
        receipts.parallelStream()
                .filter(r -> r.getType() == Receipt.FUEL)
                .sorted(comparing(Transaction::getAmount).
                 reversed())
                .map(Receipt::getId)
                .collect(toList());
```

Reactive programming raises the level of abstraction in your code so you can focus on the interdependence of messages that define the business logic, rather than having to handle a large amount of implementation details. And working with streams is only one part of it. In order to be fast and lightweight, the number of updates to existing data structures should be limited or eliminated. This is where immutable (or unmodifiable) data structures come in.

Basic Microservices Requirements

When designing individual reactive microservices, it is important to adhere to the core traits of:

- Isolation
- Autonomy
- Single responsibility
- Exclusive state
- Asynchronous message passing
- Mobility

Microservices are collaborative in nature and only make sense as systems. A complete explanation of all of these traits can be found in the *Reactive Microservices Architecture* report.

CHAPTER 3

Implementing Reactive Microservices in Java

Getting just what you need is more valuable than getting lots of something.

—Martin Elwin, *on the meaning of the Swedish word "Lagom" at Monikgras 2015*

When people talk about microservices, they focus on the "micro" part, saying that a service should be small. I want to emphasize that the important thing to consider when splitting a system into services is to find the right boundaries between services, aligning them with bounded contexts, business capabilities, and isolation requirements. As a result, a microservices-based system can achieve its scalability and resilience requirements, making it easy to deploy and manage.

This report uses the new Lagom framework to build a reactive microservices system. Lagom (pronounced [ˈlɑ̀ːɡɔm]) is a Swedish word meaning "just the right amount." This clearly expresses the focus of the framework on having services that are just the right size. It offers four main features:

- A Service API that provides a way to declare and implement service interfaces, to be consumed by clients. For location transparency, clients use stable addresses and discover services through a service locator. The Service API supports synchronous request-response calls as well as asynchronous messaging and streaming between services.

- The *Persistence API* provides event-sourced persistent entities for services that store data, with CQRS read-side support for queries. It manages the distribution of persisted entities across a cluster of nodes, enabling sharding and horizontal scaling, with Cassandra as the default database backend.
- The *Development Environment* allows you to run all your services, and the supporting Lagom infrastructure, with one command. It hot-reloads your services when code changes.
- *Lagom Services* can be deployed as they are directly to production using ConductR, a platform for monitoring—and scaling—of Lagom services in a container environment—no additional infrastructure needed.

Lagom also relies on a couple of well-known technologies that are wired together to raise productivity during the development of a distributed microservices system.

Java and Scala

The first version of Lagom provides a Java API for writing microservices. Before long, a subsequent version will add a Scala API. Lagom's Java APIs target Java 8. They assume familiarity with Java 8 features such as lambdas, default methods, and optional.

sbt Build Tool

One place where some light Scala coding is required, even for Java users, is in Lagom build definitions, which must be written using sbt's Scala DSL. Lagom's sbt-based development console allows you to run any number of services together with a single command.

Cassandra

Lagom has support for Cassandra as a data store, both for the write-side and read-side. Cassandra is a very scalable distributed database, and it is also flexible enough to support typical use cases of reactive services. By default, Lagom services needing to persist data use Cassandra as a database. For convenience, the development environment embeds a Cassandra server.

Play Framework

Lagom is implemented on top of the Play Framework. This is an implementation detail that will not directly concern simple micro-services. More advanced users may wish to use some Play APIs directly. Play is built on the popular standard Netty network transport library. If you have an existing Play Framework application that you want to add microservices to there is an sbt plugin (PlayLagom) to help you.

Guice

Like Play, Lagom uses Guice for dependency injection.

Akka

Lagom Persistence, Publish-Subscribe, and Cluster are implemented on top of Akka, a toolkit for building concurrent, distributed, and resilient message-driven applications. This is an implementation detail that will not directly concern simple microservices. More advanced users may wish to use some Akka APIs directly.

Akka Streams

A Lagom service may be "simple" or "streamed"; this is described further later in this report in the section "Service Implementation" on page 25. Streaming asynchronous Lagom services are built on top of Akka Streams.

Akka Cluster

If you want to scale your microservices out across multiple servers, Lagom provides clustering via Akka Cluster.

Configuration

Lagom and many of its component technologies are configured using the Lightbend Config library. The configuration file format is HOCON, a powerful and expressive superset of JSON.

Logging

Lagom uses SLF4J for logging, backed by Logback as its default logging engine.

Example Application

The core concepts of reactive microservices have been introduced by Jonas Bonér in his report, *Reactive Microservices Architecture*. He clearly explains the main characteristics of microservices as a set of isolated services, each with a single area of responsibility. This forms the basis for being able to treat each service as a single unit that lives and dies in isolation—a prerequisite for resilience—and can be moved around in isolation—a prerequisite for elasticity. But they also own their state.

Each microservice has to take sole responsibility for its own state and persistence. Modeling each service as a bounded context can be helpful since each service usually defines its own domain, each with its own ubiquitous language. Both of these techniques are taken from domain-driven design (DDD). Of all the new concepts introduced in reactive microservices applications, consider DDD a good place to start learning. Microservices and the terms you hear in context are heavily influenced by DDD.

The philosophy of DDD is about placing the attention at the heart of the application, focusing on the complexity of the core business domain. Alongside the core business features, you'll also find supporting subdomains that are often generic in nature, such as money or time. DDD aims to create models of a problem domain. All the implementation details—like persistence, user interfaces, and messaging—come later.

DDD was first introduced by Eric Evans in his book *Domain-Driven Design: Tackling Complexity in the Heart of Software*. In this book he also discusses an example application, the Cargo tracker. The example in this report follows the main business ideas showcased there and implements a first set of features.

Getting Started with the Example Application

The source code for the example application is available on GitHub. Please keep in mind that it is not a complete implementation of the well-known DDD example but merely a starting point. It will help you understand how to develop a reactive microservices system and is meant to be a playground for getting started.

The following examples assume you are using Mac OS. Find more information on how to install sbt and Java 8 in the Lagom documentation.

The first step is to install sbt version 0.13.11. Make sure to have Homebrew installed:

```
$ brew install sbt
```

After you're done with this you can fork and clone the example GitHub repository to your local machine:

```
$ git clone git@github.com:<username>/activator-lagom-
  cargotracker.git
```

If you want to contribute to the project, make sure to add the remote upstream accordingly. Change into the newly created directory and start sbt:

```
$ sbt
[info] Loading project definition from /cargotracker/project
[info] Set current project to cargotracker
(in build file:/cargotracker/)
```

You've now entered the sbt prompt. As a Java developer, keep in mind that sbt is not really comparable with Maven. sbt is a lot more than just a build tool. It is a complete development environment that offers an efficient, powerful, and convenient development experience. A Lagom system is typically made up of a set of sbt builds, each build providing multiple services. If you inspect the example project's directory structure, you'll see the following:

- API and implementation projects for each service (shipping and registration)

- A *front-end* project that contains the ReactJS-based web-ui for the example
- The project folder with sbt-specific files.
- The *build.sbt* file, which contains all the information necessary to build, run, and deploy your services

Let's take the example for a little test drive before diving in deeper:

```
$ sbt runAll

[info] Starting embedded Cassandra server
..........
[info] Cassandra server running at 127.0.0.1:4000
..........
[info] Service locator is running at http://localhost:8000
[info] Service gateway is running at http://localhost:9000
..........
[info] Service registration-impl listening for
       HTTP on 0:0:0:0:0:0:0:0:20019
[info] Service shipping-impl listening for
       HTTP on 0:0:0:0:0:0:0:0:25003
[info] Service front-end listening for
       HTTP on 0:0:0:0:0:0:0:0:28118
[info] (Services started, use Ctrl+D to stop and go
        back to the console...)
```

When inspecting the list of running services, you may wonder how ports are being assigned. Ports are assigned consistently, meaning that once a port is assigned to a service it will never change again. The port is deterministic even for different machines. An algorithm creates it out of the service and project name and maps it onto the default port range. With that said, you can change the settings easily.

Pointing your browser to the service gateway address will bring up the screen shown in Figure 3-1.

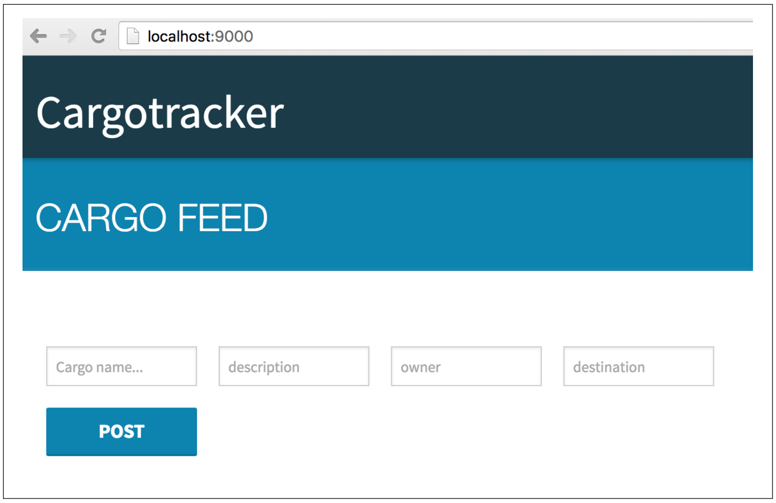


Figure 3-1. Cargo tracker start screen

You can enter random text in each field and click "Post" to get it submitted. If you check the logs you'll see that a cargo with an assigned ID was persisted:

```
[info] s.c.r.i.RegistrationServiceImpl - Cargo ID: 154038.
```

Now it's time to look at what happened under the hood. Open the project in IntelliJ and review the registration service. You can also refer to the Lagom documentation for importing your project into your IDE of choice.

This example makes use of Immutables, which is the recommended way for Lagom applications to handle commands, events, and states. Immutables is a library that creates immutable objects and reduces boilerplate code to a minimum. Make sure to set up the code generation in your IDE for the project correctly.

API Versus Implementation

There are two projects with the prefix *registration*. One is the API project and the other the implementation project. The API project contains a service interface through which consumers may interact with the service, while the implementation project contains the actual service implementation.

The *registration-api* project contains two classes. The `RegistrationService` contains the service description that is defined by an interface. 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, and its implementation and consumption, should remain agnostic to what transport is being used, regardless of whether that's REST, Websocket, or other transports.

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

public interface RegistrationService extends Service {

   ServiceCall<NotUsed, Cargo, Done> register();
   ServiceCall<NotUsed, NotUsed, Source<Cargo, ?>>
                                getLiveRegistrations();

   @Override
   default Descriptor descriptor() {
       return named("registrationService").with(
           restCall(Method.POST, "/api/registration",
                    register()),
           pathCall("/api/registration/live",
                    getLiveRegistrations())
         ).withAutoAcl(true);
   }
}
```

The descriptor defines a service with two calls: `register()` and `getLiveRegistrations()`. Both return a `ServiceCall`, which is a representation of the call that can be invoked when consuming the service. Unlike what most Java developers are used to, an invocation of both methods does not actually invoke the call but gives you a handle to it, which has to be invoked with the `ServiceCall.invoke()` method. A `ServiceCall` takes three type parameters and `id`, which is extracted from the incoming identifier, e.g., the path in the case of a REST request. As the `akka.NotUsed` indicates, this example doesn't have an ID and its REST transport implementation is going to use a static path. `Request` and `Response` reflect incoming and outgoing messages. `Cargo` is the request and `akka.Done` the response.

After implementing and invoking the `register()` call, the mapping to the actual transport is still missing. This is done by providing a `default` implementation of the `Service.descriptor()` method.

The implementation is pretty self-explanatory. A service named `registrationService` is returned that contains two REST calls. The call mapping to the `register()` service call is a POST request to `/api/registration` and the `getLiveRegistrations()` call is a GET request mapped to `/api/registration/live`.

We have to look back to the identifiers a bit. In fact, the mapping to the transport is only one aspect. Even more important is that each service call needs to have an identifier to provide routing information to the client and the service to make sure the calls are mapped appropriately. As explained before, our example doesn't use a dynamic part and thus returns `akka.NotUsed`. But we still need one and this example uses both, a path-based identifier that uses a URI path and query string (`pathCall()`) to route calls and a REST identifier (`restCall()`). REST identifiers should be used when you're creating a semantic REST API. They use both a path, as with the path-based identifier, and a request method to identify them. To learn more about configuration and usage of identifiers in general, refer to the path-based identifier section in the Lagom documentation.

We already learned that reactive microservices systems communicate via messages. Every service call in Lagom has a request message type and a response message type that fall into two categories:

- 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. When both message types are strict, the call is said to be a synchronous call, i.e., a request is sent and received, then a response is sent and received. The caller and callee have synchronized their communication.
- A *streamed message* is a message of type `Source`, which is an Akka Streams API that allows asynchronous streaming and handling of messages. This is what the `getLiveRegistrations()` call returns. A stream of `Source<Cargo>` is returned as the response message. Instead of using synchronous REST calls, Lagom chooses an appropriate transport for the stream, which will typically be WebSockets. WebSockets support bidirectional streaming, and is a good general-purpose option for streaming.

By default all messages are serialized and deserialized to JSON by using Jackson. To change this behavior or to write and configure custom message serializers, see Message Serializers in the Lagom documentation.

The message passed around in this example is a `Cargo` object. And it is also part of the *registration-api* project because it is part of the external API.

```
import
  com.fasterxml.jackson.databind.annotation.JsonDeserialize;
import com.lightbend.lagom.javadsl.immutable.ImmutableStyle;
import org.immutables.value.Value;

@Value.Immutable
@ImmutableStyle
@JsonDeserialize(as = Cargo.class)
public interface AbstractCargo {

    @Value.Parameter
    String getId();

    @Value.Parameter
    String getName();

    //...
}
```

This is a simple value object on first sight. But looking closer, we can see that it is an immutable object, which means it cannot be modified after it is created. All fields are final and are assigned at construction time. There are no setter methods.

Immutable objects have two great advantages:

- Code based on immutable objects is clearer and likelier to be correct. Bugs involving unexpected changes simply can't occur.
- Multiple threads can safely access immutable objects concurrently.

Lagom doesn't care which library you use to create your immutable objects. You can also write your immutable objects yourself or use another library. If you are going to use the Immutables library, make sure to also add the `lagomJavadslImmutables` dependency in your *project/plugins.sbt* file.

The generated `Cargo.java` class can be found in the *target/scala-2.11/src_managed* folder of the *registration-api* project. Deeper coverage about immutables and how to use them is part of the Lagom documentation.

We have now explored the complete *registration-api* project and it is time to move on to the service implementation.

Service Implementation

Services are implemented by providing an implementation of the service descriptor interface and all its calls. Before we dive into the implementation we need to make sure it is registered with Lagom. The service implementation is a module and the registration can be done by extending `AbstractModule` and implementing `ServiceGuiceSupport`:

```
public class RegistrationServiceModule extends
             AbstractModule
implements ServiceGuiceSupport {
    @Override
    protected void configure() {
      bindServices(serviceBinding(RegistrationService.class,
      RegistrationServiceImpl.class));
    }
}
```

Now onto the actual service, `RegistrationServiceImpl`. The first call we tried out as a smoke test was `register()`. And now that you have seen the service descriptor, we can look at the implementation:

```
    private final PersistentEntityRegistry persistentEntityRegistry;
    private final PubSubRegistry topics;
    private final Logger log =
            LoggerFactory.getLogger(RegistrationServiceImpl.class);

    @Override
       public ServiceCall<NotUsed, Cargo, Done> register() {
           return (id, request) -> {
               // Publish received entity into topic named "Topic"
               PubSubRef<Cargo> topic = topics.refFor(
                                 TopicId.of(Cargo.class, "topic"));
               topic.publish(request);

               log.info("Cargo ID: {}.", request.getId());

              //...
           };
       }
```

The service call just returns `akka.Done`. Let's try to find out what actually happens here. The first two lines represent a publish/subscribe topic. The line starting with `PubSubRef<Cargo> topic = ...` creates a topic with the name "topic" and registers it with the `PubSubRegistry`. The next line publishes a single object into the topic. You can find the consuming side of the topic in the second service call, `getLiveRegistrations()`.

A simple log entry via Log4J was the initial smoke test. And now it is time to learn about how to persist data.

CHAPTER 4

Dealing with Persistence, State, and Clients

There is no serious enterprise-grade application without persistent data. And the same is true for microservices-based applications. Event sourcing and CQRS are fundamental concepts behind Lagom's support for services that store information. To understand these concepts, it is recommended that you read Lagom's CQRS and Event Sourcing documentation.

When using event sourcing, all changes are captured as domain events, which are immutable facts of things that have happened. The persistent equivalent is called an *AggregateRoot* (`PersistentEntity`). This is a cluster of domain objects that can be treated as a single unit. An example may be a piece of cargo and its transport legs, which will be separate objects, but it's useful to treat the cargo (together with its transport legs) as a single aggregate. The aggregate can reply to queries for a specific identifier but it cannot be used for serving queries that span more than one aggregate. Therefore, you need to create another view of the data that is tailored to the queries that the service provides (see Figure 4-1). This separation of the write-side and the read-side of the persistent data is often referred to as the CQRS pattern.

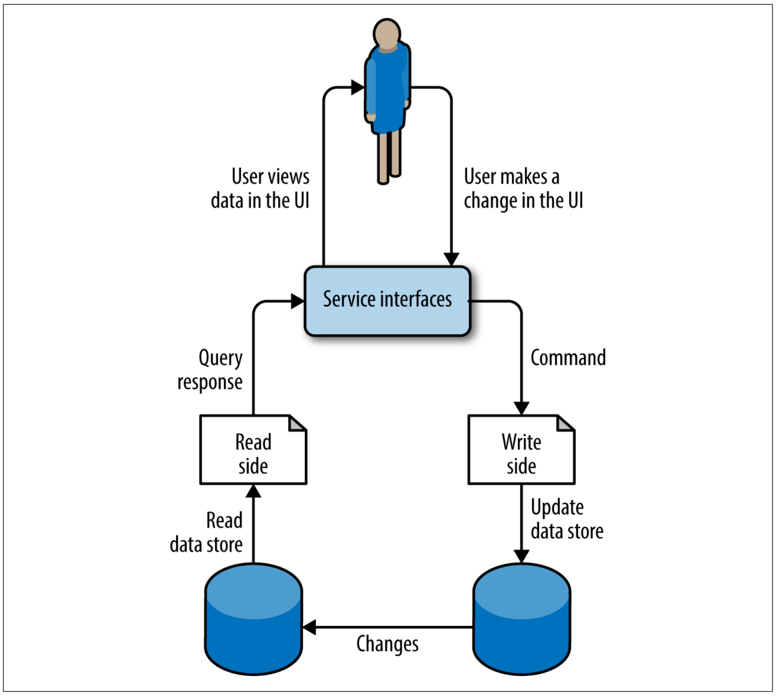


Figure 4-1. Command query responsibility segregation

To implement persistence in Lagom you have to implement a class that extends `PersistentEntity<Command, Event, State>`.

If you are familiar with JPA, you might want to consider a `PersistentEntity` as a mixture between data access objects (DTOs) and a JPA `@Entity`. But the differences are obvious. While a JPA entity is loaded from the database wherever it is needed, 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 the 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. Commands are processed sequentially, one at a time, for a specific entity instance. A command may result in state changes that are persisted as events, representing the effect of the command. The current state is not stored for every change, since it can be derived from the events. These events are only ever appended to storage; nothing is ever mutated, which allows for very high transaction rates and efficient replication.

With all this knowledge, the remaining lines of code in the `RegistrationServiceImpl` just got a little clearer. In Lagom, the way to send commands to a `PersistentEntity` is by using a `PersistentEntityRef`, which needs to be looked up via the `PersistentEntityRegistry`. This means that `CargoEntity` is the `PersistentEntity` and `RegisterCargo` is the command we want to send:

```java
// Look up the CargoEntity for the given ID.
PersistentEntityRef<RegistrationCommand> ref =
        persistentEntityRegistry.refFor(CargoEntity.class,
                                        request.getId());
// Tell the entity to use the Cargo information in the request.
return ref.ask(RegisterCargo.of(request));
```

We made it all the way through the service implementation down to the `CargoEntity`. This is an event-sourced entity. It has a state `CargoState`, which stores information about the registered cargo. It can also receive commands that are defined in the `RegistrationCommand` and translate them into events that are defined in the `RegistrationEvent` class.

As mentioned before, this is the place where commands are translated into events. And as such, the `PersistentEntity` has to define a behavior for every command it understands. A behavior is defined by registering commands and event handlers:

```java
public class CargoEntity
        extends PersistentEntity<RegistrationCommand,
                                 RegistrationEvent,
                                 CargoState> {
    @Override
    public Behavior initialBehavior(
                Optional<CargoState> snapshotState) {

        //  command and event handlers

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

If you look at the supported commands, you will find only one, `RegisterCargo`. It is sent down from the UI when a user adds a new cargo. By convention, the commands should be inner classes of the interface, which makes it simple to get a complete picture of what commands an entity supports. Commands are also immutable objects:

```
public interface RegistrationCommand extends Jsonable {
    @Value.Immutable
    @ImmutableStyle
    @JsonDeserialize(as = RegisterCargo.class)
    public interface AbstractRegisterCargo extends
            RegistrationCommand,
            CompressedJsonable,
            PersistentEntity.ReplyType<Done> {
        @Value.Parameter
        Cargo getCargo();
    }
}
```

Commands get translated to events, and it's the events that get persisted. Each event will have an event handler registered for it, and an event handler simply applies an event to the current state. This will be done when the event is first created, and it will also be done when the entity is loaded from the database—each event will be replayed to re-create the state of the entity. The `RegistrationEvent` interface defines all the events supported by the `CargoEntity`. In our case it is exactly one event: `CargoRegistered`:

```
public interface RegistrationEvent extends
                 Jsonable,
                 AggregateEvent<RegistrationEvent> {

    @Immutable
    @ImmutableStyle
    @JsonDeserialize(as = CargoRegistered.class)
    interface AbstractCargoRegistered extends
             RegistrationEvent {
        @Override
        default public AggregateEventTag<RegistrationEvent>
             aggregateTag() {
               return RegistrationEventTag.INSTANCE;
             }
        @Value.Parameter
        String getId();
        @Value.Parameter
        Cargo getCargo();
    }
}
```

This event is emitted when a `RegisterCargo` command is received. Events and commands are nothing more than immutable objects.

Let's add the different behaviors to handle the command and trigger events.

Behavior is defined using a behavior builder. The behavior builder starts with a state, and if this entity supports snapshotting—with an optimization strategy that allows the state itself to be persisted to combine many events into one—then the passed-in `snapshotState` may have a value that can be used. Otherwise, the default state is to use a dummy `cargo` with an `id` of empty string:

```
@Override
  public Behavior initialBehavior(
                Optional<CargoState> snapshotState) {

  BehaviorBuilder b = newBehaviorBuilder(
              snapshotState.orElse(
                CargoState.builder().cargo(
                        Cargo.builder()
                                .id("")
                                .description("")
                                .destination("")
                                .name("")
                                .owner("").build())
                        .timestamp(LocalDateTime.now()
                        ).build()));
            //...
```

The functions that process incoming commands are registered in the behavior using `setCommandHandler` of the `BehaviorBuilder`. We start with the initial `RegisterCargo` command. The command handler validates the command payload (in this case it only checks if the cargo has a name set) and emits the `CargoRegistered` event with the new payload. A command handler returns a persist directive that defines what event or events, if any, to persist. This example uses the `thenPersist` directive, which only stores a single event:

```
//...
b.setCommandHandler(RegisterCargo.class, (cmd, ctx) -> {
       if (cmd.getCargo().getName() == null ||
        cmd.getCargo().getName().equals("")) {
           ctx.invalidCommand("Name must be defined");
           return ctx.done();
       }

       final CargoRegistered cargoRegistered =
               CargoRegistered.builder().cargo(
               cmd.getCargo()).id(entityId()).build();
             return ctx.thenPersist(cargoRegistered,
             evt -> ctx.reply(Done.getInstance()));
      });
```

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 also registered with the `setEventHandler` method of the `BehaviorBuilder`. The event handler returns the new state. The state must be immutable, so you return a new instance of the state:

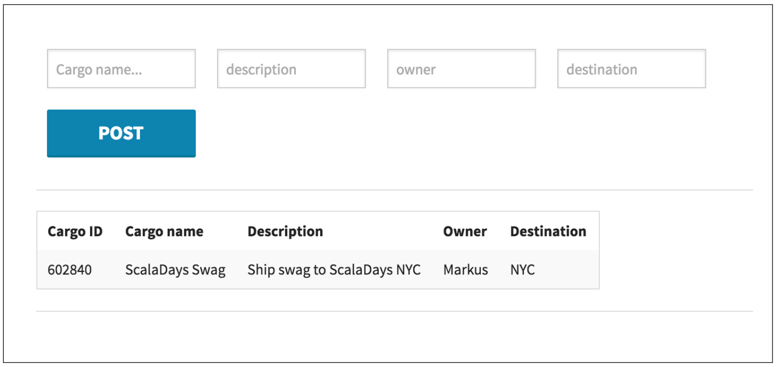
```
b.setEventHandler(CargoRegistered.class,
             // We simply update the current
             // state to use the new cargo payload
             // and update the timestamp
             evt -> state()
                     .withCargo(evt.getCargo())
                     .withTimestamp(LocalDateTime.now())
                     );
```

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. Learn more about this in the `PersistentEntity` documentation.

We successfully persisted an entity. Let's finish the example and see how it is displayed to the user. The `getLiveRegistrations()` service call subscribes to the topic that was created in the `register()` service call before and returns the received content:

```
@Override
   public ServiceCall<NotUsed, NotUsed, Source<Cargo, ?>>
    getLiveRegistrations() {
       return (id, req) -> {
           PubSubRef<Cargo> topic = topics.refFor(
           TopicId.of(Cargo.class, "topic"));
           return CompletableFuture.completedFuture(
           topic.subscriber()
           );
       };
   }
```

To see the consumer side, you have to look into the *front-end* project and open the ReactJS application in *main.jsx*. The `createCargo Stream()` function points to the API endpoint and the live cargo events are published to the `cargoNodes` function and rendered accordingly (see Figure 4-2).



Cargo ID	Cargo name	Description	Owner	Destination
602840	ScalaDays Swag	Ship swag to ScalaDays NYC	Markus	NYC

Figure 4-2. Publishing cargo events to the UI

One last step in this example is to add a REST-based API to expose all the persisted cargo to an external system. While persistent entities are used for holding the state of individual entities—and to work with them you need to know the identifier of an entity—the `readAll (select *)` is a different use case. Another view on the persisted data is tailored to the queries the service provides. Lagom has support for populating this read-side view of the data and also for building queries on the read-side.

We start with the service implementation again. The `CassandraSession` is injected in the constructor of the implementation class. `CassandraSession` provides several methods in different flavors for executing queries. All methods are nonblocking and they return a `CompletionStage` or a `Source`. The statements are expressed in Cassandra query language (CQL) syntax:

```
@Override
   public ServiceCall<NotUsed, NotUsed, PSequence<Cargo>>
   getAllRegistrations() {
       return (userId, req) -> {
           CompletionStage<PSequence<Cargo>>
           result = db.selectAll("SELECT cargoid,"
                         + "name, description, owner,"
                         + "destination FROM cargo")
                   .thenApply(rows -> {
                       List<Cargo> cargos =
                       rows.stream().map(row ->
                       Cargo.of(row.getString("cargoid"),
                               row.getString("name"),
                               row.getString("description"),
                               row.getString("owner"),
                               row.getString("destination")))
                               .collect(Collectors.toList());
                       return TreePVector.from(cargos);
                   });
           return result;
       };
   }
```

Before the query side actually works, we need to work out a way to transform the events generated by the persistent entity into database tables. This is done with a `CassandraReadSideProcessor`:

```
public class CargoEventProcessor extends
CassandraReadSideProcessor<RegistrationEvent> {

  @Override
  public AggregateEventTag<RegistrationEvent> aggregateTag() {
    return RegistrationEventTag.INSTANCE;
  }

  @Override
  public CompletionStage<Optional<UUID>>
        prepare(CassandraSession session) {
    // TODO prepare statements, fetch offset
    return noOffset();
  }

  @Override
  public EventHandlers
      defineEventHandlers(EventHandlersBuilder builder) {
    // TODO define event handlers
    return builder.build();
  }

}
```

To make the events available for read-side processing, the events must implement the `aggregateTag` method of the `AggregateEvent` interface to define which events belong together. Typically, you define this `aggregateTag` on the top-level event type of a `PersistentEntity` class. Note that this is also used to create read-side views that span multiple `PersistentEntities`:

```
public class RegistrationEventTag {
    public static final AggregateEventTag<RegistrationEvent>
    INSTANCE = AggregateEventTag.of(RegistrationEvent.class);
}
```

Finally, the `RegistrationEvent` also needs to extend the `AggregateEvent<E>` interface. Now, we're ready to implement the remaining methods of the `CargoEventProcessor`.

Tables and prepared statements need to be created first. Further on, it has to be decided how to process existing entity events, which is the primary purpose of the `prepare` method. Each event is associated with a unique offset, a time-based UUID. The offset is a parameter to the event handler for each event and should typically be stored so that it can be retrieved with a select statement in the `prepare` method. You can use the `CassandraSession` to get the stored offset.

Composing all of the described asynchronous `CompletionStage` tasks for this example look like this:

```
@Override
   public CompletionStage<Optional<UUID>>
          prepare(CassandraSession session) {
       return
          prepareCreateTables(session).thenCompose(a ->
          prepareWriteCargo(session).thenCompose(b ->
          prepareWriteOffset(session).thenCompose(c ->
          selectOffset(session))));
   }
```

Starting with the table preparation for the read-side is simple. Use the `CassandraSession` to create the two tables:

```
private CompletionStage<Done>
   prepareCreateTables(CassandraSession session) {
     return session.executeCreateTable(
       "CREATE TABLE IF NOT EXISTS cargo ("
       + "cargoId text, name text, description text,"
       + "owner text, destination text,"
       + "PRIMARY KEY (cargoId, destination))")
       .thenCompose(a -> session.executeCreateTable(
       "CREATE TABLE IF NOT EXISTS cargo_offset ("
         + "partition int, offset timeuuid, "
         + "PRIMARY KEY (partition))"));
     }
```

The same can be done with the prepared statements. This is the example for inserting new cargo into the cargo table:

```
    private CompletionStage<Done>
       prepareWriteCargo(CassandraSession session) {
           return session
              .prepare("INSERT INTO cargo"
              + "(cargoId, name, description, "
              + "owner,destination) VALUES (?, ?,?,?,?)")
              .thenApply(ps -> {
               setWriteCargo(ps);
               return Done.getInstance();
           });
    }
```

The last missing piece is the event handler. Whenever a `CargoRegistered` event is received, it should be persisted into the table. The events are processed by event handlers that are defined in the method `defineEventHandlers`, one handler for each event class. A handler is a `BiFunction` that takes the event and the offset as parameters and returns zero or more bound statements that will be executed before processing the next event.

```
    @Override
    public EventHandlers
        defineEventHandlers(EventHandlersBuilder builder) {
          builder.setEventHandler(CargoRegistered.class,
          this::processCargoRegistered);
          return builder.build();
    }

    private CompletionStage<List<BoundStatement>>
    processCargoRegistered(CargoRegistered event, UUID offset) {
            // bind the prepared statement
            BoundStatement bindWriteCargo = writeCargo.bind();
            // insert values into prepared statement
            bindWriteCargo.setString("cargoId",
                    event.getCargo().getId());
            bindWriteCargo.setString("name",
                    event.getCargo().getName());
            bindWriteCargo.setString("description",
            event.getCargo().getDescription());
            bindWriteCargo.setString("owner",
                    event.getCargo().getOwner());
            bindWriteCargo.setString("destination",
                    event.getCargo().getDestination());
            // bind the offset prepared statement
            BoundStatement bindWriteOffset =
                        writeOffset.bind(offset);
            return completedStatements(
                    Arrays.asList(bindWriteCargo,
                        bindWriteOffset));
    }
```

In this example we add one row to the cargo table and update the current offset for each `RegistrationEvent`.

> It is safe to keep state in variables of the enclosing class and update it from the event handlers. The events are processed sequentially, one at a time. An example of such state could be values for calculating a moving average.
>
> If there is a failure when executing the statements the processor will be restarted after a backoff delay. This delay is increased exponentially in the case of repeated failures.

There is another tool that can be used if you want to do something else with the events other than updating tables in a database. You can get a stream of the persistent events with the `eventStream` method of the `PersistentEntityRegistry`.

You have already seen the service implementation that queries the database. Let's try out the API endpoint and get a list of all the registered cargo in the system by curling it:

```
curl http://localhost:9000/api/registration/all

[
   {
      "id":"522871",
      "name":"TEST",
      "description":"TEST",
      "owner":"TEST",
      "destination":"TEST"
   },
   {
      "id":"623410",
      "name":"SECOND",
      "description":"SECOND",
      "owner":"SECOND",
      "destination":"SECOND"
   }
]
```

Consuming Services

We've seen how to define service descriptors and implement them, but now we need to consume services. The service descriptor contains everything Lagom needs to know to invoke a service. Consequently, Lagom is able to implement service descriptor interfaces for you.

The first thing necessary to consume a service is to bind it, so that Lagom can provide an implementation for your application to use. We've done that with the service before. Let's add a client call from the *shipping-impl* to the *registration-api* and validate a piece of cargo before we add a leg in the *shipping-impl*:

```
public class ShippingServiceModule extends
      AbstractModule implements ServiceGuiceSupport {
    @Override
    protected void configure() {
        bindServices(serviceBinding(ShippingService.class,
        ShippingServiceImpl.class));
        bindClient(RegistrationService.class);
    }
}
```

Make sure to also add the dependency between both projects in the *build.sbt* file by adding `.dependsOn(registrationApi)` to the *shipping-impl* project.

Having bound the client, you can now have it injected into any Lagom component using the `@Inject` annotation. In this example it is injected into the *ShippingServiceImpl*:

```
public class ShippingServiceImpl implements ShippingService {
    private final RegistrationService registrationService;

    @Inject
    public ShippingServiceImpl(PersistentEntityRegistry
    persistentEntityRegistry,
    RegistrationService registrationService) {
        this.registrationService = registrationService;
        //...
    }
```

The service can be used to validate a cargo ID in the `shipping-impl` before adding a leg:

```
    @Override
        public ServiceCall<String, Leg, Done> addLeg() {
            return (id, request) -> {
                CompletionStage<Cargo> response =
                registrationService.getRegistration()
                  .invoke(request.getCargoId(),
                     NotUsed.getInstance());
                PersistentEntityRef<ShippingCommand>
                itinerary = persistentEntityRegistry
                        .refFor(ItineraryEntity.class, id);
                return itinerary.ask(AddLeg.of(request));
            };
        }
```

All service calls with Lagom service clients are by default using circuit breakers. Circuit breakers are used and configured on the client side, but the granularity and configuration identifiers are defined by the service provider. By default, one circuit breaker instance is used for all calls (methods) to another service. It is possible to set a unique circuit breaker identifier for each method to use a separate circuit breaker instance for each method. It is also possible to group related methods by using the same identifier on several methods. You can find more information about how to configure the circuit breaker in the Lagom documentation.

CHAPTER 5

Migration and Integration

You never change things by fighting the existing reality. To change something, build a new model that makes the existing model obsolete.

—R. Buckminster Fuller

One of the most pressing concerns that come bundled with every new technology stack is how to best integrate with existing systems. With the fundamental switch from monolithic to distributed applications, the integrity of a migration of existing code or functionality will have to be considered. The need to rearchitect and redesign existing systems to adopt the principles of the new world is undoubtedly the biggest challenge.

Migration Approaches

While Lagom and the reactive programing model is clearly favoring the greenfield approach, nothing is stopping you from striving for a brownfield migration. You have three different ways to get started with this.

Selective Improvements

The most risk-free approach is using selective improvements. After the initial assessment, you know exactly which parts of the existing application can take advantage of a microservices architecture. By scraping out those parts into one or more services and adding the necessary glue to the original application, you're able to scale out the microservices. There are many advantages to this approach. While doing archaeology on the existing system, you'll receive a very good overview of the parts that would make for ideal candidates. And while moving out individual services one at a time, the team has a fair chance to adapt to the new development methodology and make its first experience with the technology stack a positive one.

The Strangler Pattern

Comparable but not equal is the second approach where you run two different systems in parallel. First coined by Martin Fowler as the StranglerApplication, the refactor/extraction candidates move into a completely new technology stack, and the existing parts of the applications remain untouched. A load balancer or proxy decides which requests need to reach the original application and which go to the new parts. There are some synchronization issues between the two stacks. Most importantly, the existing application can't be allowed to change the microservices' databases.

Big Bang: Refactor an Existing System

In very rare cases, complete refactoring of the original application might be the right way to go. It's rare because enterprise applications will need ongoing maintenance during the complete refactoring. What's more, there won't be enough time to make a complete stop for a couple of weeks—or even months, depending on the size of the application—to rebuild it on a new stack. This is the least recommended approach because it carries comparably high business risks.

This ultimately leads to the question of how to integrate the old world into the new.

Legacy Integration

Even if the term "legacy" has an old and outdated touch to it, I use it to describe everything that is not already in a microservices-based architecture. You have existing business logic in your own applications. There are libraries and frameworks that provide access to some proprietary system and there may be host systems that need to be integrated. More specifically, this is everything that exists and still needs to function while you're starting to modernize your applications. There are many ways to successfully do this. Please keep in mind, though, that this isn't an architectural discussion of enterprise integration but a technical assessment of the interface technologies and how you can use them. Figure 5-1 gives a high-level overview about integration technologies and how to use them with Lagom.

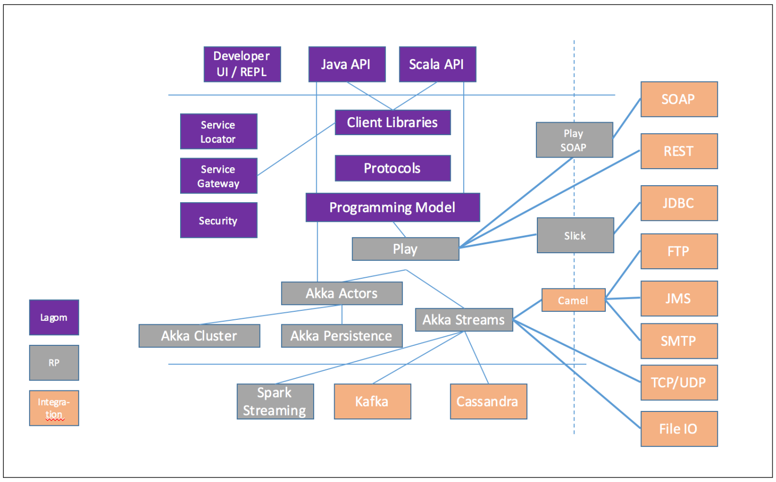


Figure 5-1. Technology integration with Lagom

Lagom builds upon and integrates very tightly with both Akka and Play. You can use Akka from your Lagom service implementations by injecting the current `ActorSystem` into it or directly into persistent entities with ordinary dependency injection. Details about the integration can be found in the Lagom documentation.

SOAP-Based Services and ESBs

Simple Object Access Protocol (SOAP) is heavily used in enterprise environments that already use an enterprise service bus (ESB). Play SOAP allows a Play application to make calls on a remote web service using SOAP. It provides a reactive interface for doing so, making HTTP requests asynchronously and returning promises/futures of the result. Keep in mind that Play SOAP builds on the JAX-WS spec, but implements the asynchronous method handling differently.

REST-Based Services

Play supports HTTP requests and responses with a content type of JSON by using the HTTP API in combination with the JSON library. JSON is mapped via the Jackson library.

Java Database Access

Because Lagom applications can be written in Java, you are free to bundle every JDBC driver you feel is necessary to access existing database systems. You could also use libraries that implement existing specifications like JPA. But it is not a good fit in general. Another option would be to use the Play JPA integration.

As soon as the Scala API for Lagom is available, Slick is the best option to choose. Slick is a modern database query and access library for Scala. It allows you to work with stored data almost as if you were using Scala collections while at the same time giving you full control over when a database access happens and which data is transferred. You can write your database queries in Scala instead of SQL, thus profiting from the static checking, compile-time safety, and compositionality of Scala. Slick features an extensible query compiler that can generate code for different backends.

TCP/UDP and File IO

Akka Streams is an implementation of the Reactive Streams specification on top of the Akka toolkit that uses an actor-based concurrency model. Using it this way, you can connect to almost all stream-based sources. A detailed explanation can be found in the Akka Streams Cookbook. This is a collection of patterns to demonstrate various usage of the Akka Streams API by solving small targeted problems in the format of "recipes."

JMS, SMPT, and FTP

The akka-camel module allows untyped actors to receive and send messages over a great variety of protocols and APIs. In addition to the native Scala and Java actor API, actors can now exchange messages with other systems over a large number of protocols and APIs, such as HTTP, SOAP, TCP, FTP, SMTP, or JMS, to mention a few. At the moment, approximately 80 protocols and APIs are supported.

Technically, there are more ways to integrate and talk to the legacy world. This chapter was written to give you a solid first overview of the most important protocols and technologies.

About the Author

Markus Eisele is a developer advocate at Lightbend. He has been working with Java EE servers from different vendors for more than 14 years, and gives presentations on his favorite topics at leading international Java conferences. He is a Java Champion, former Java EE Expert Group member, Java community leader of German DOAG, and founder of JavaLand. He is excited to educate developers about how microservices architectures can integrate and complement existing platforms, as well as how to successfully build resilient applications with Java. He is also the author of *Modern Java EE Design Patterns* by O'Reilly. You can follow more frequent updates on his Twitter feed and blog.